

Козин Р.Г.

**ПРОГРАММИРОВАНИЕ АЛГОРИТМОВ ЧИСЛЕННЫХ МЕТОДОВ
ЛИНЕЙНОЙ АЛГЕБРЫ**

учебно-методическое пособие

2014

Оглавление

Введение	4
Глава 1. Нормы вектора и матриц, мера обусловленности матрицы.....	5
1.1. Норма вектора	5
1.2. Норма матрицы	6
1.3. Мера обусловленности и ранг матрицы	13
Глава 2. Методы решения систем линейных уравнений.....	27
2.1. Итерационные методы решения систем линейных уравнений.....	27
2.2. Прямые методы решения системы линейных уравнений	59
2.3. Метод регуляризации для решения плохо обусловленных систем.....	124
2.4. Решение систем с прямоугольными матрицами.....	127
Глава 3. Методы решения задачи на собственные значения	138
3.1. Собственные значения и собственные вектора матрицы	138
3.2. Решение частичной проблемы собственных чисел для симметричной матрицы.....	140
3.3. Решение задачи на собственные значения методом Данилевского	146
3.4. Модифицированный метод Данилевского.....	154
3.5. Метод Крылова	158
3.6. Определение собственных векторов в общем случае	163
3.7. Метод вращения.....	165
Глава 4. Алгоритмы для работы с полиномами	174
4.1. Подсчет значения полинома и деление на множители	174
4.2. Локализация корней полинома.....	176
4.3. Метод парабол для нахождения всех корней полинома.....	191
Задания для практикума	204
Список литературы.....	206
Приложение. Некоторые сведения о математическом пакете Maxima.....	207

Введение

Методы решения задач линейной алгебры являются основополагающими в прикладной математике, поскольку большинство реальных вычислительных задач из других разделов математики, как правило, сводятся к аппроксимирующими их задачам линейной алгебры. Кроме того, результаты линейной алгебры непосредственно используются при вычислении конкретных прикладных величин.

Наиболее востребованными задачами линейной алгебры являются следующие:

1. Найти вектор x с координатами $x_i, i = 1 \div n$, являющейся решение системы линейных уравнений

$$Ax = b$$

где $A (A_{ij}, i, j = 1 \div n)$ – исходная матрица; $b (b_i, i = 1 \div n)$ - заданный вектор правых частей системы.

2. Для заданной матрицы A вычислить определитель - $\det(A)$.

3. Найти матрицу A^{-1} , обратную заданной A , для которой выполняется матричное равенство $A^{-1}A = E$, где E - единичная матрица ($E_{ij} = \delta_{ij}$).

4. Для заданной матрицы A определить собственные числа λ_i и соответствующие им собственные векторы x_i , которые удовлетворяют следующему матричному уравнению $Ax = \lambda x$.

5. Нахождение корней полинома:

$$P_n(x) = \sum_{i=0}^n a_i x^{n-i} = a_0 x^n + a_1 x^{n-1} + \dots + a_n = 0,$$

выделение из полинома линейных и квадратичных множителей.

Глава 1. Нормы вектора и матриц, мера обусловленности матрицы

Введем понятия норм вектора и матрицы, которые используются при «интегральной» оценке результатов операций, выполняемых над этими распределенными объектами.

1.1. Норма вектора

Норма вектора $\|x\|$ – положительная скалярная величина, вычисляемая через его компоненты $x_i, i = 1 \div n$ и явно или косвенно характеризующая его длину. Как и длина вектора, она должна удовлетворять следующим соотношениям:

$$\|x\| \geq 0, \quad \|cx\| = |c|\|x\|, \quad \|x + y\| \leq \|x\| + \|y\| \quad (1.1)$$

Приведем три способа определения нормы вектора.

1. Кубическая норма

$$\|x\|_{куб} = \max_i |x_i| \quad (1.2)$$

2. Октаэдрическая норма

$$\|x\|_{окт} = \sum_i |x_i| \quad (1.3)$$

3. Сферическая норма

$$\|x\|_{сф} = \sqrt{\sum_i |x_i|^2} = \sqrt{(x, x)} \quad (1.4)$$

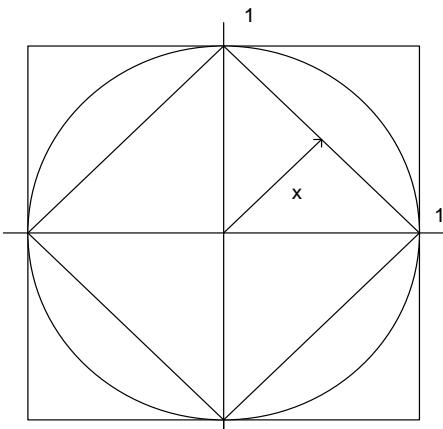
Концы множества векторов, нормы которых удовлетворяют равенствам $\|x\|_{куб} = 1, \|x\|_{окт} = 1$ и $\|x\|_{сф} = 1$, нарисуют в n -мерном пространстве соответственно поверхности n -мерного куба, n -мерного октаэдра и n -мерной сферы. Это обстоятельство и объясняет их название.

Легко проверить, что все указанные нормы удовлетворяют трем соотношениям (1.1).

Из рисунка (на котором для двумерного случая построены куб, октаэдр и сфера, соответствующие уравнениям: $\|x\|_{куб} = 1, \|x\|_{окт} = 1$ и $\|x\|_{сф} = 1$) видно, что для любого вектора x между этими нормами выполняются следующие соотношения

$$\|x\|_{окт} \geq \|x\|_{сф} \geq \|x\|_{куб}$$

Справедливость соотношения легко подтверждается на примере конкретного вектора $x = [1, 2, 3]$, для которого соответствующие нормы равны $\|x\|_{куб} = 3, \|x\|_{окт} = 6$ и $\|x\|_{сф} = \sqrt{1+4+9} = \sqrt{14} \approx 3,74$.



Замечание. Из равенства норм двух векторов не следует равенство самих векторов, в то время как равные вектора всегда имеют равные нормы.

Приведем алгоритмы вычисления двух норм $\|x\|_{куб}$ и $\|x\|_{окт}$ (поскольку они одновременно демонстрируют, как можно найти максимальное значение и сумму любой последовательности чисел)

1.

```

norm = 0
for i = 1, n
{
    buf = |xi|
    if buf > norm then norm = buf
}

```

2.

```

norm = 0
for i = 1, n
{
    norm = norm + |xi|
}

```

1.2. Норма матрицы

Норма матрицы $\|A\|$ – положительная скалярная величина, которая устанавливает соответствие между нормами исходного вектора $\|x\|$ и вектора, являющегося результатом действия на исходный вектор матрицы $\|Ax\|$. При этом в зависимости от критерия, используемого для нахождения нормы матрицы

$$\begin{aligned} \|Ax\| &\leq \|A\| \|x\| \\ \|A\| &= \sup_x \frac{\|Ax\|}{\|x\|} \end{aligned}$$

определяются два вида норм: согласованная и подчиненная (наименьшая из всех согласованных).

Выведем нормы матрицы подчиненные, соответственно, кубической, октаэдрической и сферической нормам вектора.

$$1. \|Ax\|_{куб} = \max_i \left| \sum_j A_{ij} x_j \right| \leq \max_i \sum_j |A_{ij}| |x_j| \leq \max_i |x_j| \max_i \sum_j |A_{ij}| = \|x\|_{куб} \max_i \sum_j |A_{ij}|$$

Отсюда

$$\|A\|_{куб} = \max_i \sum_j |A_{ij}| \quad (1.5)$$

2.

$$\begin{aligned}\|Ax\|_{okm} &= \sum_i \left| \sum_j A_{ij} x_j \right| \leq \sum_i \sum_j |A_{ij}| |x_j| = \sum_j |x_j| \sum_i |A_{ij}| \leq \max_j \sum_i |A_{ij}| \sum_j |x_j| = \\ &= \|x\|_{okm} \max_j \sum_i |A_{ij}|\end{aligned}$$

Отсюда

$$\|A\|_{okm} = \max_j \sum_i |A_{ij}| \quad (1.6)$$

3. $\|Ax\|_{cph}^2 = (Ax, Ax) = (x, A^T Ax) = \dots$

Матрица $A^T A$ симметрична и положительно определенна (т.к. $(A^T A)^T = A^T (A^T)^T = A^T A$ и $(A^T Ax, x) = (Ax, Ax) > 0$). Поэтому она имеет полную линейно-независимую систему собственных векторов $x^i, i = 1 \div n$, а ее собственные числа действительны и положительны: $\Lambda_1 \geq \Lambda_2 \geq \dots \geq \Lambda_n > 0$. Представим произвольный вектор x в виде разложения по этим векторам $x = \sum_i c_i x^i$ и подставим его в верхнее скалярное произведение

$$= (\sum_i c_i x^i, \sum_i c_i A^T Ax^i) = (\sum_i c_i x^i, \sum_i c_i \Lambda_i x^i) \leq \Lambda_1 (\sum_i c_i x^i, \sum_i c_i x^i) = \Lambda_1 (x, x) = \Lambda_1 \|x\|_{cph}^2$$

Отсюда

$$\|A\|_{cph} = \sqrt{\Lambda_1} \quad (1.7)$$

Замечание. Поскольку для симметричной матрицы $A^T A = A^2$, то $\Lambda_1 = \lambda_1^2$ и $\|A\|_{cph} = |\lambda_1|$.

В качестве примера нормы матрицы согласованной со сферической нормой вектора приведем евклидову норму

$$\|A\|_{eek} = \sqrt{\sum_i \sum_j A_{ij}^2}$$

Она связана со сферической нормой следующим образом $\|A\|_{cph} \leq \|A\|_{eek}$. Это

соотношение подтверждается примерами, представленными на рис.1.1 и 1.2.

Замечание. Приведенные ниже неравенства

$$\|A\|_{kyb} = \max_i \sum_j |A_{ij}| \leq n \max_{ij} |A_{ij}|$$

$$\|A\|_{okm} = \max_j \sum_i |A_{ij}| \leq n \max_{ij} |A_{ij}|$$

$$\|A\|_{eek} = \sqrt{\sum_i \sum_j A_{ij}^2} \leq \sqrt{n^2 \max_{ij} A_{ij}^2} = n \max_{ij} |A_{ij}|$$

$$\Lambda_1 < \sum_i \Lambda_i = \sum_i (A^T A)_{ii} = \sum_i \sum_k A_{ki} A_{ki} \leq n^2 (\max_{ij} |A_{ij}|)^2$$

$$\|A\| \|x\| \geq \|Ax\| = |\lambda_1| \|x\| \Rightarrow \|A\| \geq |\lambda_1|$$

устанавливают нижнюю и верхнюю грани для всех указанных норм

$$|\lambda_1| \leq \|A\| \leq n \max_{i,j} |A_{ij}|,$$

где λ_1 - максимальное по модулю собственное число матрицы. Учтено, что сумма собственных значений матрицы равна ее следу.

На практике выход значения нормы матрицы за пределы этого интервала указывает на то, что норма матрицы определена неверно. Кроме того, эти границы позволяют грубо оценить само значение нормы матрицы.

В заключение приведем алгоритмы вычисления кубической нормы матрицы (для октаэдрической нормы в этом алгоритме необходимо поменять местами индексы в операторах цикла) и произведения матриц $B = A^T A$, которое используется при определении сферической нормы матрицы.

$$1. \|A\|_{куб} = \max_i \sum_j |A_{ij}|$$

norm = 0

for i = 1, n

{

sum = 0

for j = 1, n

{

$$\text{sum} = \text{sum} + |A_{ij}|$$

}

if sum > norm then norm = sum

}

$$2. B_{ij} = \sum_k A_{ik}^T A_{kj} = \sum_k A_{ki} A_{kj}; i = 1, n; j = 1, n$$

for i = 1, n

{

for j = 1, n

{

$$B_{ij} = 0$$

for k = 1..n

{

$$B_{ij} = B_{ij} + A_{ki} A_{kj}$$

}

}

}

На рис. 1.1. показана форма программы, позволяющей вычислять четыри введенные выше нормы матрицы. Для расчета сферической нормы необходимо предварительно умножить матрицу на транспонированную, а затем методом вращения найти все собственные числа полученной симметричной положительно определенной матрицы. Собственные числа этой матрицы формируются на месте ее диагональных компонент (см. рис. 1.1). Метод вращения описывается в параграфе 3.7.

Нахождение четырех норм и меры обусловленности произвольной матрицы

справка выход

Размерность матрицы [>1]	<input type="text" value="5"/>	<input checked="" type="checkbox"/> - сформировать случайную матрицу ($0 < A[i,j] < 1$)								
Исходная матрица - A		Матрица - $A(t)^*A$ или матрица с собств. значениями на диагонали								
i \ i	1	2	3	4	5	1	2	3	4	5
1	.4119427	.4034964	.0015281	.9268544	.6801113	.01112541
2	.7701329	.6381908	.0571494	.4183766	.4178859	.	.6169585	.	.	.
3	.3572853	.1628053	.5625905	.8516992	.7064801	.	.	.3853288	.	.
4	.4133167	.1667692	.2220246	.2969201	.787748678790054	.
5	.8562178	.9654464	.881352	.8403611	.53065561964962

Строка, столбец и задаваемое значение компоненты:

Требуемая точность (max $|A_{ij}| < \text{eps}$)

| Кубическая | | Сферическая | |
| Октаэдрическая | | Евклидова | |

Мера обусловленности матрицы

Рис. 1.1. Экранная форма программы, которая для введенной матрицы вычисляет нормы и меру обусловленности ($\nu(A) = \|A^{-1}\|^* \|A\|$). Результаты расчета подтверждают, что $\|A\|_{cf} \leq \|A\|_{евк}$ и для любой из указанных норм матрицы справедливо неравенство $\|A\| < n \max_{i,j} |A_{ij}| = 5 * 0.92685 = 4.63425$.

Дополнительно приведем листинг (рис. 1.2) функций и подпрограмм для математического пакета Maxima, которые дублируют вычисления, отображенные на рис. 1.1. Результаты их исполнения представлены на рис. 1.3 и 1.4.

```
/* функции вычисляют нормы матрицы: кубическая, октаэдрическая и
евклидова */
nrm1(A):=lmax(create_list(sum(abs(A[i,j])),j,1,length(A)),i,1,length(A));
nrm2(A):=lmax(create_list(sum(abs(A[i,j])),i,1,length(A)),j,1,length(A));
```

```

nrm4(A):=sqrt(sum(sum(A[i,j]^2,j,1,length(A)),i,1,length(A)));

/* подпрограмма умножает транспонированную матрицу на исходную */
/* результат возвращает в матрице C2 */
Ct_C(C):=block
(
  [i,j,k,n,Ct], /* локальные переменные подпрограммы */
  n:length(C), /* определяем размерность матрицы */
  C2:zeromatrix(n,n), /* формируем нулевую матрицу, размерности n */
  Ct:transpose(C), /* трансформируем матрицу */
  for i thru n do /* три вложенных цикла */
    for j thru n do for k thru n do C2[i,j]:=C2[i,j]+Ct[i,k]*C[k,j]
);
/* подпрограмма находит положение максимального по модулю
внедиагонального элемента матрицы; результат возвращается в переменных:
lmax, kmax, Cmax, fi */
maxElem(C):=block
(
  [i,j,n,buf], /* локальные переменные */
  Cmax:-1,lmax:0, kmax:0, n:length(C),
  for i thru n-1 do
    for j:i+1 thru n do
    (
      buf:abs(C[i,j]),
      if buf>Cmax then (Cmax:buf,lmax:i,kmax:j)
    ),
    /* условный оператор */
    if C[lmax,lmax]=C[kmax,kmax] then fi:signum(C[lmax,kmax])%pi/4
    else fi:0.5*atan(2*C[lmax,kmax]/(C[lmax,lmax]-C[kmax,kmax]))
);
/* подпрограмма реализует метод вращения для симметричной матрицы */
/* iter, eps – число итераций и точность */
lmd(A,iter,eps):=block
(
  [i,j,n,it,buf], /* локальные переменные */
  n:length(A),maxIt:0,
  for it thru iter do
  (
    maxElem(A),
    if Cmax<eps then return(maxIt:it), /* возвращает число использованных

```

```

итераций */
sinfi:sin(fi), cosfi:cos(fi),
for i thru n do
(
  buf:A[i,lmax]*cosfi+A[i,kmax]*sinfi,
  A[i,kmax]:-A[i,lmax]*sinfi+A[i,kmax]*cosfi,
  A[i,lmax]:buf
),
for j thru n do
(
  buf:A[lmax,j]*cosfi+A[kmax,j]*sinfi,
  A[kmax,j]:-A[lmax,j]*sinfi+A[kmax,j]*cosfi,
  A[lmax,j]:buf
)
)
);

/* главная программа */
numer:true; /* выводить числа в десятичном формате */
fpprintprec:5; /* устанавливаем число цифр в выводимом числе = 5 */

D:matrix([1,2,3],[4,5,6],[7,8,9]); /* создаем матрицу */
DD:copy(D); /* сохраняем исходную матрицу */
Ct_C(D); /* вызываем ранее определенную функцию */
D:copy(C2);
str:"Список диагональных компонент матрицы D(T)*D = ";
L:create_list(D[i,i],i,1,length(D)); /* создаем список */
str:"След матрицы D(T)*D = ";
Sled:sum(D[i,i],i,1,length(D)); /* суммируем список */
lmd(D,1000,0.0000000001);
str:"Список собственных значений матрицы D(T)*D = ";
L:create_list(D[i,i],i,1,length(D));
str:"Сумма собственных значений матрицы D(T)*D = ";
Sled:sum(L[i],i,1,length(D));

print("Все нормы (включая евклидову) исходной матрицы D = ",
nrm1(DD),",",nrm2(DD),",",sqrt(lmax(L)),",",nrm3(DD));
/* здесь используется abs(), т.к. приближенное мин.собств.значение может
оказаться <0 */
L1:create_list(abs(D[i,i]),i,1,length(D));
Lmn:lmin(L1); Lmx:lmax(L); /* мин. и макс. собств. значения матрицы D(T)*D */
print("Мера обусловленности исходной матрицы = ",

```

```
sqrt(Lmx/Lmn));
```

Рис. 1.2. Листинг программы для Maxima.

Прокомментируем некоторые команды и операции Maxima, использованные в листинге:

: и =, < - операции присвоения и сравнения;

внутренние команды:

lmax() – определяет максимальное число в списке; **sum()** – суммирует элементы списка;

create_list() – создает список элементов; **matrix()** – формирует матрицу.

$$\begin{aligned} (\%o1) \quad nrm1(A) := lmax\left(\text{create_list}\left(\sum_{j=1}^{\text{length}(A)} |A_{i,j}|, i, 1, \text{length}(A) \right) \right) \\ (\%o2) \quad nrm2(A) := lmax\left(\text{create_list}\left(\sum_{i=1}^{\text{length}(A)} |A_{i,j}|, j, 1, \text{length}(A) \right) \right) \\ (\%o3) \quad nrm3(A) := \sqrt{\sum_{i=1}^{\text{length}(A)} \sum_{j=1}^{\text{length}(A)} A_{i,j}^2} \end{aligned}$$

Рис. 1.3. Расшифровка пакетом Maxima трех первых строчек листинга.

```

(%o10) ⎡ 1  2  3 ⎤
      ⎢ 4  5  6 ⎥
      ⎣ 7  8  9 ⎦
(%o11) done
(%o12) ⎡ 66   78   90 ⎤
      ⎢ 78   93   108 ⎥
      ⎣ 90   108  126 ⎦
(%o13) Список диагон.компонент матрицы D(T)*D =
(%o14) [ 66, 93, 126 ]
(%o15) След матрицы D(T)*D =
(%o16) 285
(%o17) 8
(%o18) Список собств.значений матрицы D(T)*D =
(%o19) [ 1.1414, -4.68723 10-15, 283.86 ]
(%o20) Сумма собств.значений матрицы D(T)*D =
(%o21) 285.0
Все нормы (включая евклидову) исходной матрицы D = 24, 18, 16.848, 16.882
(%o22) 16.882
(%o23) [ 1.1414, 4.68723 10-15, 283.86 ]
(%o24) 4.68723 10-15
(%o25) 283.86
Мера обусловленности исходной матрицы D = 2.4609 108

```

Рис. 1.4. Результаты выполнения главной программы листинга. Видно, что след матрицы $D(T)^*D$, как и положено, равен сумме собственных значений матрицы. Кроме того: сферическая и евклидова нормы матрицы, соответственно, равны: $16.848 < 16.882$ и для любой из указанных норм матрицы выполняется неравенство $\|A\| < n \max_{i,j} |A_{ij}| = 4 * 9 = 36$.

1.3. Мера обусловленности и ранг матрицы

В зависимости от того, какие соотношения устанавливает матрица A между входными ($\delta A, \delta b$) и выходными (δx) погрешностями в системе $Ax=b$, матрицы делятся на «плохие» и «хорошие» или в общепринятой терминологии на плохо и хорошо обусловленные. Качественно это иллюстрируют рис. 1.5 и 1.6, на которых показаны области разброса решений для системы двух уравнений при одинаковых погрешностях в правых частях

$$F1(x) \equiv A_{11}x_1 + A_{12}x_2 = b_1$$

$$F2(x) \equiv A_{21}x_1 + A_{22}x_2 = b_2$$

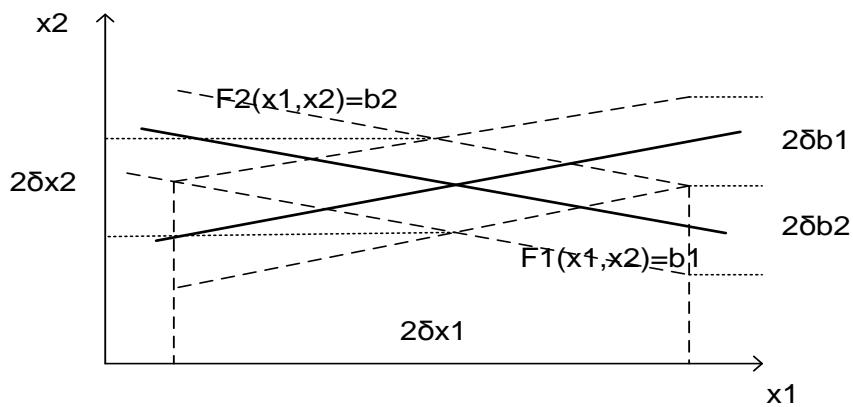


Рис. 1.5. Область разброса плохо обусловленной системы

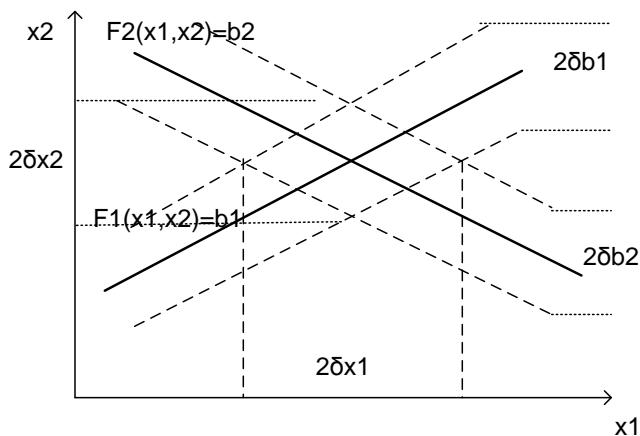


Рис. 1.6. Область разброса хорошо обусловленной системы

Видно, что чем меньше угол между прямыми $F_1(x) = b_1$ и $F_2(x) = b_1$, тем больше ошибки в решении системы при одних и тех же величинах погрешностей в правых частях, а, следовательно, тем хуже матрица. Качественно это можно объяснить тем, что с уменьшением угла возрастает линейная зависимость между строками матрицы и поэтому справедлива следующая цепочка соотношений:

$$\det(A) \rightarrow 0, A^{-1} \approx \frac{1}{\det(A)}, \delta x = A^{-1} \delta b \approx \frac{\delta b}{\det(A)} \rightarrow \infty.$$

После этих качественных рассуждений перейдем к выводу количественной оценке меры обусловленности матрицы, которая устанавливает количественную зависимость между относительной погрешностью решения $\frac{\|\delta x\|}{\|x\|}$ системы линейных уравнений $Ax = b$ и относительными погрешностями исходных параметров системы $\frac{\|\delta A\|}{\|A\|}$ и $\frac{\|\delta b\|}{\|b\|}$.

Для этого запишем два матричных уравнения (исходное и возмущенное погрешностями)

$$Ax = b$$

$$(A + \delta A)(x + \delta x) = (b + \delta b) \Rightarrow Ax + \delta Ax + A\delta x + \delta A\delta x = b + \delta b$$

Вычитая первое уравнение из второго и отбрасывая слагаемое второго порядка малости относительно δ , получим

$$A\delta x = \delta b - \delta Ax \text{ или } \delta x = A^{-1}(\delta b - \delta Ax)$$

Отсюда можно построить следующую цепочку неравенств (учтено очевидное соотношение $\|A\| \|x\| \geq \|Ax\| = \|b\|$)

$$\begin{aligned} \|\delta x\| &\leq \|A^{-1}\| (\|\delta b\| + \|\delta A\| \|x\|) \\ \frac{\|\delta x\|}{\|x\|} &\leq \|A^{-1}\| \left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\| \|A\|}{\|A\|} \right) \leq \|A\| \|A^{-1}\| \left(\frac{\|\delta b\|}{\|b\|} + \frac{\|\delta A\|}{\|A\|} \right) \end{aligned} \quad (1.8)$$

Последнее неравенство устанавливает для системы $Ax = b$ количественное соотношение между входными и выходными относительными погрешностями. Присутствующий в нем коэффициент пропорциональности

$$\nu(A) = \|A\| \|A^{-1}\| \quad (1.9)$$

называется **мерой обусловленности** матрицы A . Она изменяется в интервале $1 \leq \nu(A) \leq \infty$.

С мерой обусловленности тесно связано понятие **ранга** матрицы. Ранг матрицы равен размерности ее максимальной квадратной невырожденной подматрицы с определителем отличным от нуля. Если ранг матрицы меньше ее размерности, то определитель матрицы равен нулю и матрица является вырожденной. Последнее косвенно указывает на то, что математическая модель, для которой получена система линейных уравнений, составлена с ошибками.

Для определения ранга матрицы обычно используется алгоритм метода исключения Гаусса с выбором ведущего элемента (подробно описан в разделе 2.2.1). Ниже приведены примеры вычисления ранга произвольной матрицы с помощью пакета Maxima. В первом случае используются встроенные команды пакета (рис.1.7), а во втором - подпрограмма (рис.1.9), реализующая алгоритм Гаусса. Результаты их выполнения показаны на рис. 1.8 и 1.10, соответственно.

```
numer:true;
fpprintprec:5;
M:matrix([1,2,3,4],[5,6,7,8],[2,4,6,8],[5,6,7,8]);
print("Исходная матрица = ",M);
print("Определитель матрицы = ",determinant(M));
print("Ранг матрицы = ",rank(M));
```

Рис.1.7. Определение ранга произвольной матрицы с помощью встроенных команд Maxima.

Исходная матрица =

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 2 & 4 & 6 & 8 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

Определитель матрицы = 0

Ранг матрицы = 2

Рис.1.8. Результаты выполнения встроенных команд

```

kill(all);
/* находим максимальный элемент в подстолбце матрицы, */
/* переставляем строки и вычисляем определитель */
maxElem(A,k):=block
(
    [buf,Amax,i,j,n], /* задание локальных переменных */
    n:length(A),
    Amax:abs(A[k,k]),kmax:k,
    for i:k+1 thru n do
    (
        buf:abs(A[i,k]),
        if buf>Amax then (Amax:buf,kmax:i)
    ),
    if Amax<1.e-10 then return(det:0),
    if kmax#k then
    (
        for j:k thru n do
        (
            buf:A[k,j],A[k,j]:A[kmax,j],A[kmax,j]:buf
        ),
        det:-det, m:m+1
    ),
    det:det*A[k,k]
);
/* подпрограмма реализует алгоритм метода исключения Гаусса */
/* с выбором ведущего элемента */
Rang(A):=block
(
    [i,j,n,buf],
    n:length(A),
    rang:0,det:1,m:0,
    for k thru n-1 do
    (
        maxElem(A,k),

```

```

if abs(det)=0 then return(det),
rang:rang+1,
for i:k+1 thru n do
(
  buf:A[i,k]/A[k,k],
  for j:k thru n do A[i,j]:=A[i,j]-buf*A[k,j]
)
),
if abs(A[n,n])<1.e-10 then det:=0 else (det:=det*A[n,n],rang:=rang+1)
);
/* главная программа */
numer:true;
fpprintprec:5;
/* пример вырожденной матрицы */
M:matrix([1,2,3,4],[5,6,7,8],[2,4,6,8],[5,6,7,8]);
Rang(M);
print("Определитель матрицы = ",det);
print("Ранг матрицы = ",rang);
print("Число перестановок строк = ",m);
print("Приобразованная матрица =",M);
/* пример «хорошей» матрицы */
R:matrix([1,2,3,5,6],[2,4,6,7,8],[7,8,9,10,11],[2,3,6,7,9],[8,6,4,1,3]);
print("Исходная матрица =",R);
Rang(R);
print("Определитель матрицы = ",det);
print("Ранг матрицы = ",rang);
print("Число перестановок строк = ",m);
print("Приобразованная матрица =",R);

```

Рис.1.9. Определение ранга произвольной матрицы с использованием подпрограммы Rang(A), реализующей алгоритм метода исключения Гаусса с выбором максимального элемента.

Случай 1. «плохая матрица».

Исходная матрица =

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 2 & 4 & 6 & 8 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

Определитель матрицы = 0
Ранг матрицы = 2
Число перестановок строк = 2

$$\text{Приведенная матрица} = \begin{bmatrix} 5 & 6 & 7 & 8 \\ 0.0 & 1.6 & 3.2 & 4.8 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0 & -0.0 & -0.0 & -0.0 \end{bmatrix}$$

Случай 2. «хорошая» матрица

$$\text{Исходная матрица} = \begin{bmatrix} 1 & 2 & 3 & 5 & 6 \\ 2 & 4 & 6 & 7 & 8 \\ 7 & 8 & 9 & 10 & 11 \\ 2 & 3 & 6 & 7 & 9 \\ 8 & 6 & 4 & 1 & 3 \end{bmatrix}$$

Определитель матрицы = 112.0

Ранг матрицы = 5

Число перестановок строк = 3

$$\text{Приведенная матрица} = \begin{bmatrix} 8 & 6 & 4 & 1 & 3 \\ 0.0 & 2.75 & 5.5 & 9.125 & 8.375 \\ 0.0 & 0.0 & 2.0 & 1.7727 & 3.6818 \\ 0.0 & 0.0 & 0.0 & -1.5455 & -0.364 \\ 0.0 & 0.0 & 0.0 & 0.0 & 1.6471 \end{bmatrix}$$

Рис.1.8. Результаты выполнения подпрограммы для двух исходных матриц. Видно, что ранг матрицы равен числу не нулевых диагональных компонент в приведенной матрице.

Замечание. Если учесть очевидное соотношение $(A^T A)^{-1} x = \frac{1}{\Lambda} x$, где x, Λ - собственный вектор и соответствующее собственное число матрицы $(A^T A)$, то, используя определение сферической нормы матрицы (1.7), для меры обусловленности матрицы можно записать выражение, не требующее вычисления обратной матрицы

$$\nu(A) = \sqrt{\frac{\Lambda_1}{\Lambda_n}} \quad (1.10)$$

Для симметричной матрицы оно перепишется следующим образом

$$\nu(A) = \sqrt{\left| \frac{\lambda_1}{\lambda_n} \right|}$$

где λ_1, λ_n - максимальное и минимальное собственные числа исходной симметричной матрицы A .

Примеры вычисления меры обусловленности в соответствии с формулой (1.10) приведены на рис. 1.1 и 1.4.

Пример.

Пусть исходная и возмущенная системы имеют вид

$$\begin{aligned} x_1 - 1,001x_2 &= 1 & x_1 - 1,0015x_2 &= 1 \\ -1,001x_1 + x_2 &= 2 & -1,0015x_1 + x_2 &= 2 \end{aligned}$$

Из характеристического уравнения

$$[A - \lambda E] = \begin{bmatrix} 1 - \lambda & -1,001 \\ -1,001 & 1 - \lambda \end{bmatrix} \equiv (1 - \lambda)^2 + 1,002 = 0$$

легко находим собственные числа $\lambda_1 = 2,001$, $\lambda_2 = -0,001$ и меру обусловленности исходной симметричной матрицы $\nu(A) = \left| \frac{\lambda_1}{\lambda_2} \right| = 2001$.

Решения исходной и возмущенной систем, соответственно, равны $x_1 = -1500,25$, $x_2 = -1499,75$ и $x_1 = -1000,25$, $x_2 = -999,75$.

В итоге легко определяются следующие сферические нормы векторов и матриц

$$(т.к.) [\delta A - \lambda E] = \begin{bmatrix} -\lambda & -0,0005 \\ -0,0005 & -\lambda \end{bmatrix} \equiv (\lambda)^2 - (0,0005)^2 = 0 \rightarrow \lambda_{1,2}(\delta A) = 0,0005$$

$$\|A\| \approx 2,001; \|\delta A\| \approx 0,0005; \|x\| \approx 1500\sqrt{2}; \|\delta x\| \approx 500\sqrt{2}$$

После этого неравенство (1.8) записывается в виде

$$\frac{\|\delta x\|}{\|x\|} = 0,333 \leq \nu(A) \frac{\|\delta A\|}{\|A\|} = 2001 * 0,00025 = 0,5$$

Видно, что оно благополучно удовлетворяется.

В заключение приведем листинг программы пакета Maxima, рассчитывающей меру обусловленности случайной матрицы A с использованием различных матричных норм: кубической, октаэдрической, евклидовой и сферической (рис.1.11). Первые три мера вычисляются с использованием обратной матрицы, последняя – через собственные значения матрицы $A^T * A$, которые определяются методом вращения. Выборочные результаты выполнения этой программы представлены на рис. 1.12.

```
kill(all);
/* функции вычисляют нормы матрицы: кубическая, октаэдрическая и
евклидова */
nrm1(A):=lmax(create_list(sum(abs(A[i,j])),j,1,length(A)),i,1,length(A));
nrm2(A):=lmax(create_list(sum(abs(A[i,j])),i,1,length(A)),j,1,length(A));
nrm3(A):=sqrt(sum(sum(A[i,j]^2,j,1,length(A)),i,1,length(A)));

/* п.п умножает транспонированную матрицу на исходную */
/* результат возвращает в матрице A2 */
At_A(A):=block
```

```

(
  [i,j,k,n,At], /* локальные переменные */
  n:length(A), A2:zeromatrix(n,n), At:transpose(A),
  for i thru n do
    for j thru n do for k thru n do A2[i,j]:=A2[i,j]+At[i,k]*A[k,j]
);
/* находит положение максимального по модулю внедиагонального элемента
матрицы */
/* возвращает lmax, kmax, Amax и угол fi */
maxElem(A):=block
(
  [i,j,n,buf], /* локальные переменные */
  Amax:-1,lmax:0, kmax:0, n:length(A),
  for i thru n-1 do
    for j:i+1 thru n do
    (
      buf:abs(A[i,j]),
      if buf>Amax then (Amax:buf,lmax:i,kmax:j)
    ),
    if A[lmax,lmax]=A[kmax,kmax] then fi:signum(A[lmax,kmax])*%pi/4
    else fi:0.5*atan(2*A[lmax,kmax]/(A[lmax,lmax]-A[kmax,kmax]))
);
/* метод вращения для симметричной матрицы */
lmd(A,iter,eps):=block
(
  [i,j,n,it,buf], /* локальные переменные */
  n:length(A),maxIt:0,
  for it thru iter do
  (
    maxElem(A),
    if Amax<eps then return(maxIt:it),
    sinfi:sin(fi), cosfi:cos(fi),
    for i thru n do
    (
      buf:A[i,lmax]*cosfi+A[i,kmax]*sinfi,
      A[i,kmax]:-A[i,lmax]*sinfi+A[i,kmax]*cosfi,
      A[i,lmax]:=buf
    ),
    for j thru n do
    (

```

```

buf:A[lmax,j]*cosfi+A[kmax,j]*sinf,
A[kmax,j]:-A[lmax,j]*sinf+A[kmax,j]*cosfi,
A[lmax,j]:buf
)
)
);
/* формируем случайную матрицу A размерности m*/
rnd(m):=block
(
A:zeromatrix(m,m), /* генерирует нулевую матрицу
for i thru m do
  for j thru m do A[i,j]:=(random(2.0)-1) /* случ.числа из диапозона [-1,1] */
);

/* --- главная программа --- */
numer:true;
fpprintprec:5; /* в выводимом числе 5 цифр, не считая точки */

rnd(5); /* формируем случайную матрицу */
print("Исходная случайная матрица = ",A);
print("Обратная матрица = ",B:invert(A)); /* внутренняя команда пакета */
print("Мера обусловленности для кубич., октаэдр., евклид.норм = ",
mu_kub:nrm1(A)*nrm1(B)," , ",mu_okt:nrm2(A)*nrm2(B)," ,
",mu_evk:nrm3(A)*nrm3(B));

At_A(A);A:copy(A2);
lmd(A,1000,0.00000000001);
print("Число использ.итераций и матрица с собст.значениями матрицы A(T)*A:
",
maxIt," ",A);
L:create_list(A[i,i],i,1,length(A)); /* создаем два списка собств.значений
матрицы A(T)*A */
L1:create_list(abs(A[i,i]),i,1,length(A));
print("Мера обусловленности для сферич.нормы =",
mu_sfer:sqrt(lmax(L)/lmin(L)));

```

Рис. 1.11. Листинг программы пакета Maxima, рассчитывающий меру обусловленности случайной матрицы с использованием различных матричных норм.

$$\text{Исходная случайная матрица} = \begin{bmatrix} 0.828 & -0.61 & -0.0352 & -0.289 & -0.496 \\ -0.169 & 0.225 & 0.684 & 0.0112 & 0.357 \\ -0.927 & 0.251 & 0.987 & -0.917 & 0.265 \\ 0.186 & 0.309 & -0.0586 & -0.896 & -0.995 \\ -0.369 & 0.72 & -0.916 & 0.697 & -0.534 \end{bmatrix}$$

$$\text{Обратная матрица} = \begin{bmatrix} -1.2999 & 0.738 & -1.6472 & 1.1425 & -1.2481 \\ -2.194 & 0.926 & -1.4643 & 1.5171 & -0.9 \\ 1.7789 & 1.6751 & 0.778 & -0.597 & 0.967 \\ 1.8509 & 1.1152 & 0.618 & -1.1627 & 1.5019 \\ -2.6962 & -0.677 & -1.3656 & 0.763 & -1.9224 \end{bmatrix}$$

Мера обусловленности для кубич., октаэдр., евклид.норм = 24.849, 27.6, 21.102

$$\text{Число использ.итераций и матрица с собст.значениями матрицы } A(T)^*A: 32 ,$$

$$\begin{bmatrix} 2.027 & -3.23117 \cdot 10^{-27} & 1.38543 \cdot 10^{-13} & 1.28642 \cdot 10^{-23} & 2.06372 \cdot 10^{-19} \\ 1.44031 \cdot 10^{-17} & 0.16 & -7.08383 \cdot 10^{-23} & -5.56155 \cdot 10^{-17} & -2.71821 \cdot 10^{-29} \\ 1.38666 \cdot 10^{-13} & 2.71605 \cdot 10^{-16} & 4.1674 & 1.05013 \cdot 10^{-26} & -2.22493 \cdot 10^{-16} \\ -1.47686 \cdot 10^{-16} & -1.53557 \cdot 10^{-16} & -2.07763 \cdot 10^{-16} & 0.0239 & -1.85992 \cdot 10^{-14} \\ -2.00914 \cdot 10^{-16} & -1.71293 \cdot 10^{-17} & -4.07963 \cdot 10^{-16} & -1.85987 \cdot 10^{-14} & 2.6844 \end{bmatrix}$$

Мера обусловленности для сферич.нормы = 13.194

Рис. 1.12. Выборочные результаты выполнения программы пакетом Maxima.

Исследования, проведенные с помощью программы, показали, что, по крайней мере, для рассматриваемого класса случайных матриц наблюдаются следующее соотношения между указанными мерами обусловленности

$$v(A)_{куб}, v(A)_{окт} \geq v(A)_{евкли} \geq v(A)_{сфера} \quad (1.11)$$

Добавим программу на языке С для среды CodeBlocks, которая позволяет тоже найти все нормы для введенной матрицы и даже рассчитать ее меру обусловленности.

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <math.h>
void maxElem(int n); //прототип функции – находит макс. недиагональный элемент
int lmd(int n,int iter,float eps); //реализует метод вращения для приведенной симм. матрицы
int lmax,kmax; //глобальные переменные
//float *Matr,*Matr1,Amax,fi;
float M[10][10],MM[10][10],Amax,fi;
```

```

int main()
{
    float norm1,norm2,norm3,norm4,norm5,sum,sled,sled1,eps;
    int i,j,k,n,iter,itt,fl;
    SetConsoleCP(1251);           //чтобы воспринимались русские буквы
    SetConsoleOutputCP(1251);
    printf(" *** Программа вычисляет некоторые нормы матрицы *** ");
    printf("\nВведите размерность матрицы: ");
    scanf("%d",&n);
    // Matr=malloc(n*n*sizeof(float));
    // Matr1=malloc(n*n*sizeof(float));
    for (i=0;i<n;i++){
        for (j=0;j<n;j++){
            printf("Введите M(%d,%d): ",i,j);
            scanf("%f",&M[i][j]);
        }
    }
    printf("Произведение Mt*M\n");
    for (i=0;i<n;i++){ //произведение Mt*M
        for (j=0;j<n;j++){
            MM[i][j]=0;
            for (k=0;k<n;k++){
                MM[i][j]=MM[i][j]+M[k][i]*M[k][j];
            }
            printf("%.3f ",MM[i][j]);
        }
        printf("\n");
    }
    sled=0;
    for (i=0;i<n;i++){
        sled+=MM[i][i];
    }
    printf("\nСлед матрицы Mt*M = %.3f",sled);
    maxElem(n);
    printf("\n%.3f,%d,%d,%.3f",Amax,lmax,kmax,f1);
    //вычисляем кубическую и прочие нормы матрицы
    norm1=0;
    for (i=0;i<n;i++){
        sum=0;
        for (j=0;j<n;j++){
            sum+=fabs(M[i][j]);
        }
        if (sum>norm1) norm1=sum;
    }
    printf("\nКубическая норма введенной матрицы = %.3f",norm1);
    norm2=0;
    for (j=0;j<n;j++){
        sum=0;
        for (i=0;i<n;i++){
            sum+=fabs(M[i][j]);
        }
    }

```

```

        if (sum>norm2) norm2=sum;
    }
printf("\nОктаэдрическая норма введенной матрицы = %.3f",norm2);
norm3=0;
for (i=0;i<n;i++){
    for (j=0;j<n;j++){
        norm3+=M[i][j]*M[i][j];
    }
}
norm3=sqrt(norm3);
printf("\nЕвклидова норма введенной матрицы = %.3f",norm3);
fl=0;
while (fl==0){
    printf("\nВведите для метода вращения число итераций и точность: ");
    scanf("%d %f",&iter,&eps);
    printf("%d, %.7f",iter,eps);
    itt=lmd(n,iter,eps);
    printf("\nЧисло использованных итераций = %d\n",itt);
    printf("Приведенная матрица => \n");
    for (i=0;i<n;i++){ //выводим произведение Mt*M
        for (j=0;j<n;j++){
            printf("%.3f ",MM[i][j]);
        }
        printf("\n");
    }
    sled1=0;norm4=0;norm5=10^10;
    for (i=0;i<n;i++){
        sled1+=MM[i][i];
        if (MM[1][i]>norm4){norm4=MM[i][i];};
        if (MM[1][i]<norm4){norm5=MM[i][i];};
    }
    printf("Сумма собственных значений = %.3f",sled1);
    norm4=sqrt(norm4); norm5=sqrt(norm5);
    printf("\nСферическая норма введенной матрицы = %.3f\n",norm4);
    printf("\nМера обусловленности введенной матрицы = %.3f\n",norm4/norm5);
    printf("\nДля завершения программы введите 1 / иначе 0 ");
    scanf("%d",&fl);
    maxElem(n);
    printf("\n%.3f,%d,%d,.3f",Amax,lmax,kmax,fi);
}
//десь s=getchar() не работает совместно с предыдущими scanf()
return 0;
}
void maxElem(int n)
{
int i,j;
float buf,pi=3.1415926535;
Amax=0.;lmax=0;kmax=0;
for (i=0;i<n-1;i++){
    for (j=i+1;j<n;j++){
        buf=fabs(MM[i][j]);
        if (buf>Amax){Amax=buf;lmax=i;kmax=j;};
    }
}
}

```

```

        }
    }
    if
(MM[lmax][lmax]==MM[kmax][kmax]) {fi=(MM[lmax][kmax]/abs(MM[lmax][kmax]))*pi/4; }
    else{fi=0.5*atan(2*MM[lmax][kmax]/(MM[lmax][lmax]-MM[kmax][kmax]));
    }
}
int lmd(int n,int iter,float eps)
{
    int i,j,it,itt;
    float buf,sinfi,cosfi;
    itt=0;
    for (it=0;it<iter;it++){
        maxElem(n);
        if (Amax<eps){return itt;}
        sinfi=sin(fi),cosfi=cos(fi);
        for (i=0;i<n;i++){
            buf=MM[i][lmax]*cosfi+MM[i][kmax]*sinfi;
            MM[i][kmax]=-MM[i][lmax]*sinfi+MM[i][kmax]*cosfi;
            MM[i][lmax]=buf;
        }
        for (j=0;j<n;j++){
            buf=MM[lmax][j]*cosfi+MM[kmax][j]*sinfi;
            MM[kmax][j]=-MM[lmax][j]*sinfi+MM[kmax][j]*cosfi;
            MM[lmax][j]=buf;
        }
        itt++;
    }
    return iter;
}

```

Скриншот с результатами ее работы

```

*** Программа вычисляет некоторые нормы матрицы ***
Введите размерность матрицы: 2
Введите M(0,0): 1
Введите M(0,1): 2
Введите M(1,0): 3
Введите M(1,1): 4
Произведение Mt*M
10.000 14.000
14.000 20.000

След матрицы Mt*M = 30.000
14.000,0,1,-0.614
Кубическая норма введенной матрицы = 7.000
Октаэдрическая норма введенной матрицы = 6.000
Евклидова норма введенной матрицы = 5.477
Введите для метода вращения число итераций и точность: 1000 0.000001
1000, 0.0000010
Число использованных итераций = 1
Приведенная матрица =>
0.134 -0.000
0.000 29.866
Сумма собственных значений = 30.000
Сферическая норма введенной матрицы = 5.465

Мера обусловленности введенной матрицы = 14.933

```

Вопросы и задания для самоконтроля

1. Подсчитайте три нормы для вектора $x = (1, 3, 5, 6)$ и сравните их между собой.
2. Для какой матрицы A справедлива следующая формула для расчета ее меры обусловленности $\nu(A) = \frac{\lambda_1(A)}{\lambda_n(A)}$?
3. Что характеризует мера обусловленности матрицы?
4. Покажите, что кубическая норма вектора удовлетворяет неравенствам (1.1).
5. Найдите двумя способами меру обусловленности следующей матрицы
$$A = \begin{bmatrix} 1 & 2 \\ 2 & 5 \end{bmatrix}$$
6. Можно ли утверждать, что из равенства норм следует равенство векторов?
7. Составьте алгоритм нахождения кубической нормы произвольной матрицы и реализуйте его в Maxima.
8. Что можно сказать о ранге вырожденной матрицы?

Глава 2. Методы решения систем линейных уравнений

Все существующие методы решения систем линейных уравнений $x - ?: Ax = b$ делятся на две группы: итерационные и прямые. В первых методах точное решение можно получить после выполнения бесконечного числа операций, для вторых – конечного.

2.1. Итерационные методы решения систем линейных уравнений

2.1.1. Общая схема итерационного процесса и ее исследование

От системы $Ax = b$ перейдем к равносильной системе

$$C \frac{x - x}{\tau} + Ax = b,$$

где C – произвольная невырожденная матрица ($C > 0$), а $\tau > 0$ – числовой параметр. Для равносильной системы можно предложить следующую итерационную схему

$$C \frac{x^{(k+1)} - x^{(k)}}{\tau} + Ax^{(k)} = b$$

или

$$x^{(k+1)} = Bx^{(k)} + b \quad (2.1)$$

Здесь $k = 0, 1, 2, \dots$ – номер итерации, $x^{(0)}$ – начальное приближение, которое задает пользователь, и

$$B = E - \tau C^{-1} A, \quad b \equiv \tau C^{-1} b \quad (2.2)$$

В дальнейшем, задавая различным образом параметры $C > 0, \tau > 0$, мы из схемы (2.1) получим схемы, соответствующие различным, известным итерационным методам. Сейчас же используем ее для решения вопросов, касающихся всех итерационных схем: условие сходимости итерационного процесса, количественная оценка скорости сходимости, выбор критерия окончания итераций и признак расходимости итерационного процесса.

1. Первую проблему решает следующая теорема.

Теорема. Для сходимости процесса (1) необходимо и достаточно, чтобы все собственные значения матрицы B были меньше единицы: $|\lambda_i(B)| < 1$.

К сожалению, задача нахождения собственных значений матрицы сама по себе не простая. Поэтому дополнительно приведем достаточное условие сходимости, которое требует простого вычисления нормы матрицы.

Следствие. Для сходимости итерационного процесса (2.1) достаточно выполнения неравенства $\|B\| < 1$.

Доказательство. Пусть $Bx = \lambda x$. Тогда $\|B\| \|x\| \geq \|Bx\| = |\lambda| \|x\|$. Отсюда следует неравенство $1 > \|B\| \geq |\lambda|$, которое согласно предыдущей теореме гарантирует сходимость итерационного процесса.

2. Получим оценку скорости сходимости итерационного процесса при условии $\|B\| < 1$.

Если $\|B\| < 1$, то справедливо матричное разложение $(E - B)^{-1} = E + B + B^2 + B^3 + \dots$. Отсюда можно записать точное решение для задачи $x = Bx + b$ в следующем виде

$$x = (E - B)^{-1}b = (E + B + B^2 + B^3 + \dots)b \quad (2.3)$$

С другой стороны, с учетом схемы (2.1) имеем

$$\begin{aligned} x^{(k)} &= Bx^{(k-1)} + b = B(Bx^{(k-2)} + b) + b = B^2x^{(k-2)} + (E + B)b = \dots \\ &= B^k x^{(0)} + (E + B + B^2 + B^3 + \dots B^{k-1})b \end{aligned} \quad (2.4)$$

С учетом соотношений (2.3), (2.4) можно записать

$$x - x^{(k)} = B^k [(E + B + B^2 + B^3 + \dots)b - x^{(0)}]$$

Отсюда легко выводится оценка скорости сходимости итерационного процесса в зависимости от номера итерации

$$\|x - x^{(k)}\| \leq \|B\|^k [(1 + \|B\| + \|B\|^2 + \|B\|^3 + \dots) \|b\| + \|x^{(0)}\|]$$

или, поскольку $\|B\| < 1$,

$$\|x - x^{(k)}\| \leq \|B\|^k \left[\frac{\|b\|}{(1 - \|B\|)} + \|x^{(0)}\| \right] \quad (2.5)$$

Для частного случая $x^{(0)} = b$ оно преобразуется к виду

$$\|x - x^{(k)}\| \leq \|B\|^{k+1} \frac{\|b\|}{(1 - \|B\|)} \quad (2.6)$$

В дополнение к верхнему неравенству выведим еще одну полезную оценку точности решения после k итераций

$$\begin{aligned} x - x^{(k)} &= Bx - Bx^{(k-1)} = Bx - Bx^{(k-1)} + Bx^{(k)} - Bx^{(k)} = B(x - x^{(k)}) + B(x^{(k)} - x^{(k-1)}) \rightarrow \\ \|x - x^{(k)}\| &\leq \|B\| \|x - x^{(k)}\| + \|B\| \|x^{(k)} - x^{(k-1)}\| \rightarrow \\ \|x - x^{(k)}\| &\leq \frac{\|B\|}{(1 - \|B\|)} \|x^{(k)} - x^{(k-1)}\| \end{aligned}$$

3. Критерий окончания итераций.

Метрическое пространство векторов R^n является полным. Поэтому в качестве критерия сходимости последовательности векторов можно использовать критерий Коши: последовательность векторов $\{x^{(k)}\}$ сходится к своему пределу $x \in R^n$ тогда и только тогда, когда для любого $\varepsilon > 0$ существует такое K , что для всех $k > K$ имеет место $\|x^{(k+1)} - x^{(k)}\| < \varepsilon$.

Наряду с абсолютным критерием целесообразно использовать его относительный аналог

$$\|x^{(k+1)} - x^{(k)}\| < \varepsilon \|x^{(k+1)}\| \quad (2.7)$$

Кроме того, число итераций итерационного процесса следует ограничивать сверху неким максимальным числом, чтобы исключить возможное зацикливание программы (когда итерационная схема не способна обеспечить требуемую точность) или наступление аварийной ситуации в случае отсутствия сходимости процесса.

4. Итерационный процесс $x^{(k+1)} = Bx^{(k)} + b$ можно считать расходящимся, если неравенство

$$\|x^{(k-1)} - x^{(k-2)}\| \leq \|x^{(k)} - x^{(k-1)}\|$$

выполняется подряд для трех-пяти последовательных значений k . Справедливость этого утверждения для одномерного случая ($n=1$) демонстрирует рис. 2.1.

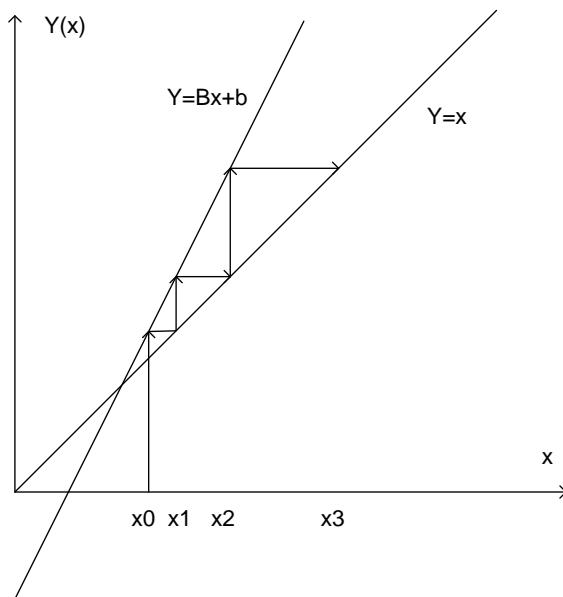


Рис. 2.1. Пример расходящегося итерационного процесса $x^{(k+1)} = Bx^{(k)} + b$ для случая $n = 1$

2.1.2. Частные итерационные схемы

Рассмотрим итерационные схемы, которые вытекают из общей схемы (2.1).

1. **Метод Якоби** получается при следующем выборе параметров: $\tau = 1, C = D$.

Здесь D - диагональная составляющая исходной матрицы $A = A_L + D + A_H$, где A_L, A_H - соответственно, ее нижняя и верхняя составляющие.

Отсюда следует

$$B = E - D^{-1}(A_L + D + A_H) = -D^{-1}(A_L + A_H) \quad (2.8)$$

С учетом очевидного вида обратной диагональной матрицы

$$D^{-1} = \begin{vmatrix} \frac{1}{A_{11}} & 0 & 0 \\ 0 & 0 & \frac{1}{A_{nn}} \end{vmatrix}$$

легко выводится покоординатная запись схемы метода Якоби

$$x_i^{(k+1)} = \frac{1}{A_{ii}} (b_i - \sum_{j \neq i} A_{ij} x_j^{(k)}), \quad i = 1, n \quad (2.9)$$

Достаточное условие сходимости $\|B\| < 1$ для метода имеет следующий вид

$$\|B\|_{ky\delta} = \max_i \sum_j |B_{ij}| = \max_i \sum_{j \neq i} \left| \frac{A_{ij}}{A_{ii}} \right| < 1$$

или

$$\forall i \quad \sum_{j \neq i} |A_{ij}| < |A_{ii}| \quad (2.10)$$

Последнее означает, что для сходимости метода Якоби необходимо преобладание диагональных элементов матрицы над остальными компонентами для каждой строки матрицы.

2. Метод простой итерации имеет место при значениях: $\tau > 0, C = E$.

При этом имеем место:

$$B = E - \tau A, \quad (2.11)$$

покомпонентная запись схемы

$$x_i^{(k+1)} = \tau b_i + \sum_j (\delta_{ij} - \tau A_{ij}) x_j^{(k)}, \quad i = 1, n \quad (2.12)$$

и достаточное условие сходимости метода

$$\|B\|_{ky\delta} = \max_i \sum_j |\delta_{ij} - \tau A_{ij}| < 1 \quad (2.13)$$

Если матрица A симметрична и положительно определена (ее собственные числа $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_n > 0$), то для нее можно указать оптимальный параметр

$$0 < \tau_{onm} = \frac{2}{\lambda_1 + \lambda_n}, \quad (2.14)$$

при котором метод простой итерации всегда сходится, причем с максимальной скоростью, т.к. в этом случае сферическая норма матрицы B принимает минимально возможное значение

$$(\|B\|_{c\phi})_{onm} = \frac{\lambda_1 - \lambda_n}{\lambda_1 + \lambda_n} < 1 \quad (2.15)$$

Покажем справедливость этих утверждений.

Если матрица A - симметричная, то $B = E - \tau A$ также симметрична и поэтому

$$\|B\|_{c\phi} = \max_i |1 - \tau \lambda_i(A)|$$

Поскольку A еще и положительно определенная матрица, то ее собственные числа $\lambda_i(A) > 0$ и поэтому все функции $f_i(\tau) = |1 - \tau\lambda_i(A)|$ будут располагаться внутри зоны, очерченной двумя ломаными линиями $|1 - \tau\lambda_1(A)|$ и $|1 - \tau\lambda_n(A)|$ (рис. 2.2). Сферической норме $\|B(\tau)\|_{c\phi} = \max_i |1 - \tau\lambda_i(A)|$, как функция параметра τ , на этом рисунке будут соответствовать верхние участки ломаных линий. Видно, что норма принимает минимальное значение при τ_{onm} , которое задается координатой точки пересечения ломаных линий

$$1 - \tau_{onm}\lambda_n = \tau_{onm}\lambda_1 - 1$$

При этом $(\|B\|_{c\phi})_{onm} = 1 - \tau_{onm}\lambda_n$.

Отсюда легко выводятся выражения (2.14) и (2.15).

Замечание.

Если матрица плохо убусловлена, то $\lambda_1 \gg \lambda_n$. Тогда согласно выражению (2.15) $(\|B\|_{c\phi})_{onm} \approx 1$ и метод простой итерации для системы будет сходиться очень медленно.

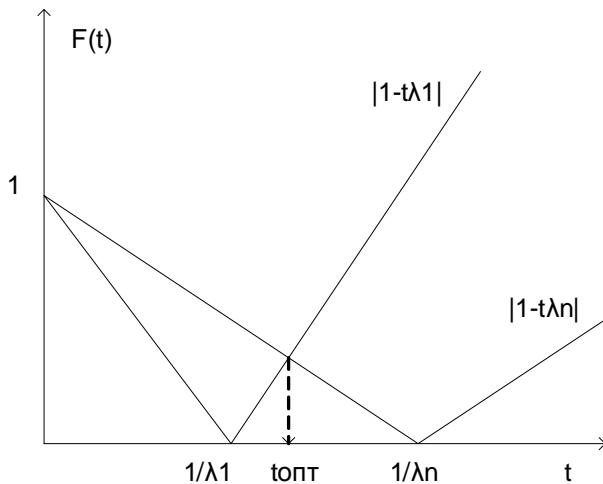


Рис. 2.2. Местоположение функций $f_i(\tau) = |1 - \tau\lambda_i(A)|$

3. В методе верхней релаксации принимается: $\tau > 0, C = D + \tau A_L$

В этом случае из уравнения

$$(D + \tau A_L) \frac{x^{(k+1)} - x^{(k)}}{\tau} + Ax^{(k)} = b$$

следует

$$B = E - \tau(D + \tau A_L)^{-1}A \quad (2.16)$$

$$Dx^{(k+1)} = \tau b - \tau(A_L x^{(k+1)} + A_H x^{(k)}) + (1 - \tau)Dx^{(k)} \quad (2.17)$$

Из матричного уравнения (2.17) легко выводится формула для пересчета координат решения для очередной итерации

$$x_i^{(k+1)} = (1 - \tau)x_i^{(k)} + \frac{\tau}{A_{ii}}[b_i - (\sum_{j=1}^{i-1} A_{ij}x_j^{(k+1)} + \sum_{j=i+1}^n A_{ij}x_j^{(k)})], \quad i = 1, n \quad (2.18)$$

Воспользоваться выражением (2.16) для проверки достаточного условия сходимости метода проблематично, поскольку для этого требуется знание обратной матрицы.

Однако доказано, что если матрица A симметрична и положительно определена, то всегда существуют такие значения $0 < \tau < 2$, при которых метод верхней релаксации сходится.

Частный вариант метода верхней релаксации при $\tau = 1$ известен как метод Зейделя.

В том случае выражения (2.16) и (2.18) преобразуются к виду

$$B = (D + A_L)^{-1} A_H \quad (2.19)$$

$$x_i^{(k+1)} = \frac{1}{A_{ii}} [b_i - (\sum_{j=1}^{i-1} A_{ij} x_j^{(k+1)} + \sum_{j=i+1}^n A_{ij} x_j^{(k)})], \quad i = 1, n \quad (2.20)$$

2.1.3. Алгоритмическая реализация методов

Особенности реализации алгоритмов:

1. В качестве нулевого вектора берется вектор правой части матричного уравнения - $x^{(0)} = b$.
2. Перед запуском итерационного процесса производится деление каждого уравнения на соответствующий диагональный элемент матрицы. При этом, если встречается нулевой диагональный элемент, переменной *code* присваивается единичное значение и осуществляется выход из алгоритма.
3. В алгоритме автоматически отлавливается «не сходимость» итерационного процесса для заданной системы с помощью критерия – увеличение нормы $norm = \|x^{(k+1)} - x^{(k)}\|_{куб}$ в течение пяти последовательных итераций. Если последнее имеет место, то переменной *code* присваивается значение двойки и осуществляется выход из алгоритма.
4. Для окончания процесса используются два критерия: число итераций ограничено сверху заданным максимальным числом K и $norm < \varepsilon$, где K, ε – задаются пользователем. Окончание процесса согласно одному из этих критериев соответствует *code* = 0. При этом в переменных K, ε возвращаются значения использованного числа итераций и достигнутой точности в оценке нормы разности соседних приближений, а в векторе x – полученное решение.
5. Используются: буферный вектор $x0_i, i = 1, n$; переменные *norm_old*, *fl* для хранения, соответственно, нормы, вычисленной на предыдущей итерации, и числа итераций, при которых последовательно возрастает значение этой нормы.
6. *exit sub* - обозначает выход из подпрограммы.

Алгоритм метода Якоби: $x_i^{(k+1)} = \frac{1}{A_{ii}} (b_i - \sum_{j \neq i} A_{ij} x_j^{(k)})$, $i = 1, n$

Входные параметры процедуры: $A, b, n, x, K, \varepsilon$.

```

code = 0,   fl = 0,   norm_old = 0
for i = 1,n  -  нормировка уравнений, задание x0
{
    x0_i = b_i
    if |Aii| < 10^-30 ( $\approx 0$ ) then { code = 1, exit sub}
    bi = bi / Aii
    for j = 1,n { if i ≠ j then Aij = Aij / Aii }
}
for k = 1,K  -  цикл по итерациям
{
    norm = 0
    for i = 1,n  -  цикл по уравнениям
    {
        xi = bi
        for j = 1,n { if i ≠ j then xi = xi - Aij x0_j }
        nt = |xi - x0_i|
        if nt > norm then norm = nt
    }
    if norm ≤ ε then { K = k, ε = norm, exit sub}
    if norm > norm_old then fl = fl + 1 else fl = 0
    if fl > 5 then { code = 2, exit sub}
    norm_old = norm
    for i = 1,n
    {
        x0_i = xi
    }
}

```

Алгоритм простой итерации: $x_i^{(k+1)} = \tau b_i + \sum_j (\delta_{ij} - \tau A_{ij}) x_j^{(k)}, \quad i = 1, n$

Входные параметры процедуры: $A, b, n, x, \tau, K, \varepsilon$.

```

code = 0,   fl = 0,   norm _ old = 0
for i = 1,n   -  нормировка уравнений, задание x0
{
    x0i = bi
    bi = τbi
    for j = 1,n { if i = j then Aij = 1 - τAij else Aij = -τAij }
}
for k = 1,K   -  цикл по итерациям
{
    norm = 0
    for i = 1,n   -  цикл по уравнениям
    {
        xi = bi
        for j = 1,n { xi = xi - Aijx0j }
        nt = |xi - x0i|
        if nt > norm then norm = nt
    }
    if norm ≤ ε then { K = k, ε = norm, exit sub}
    if norm > norm _ old then fl = fl + 1 else fl = 0
    if fl > 5 then { code = 2, exit sub}
    norm _ old = norm
    for i = 1,n { x0i = xi }
}

```

Алгоритм метода верхней релаксации:

$$x_i^{(k+1)} = (1 - \tau)x_i^{(k)} + \frac{\tau}{A_{ii}}[b_i - (\sum_{j=1}^{i-1} A_{ij}x_j^{(k+1)} + \sum_{j=i+1}^n A_{ij}x_j^{(k)})], \quad i = 1, n$$

Входные параметры процедуры: $A, b, n, x, \tau, K, \varepsilon$. Используются буферные переменные: $\tau1, buf, nt$.

```

code = 0,   fl = 0,   norm_old = 0,   τ1 = 1 - τ
for i = 1, n   –  нормировка уравнений, задание x0
{
    xi = bi
    if |Aii| < 10-30 (≈ 0) then { code = 1, exit sub}
    bi = τ bi / Aii
    for j = 1, n { if i ≠ j then Aij = τ Aij / Aii }
}
for k = 1, K   –  цикл по итерациям
{
    norm = 0
    for i = 1, n   –  цикл по уравнениям
    {
        buf = τ1 xi + bi
        for j = 1, n { if i ≠ j then buf = buf - Aij xj }
        nt = |buf - xi|
        xi = buf
        if nt > norm then norm = nt
    }
    if norm ≤ ε then { K = k, ε = norm, exit sub}
    if norm > norm_old then fl = fl + 1 else fl = 0
    if fl > 5 then { code = 2, exit sub}
    norm_old = norm
}

```

Обсудим программу, которая позволяет применить любой итерационный метод для решения заданной пользователем системы.

На рис. 2.3 и 2.4 приведены формы этой программы на начальной и конечной стадиях решения системы линейных уравнений методом верхней релаксации. Чтобы матрицу системы сделать симметричной и положительно определенной систему следует умножить на исходную транспонированную матрицу. Если система не является плохо обусловленной, то такое преобразование обеспечит сходимость итерационного процесса. Для запуска выбранного итерационного метода необходимо задать параметр τ , максимальное число итераций и требуемую точность $\varepsilon > \|x^{(k)} - x^{(k-1)}\|$. По окончанию итерационного процесса в таблице выдается решение $x^{(k)}$ и невязка ($\delta = b - Ax^{(k)}$).

Замечание. В принципе, в качестве критерия окончания итерационного процесса можно было бы использовать условие $\|\delta\| = \|b - Ax^{(k)}\| < \text{eps}$. Этот критерий является более качественным, но на каждой итерации он потребует значительного объема дополнительных операций. К тому же, как видно из таблицы на рис.2.4., применяемый критерий обеспечивает выполнение и этого критерия.

С помощью программы было проведено исследование сходимости методов простой итерации и верхней релаксации для модельной системы (см. таблицу на рис. 2.3). Результаты исследования представлены в таблицах 2.1 и 2.2. Для преобразованной системы метод Якоби также сходится. При этом ее решение определяется с заданной точностью за 1895 итераций. Заметим, что в этом случае для матрицы $B = \begin{bmatrix} 0 & 1,4 \\ 0,7 & 0 \end{bmatrix}$ (матрица получена с учетом (2.8)) имеем $\lambda_{1,2} = \pm 0,99 < 1 < \|B\|_{\text{куб}} = 1,4$. Т.е. «необходимое и достаточное условие» выполнено, а «только достаточное условие» – нет.

Сравнивая все полученные результаты, можно сделать вывод, что для выбранной модельной системы среди всех итерационных методов метод верхней релаксации имеет наиболее высокую скорость сходимости. Естественно этот вывод не распространяется на все системы.

i \ j	1	2	b	x	невязка
1	1	2	1		
2	3	4	2		

Строка, столбец и значение компоненты:

Значение параметра - t	<input type="text" value="1.5"/>	Выберите метод
Требуемая точность - eps	<input type="text" value=".0000001"/>	<input type="radio"/> - метод Якоби
Максимальное число итераций	<input type="text" value="1000"/>	<input type="radio"/> - метод простой итерации
		<input checked="" type="radio"/> - метод верхней релаксации

Рис. 2.3. Задание исходной системы уравнений

Решение системы линейных уравнений $Ax=b$ итерационными методами

справка выход

Размерность системы (>1) 2

Матрица, правая часть системы $Ax=b$, ее решение и невязка для полученного решения

i \ j	1	2	b	x	невязка
1	10,	14,	7,	-0,0000015	0,0000006
2	14,	20,	10,	0,500001	0,0000005

Строка, столбец и значение компоненты: 2 3 2

Значение параметра - t 1,5

Требуемая точность - eps 0,000001

Максимальное число итераций 1000

Выберите метод

- метод Якоби
 - метод простой итерации
 - метод верхней релаксации

Рис. 2.4. Результаты после окончания итерационного процесса. Система была предварительно умножена на исходную транспонированную матрицу

Таблица 2.1. Зависимость предельного числа итераций от параметра τ для метода верхней релаксации (принято $\varepsilon = 0,0000001$) .

τ	1	1,25	1,5	1,75	1,8	1,85	1,86	1,87
итерации	621	397	233	77	75	99	109	расходится

Таблица 2.2. Зависимость предельного числа итераций от параметра τ (при $\varepsilon = 0,0000001$) для метода простой итерации. В данном случае для модельной системы имеем: $\lambda_1 = 29,866$, $\lambda_2 = 0,1339$, $\tau_{onm} = 0,066$.

τ	0,01	0,02	0,04	0,05	0,06	0,066	0,0665	0,067
итерации	5905	3310	1733	1419	1204	1105	1359	расходится

Характер зависимости, зафиксированной в таблице 2.2, хорошо объясняет рис. 2.5.

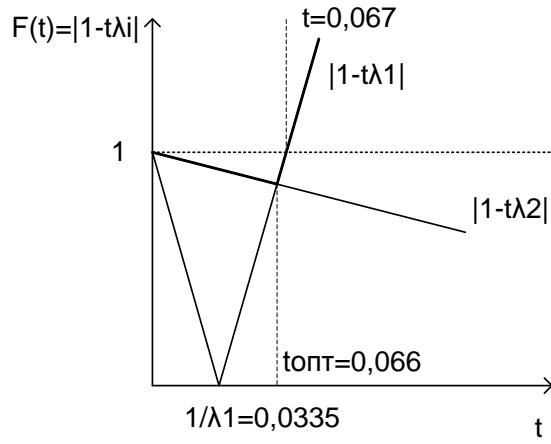


Рис. 2.5. Местоположение функций $f_i(\tau) = |1 - \tau\lambda_i(A)|$ и τ_{onm} для метода простой итерации.

В качестве второго примера рассмотрим систему $\begin{bmatrix} 1 & 0,2 \\ 0,3 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$.

Результаты, полученные с помощью программы для данной системы, приведены в таблице 2.3.

Таблица 2.3. Значения оптимального значения параметра τ и число итераций, использованное для получения решения системы с точностью $\|x^k - x^{k-1}\| < \varepsilon = 0,0000001$.

Метод	τ_{onm}	Число итераций (для $\varepsilon = 0,0000001$)
Якоби	-	12
Простой итерации	0,95	12
Верхней релаксации	1.05	6

Для сравнения приведем листинг и результаты работы программы, реализующей метод верхней релаксации в рамках пакета xwMaxima.

```

kill(all);
/* п/п метода верхней релаксации */
metod.UR(A,n,t,eps,k_Max):=block
(
  [i,j,k,norm_old,nt,fl,t1,norm,buf],
  code:0,norm_old:0,fl:0,t1:1-t,k_Out:0,
  /* нормировка системы */
  for i thru n do
  (
    A[i,n+2]:A[i,n+1],
    if abs(A[i,i])<0.00000001 then return(code:1),
    A[i,n+1]:=t*A[i,n+1]/A[i,i],
    for j thru n do if i#j then A[i,j]:=t*A[i,j]/A[i,i]
)

```

```

),
/* цикл по итерациям */
for k thru k_Max do
(
    norm:0,k_Out:k_Out+1,
    /* цикл по уравнениям */
    for i thru n do
    (
        buf:t1*A[i,n+2]+A[i,n+1],
        /* обработка текущего уравнения */
        for j thru n do if i#j then buf:buf-A[i,j]*A[j,n+2],
        nt:abs(buf-A[i,n+2]), A[i,n+2]:buf,
        if nt>norm then norm:nt
    ),
    if norm<=eps then (eps_Out:norm, return(code:0)),
    if norm>norm_old then fl:fl+1 else fl:0,
    if fl>3 then (eps_Out:norm,return(code:2)),
    norm_old:norm
)
);
/* главная программа */
numer:true;
fpprintprec:5;
n:5; eps:0.00001; k_Max:5000; t:1;
A:zeromatrix(n,n); b:zeromatrix(n,1); x:zeromatrix(n,1);
/* формируем систему случайными числами  $\in [-0.5: 0.5]$  */
for i thru n do for j thru n do A[i,j]:random(1.0)-0.5; A:A.transpose(A);
for i thru n do b[i]:random(1.0)-0.5;
for i thru n do for j thru n do b[i]:b[i]+A[i,j];
A:addcol(A,b,x); /* формируем расширенную матрицу */
A_old:copy(A);
metod_UR(A,n,t,eps,k_Max);
print("code, eps, k : ",code," ,",eps_Out," ,",k_Out);
eps; k_Max;
for i thru n do x[i]:A[i,n+2]; x; /* решение */
/* строим зависимость */
t:0.05; dt:0.05; Lt:[]; Lk:[];
for i thru 45 do
(
    A:copy(A_old), metod_UR(A,n,t,eps,k_Max),
    if code=0 then (Lt:append(Lt,[t]), Lk:append(Lk,[k_Out])),
    t:t+dt
);
Lt; Lk;
plot2d([discrete,Lt,Lk],[gnuplot_preamble,"set grid;"],[xlabel,"параметр - t"],
[ylabel,"число использованных итераций - iter"]);

```

Сформированная система

$$(A) \quad \begin{bmatrix} 0.52957 & 0.3722 & -0.045244 & 0.34236 & 0.47583 & 1.6747 & 0 \\ 0.3722 & 0.60408 & 0.2534 & 0.22513 & 0.36081 & 1.8156 & 0 \\ -0.045244 & 0.2534 & 0.31359 & -0.040235 & 0.013459 & 0.49497 & 0 \\ 0.34236 & 0.22513 & -0.040235 & 0.2984 & 0.24582 & 1.0715 & 0 \\ 0.47583 & 0.36081 & 0.013459 & 0.24582 & 0.5253 & 1.6212 & 0 \end{bmatrix}$$

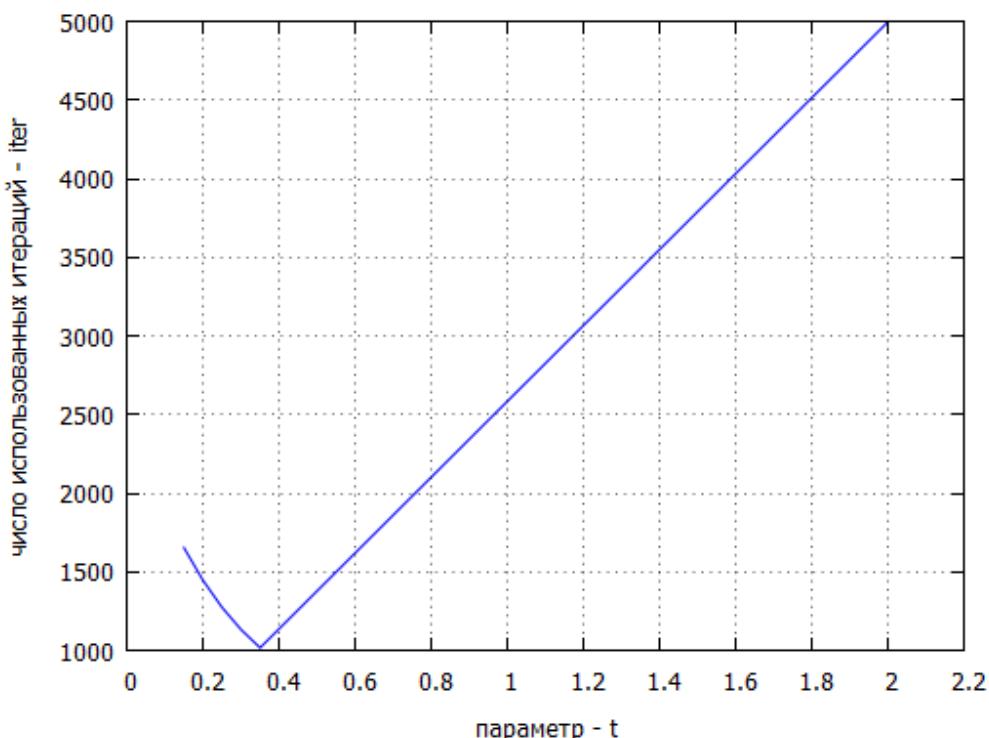
Полученное решение

$$(\%o24) \quad \begin{bmatrix} -0.002727 \\ 1.2616 \\ 0.68498 \\ 1.4841 \\ 1.5102 \end{bmatrix}$$

Два списка, по которым для заданной точности $\varepsilon = 0.00001$ построена зависимость числа итераций от параметра τ

(%o30) [0.15, 0.2, 0.25, 0.3, 0.35, 2.0]

(%o31) [1653, 1445, 1273, 1132, 1016, 5000]



Рассмотрим еще один вариант программы, реализующей метод верхней релаксации на С в среде программирования CodeDlocks.

Сценарий программы:

1. Ввод размерности системы, запрос памяти под систему и ее преобразованный вариант - ($A^T Ax = A^T b$)
2. Ввод системы вручную или сформировать, используя датчик псевдослучайных чисел из диапазона [-0.5;0.5] – используется расширенная матрица $A_{in+1} = b_i$
3. Преобразование системы $A^T Ax = Ab$

4. Ввод K , ε , τ
5. Вызов процедуры, реализующей метод - вернуть решение в последнем столбце расширенной матрицы $A_{in+2} = x_i$
6. Выдать полученное решение, достигнутое ε и число использованных итераций.
7. Выдать невязку для полученного решения $\delta = b - Ax$
8. Если есть запрос на поиск оптимального значения параметра τ , то осуществить его и выдать полученные при этом величины: число затраченных итераций и оптимальное τ .
9. Предоставить возможность повторить вычисления с новыми итерационными параметрами.

Листинг программы

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <math.h>
#include <time.h>
#define B(i,j) MM[i*(n+2)+j]
#define A(i,j) M[i*(n+2)+j]

float *M,*MM,eps,epst,t;
int n,iter,it;
int iter_Up();
void var_M();
void Ax_b();

int main()
{
    int i,j,code,fl=0,c;
    int fl1=0,iopt;
    float topt,dt=0.05;
    char s[2];
    SetConsoleCP(1251);           //чтобы воспринимала русские буквы
    SetConsoleOutputCP(1251);
    printf(" *** Программа находит решение системы методом верхней релаксации *** ");
    printf("\nВведите размерность системы: ");
    scanf("%d",&n);
    M=malloc(n*(n+2)*sizeof(float));
    MM=malloc(n*(n+2)*sizeof(float));
    printf("Введите 0/1 - задать систему случайным образом или вручную ");
    scanf("%d",&c);
    switch (c){
        case 0:
            //инициализация датчика п.с. чисел текущим временем
            srand(time(NULL));
            for(i=0;i<n;i++)for(j=0;j<n+1;j++)B(i,j)=0.5-rand()/(RAND_MAX+1.0);
            break;
        default:
            for (i=0;i<n;i++){
                for (j=0;j<n;j++){
                    if (i==j) B(i,j)=1.0;
                    else B(i,j)=0.0;
                }
            }
    }
}
```

```

for (j=0;j<n;j++){
    printf("Введите B(%d,%d): ",i,j);
    scanf("%f",&B(i,j));
}
printf("Введите b(%d): ",i);
scanf("%f",&B(i,n));
}
}
printf("Введенная система:\n");
for (i=0;i<n;i++){
    for (j=0;j<n;j++){
        printf("%f ",B(i,j));
    }
    printf("%f\n",B(i,n));
}
var_M(); //умножаем систему на At
printf("Преобразованная система:\n");
for (i=0;i<n;i++){
    for (j=0;j<n;j++){
        printf("%f ",A(i,j));
    }
    printf("%f\n",A(i,n));
}
while (fl==0){
    printf("Введите число итераций,точность и параметр t: ");
    scanf("%d %f %f",&iter,&eps,&t);
    printf("Введенные точность, число итераций и t = %f %d %.2f",eps,iter,t);
    var_M();
    code=iter_Up();
    switch (code)
    {
        case 0: //все хорошо
            printf("\nКод = %d, Число использованных итераций = %d, Точность = %.8f",code,it,epst);
            printf("\nРешение:\n");
            for(i=0;i<n;i++){
                printf("%f ",A(i,n+1));
            }
            Ax_b();
            printf("\nНевязка:\n");
            for(i=0;i<n;i++){
                printf("%f ",B(i,n+1));
            }
            printf("\nНайти оптимальное t -> Введите 0, иначе 1: "); scanf("%d",&fl1);
            if(fl1==0){ //нужно найти оптимальное t
                iopt=iter;t=0;
                for(i=0;i<40;i++){
                    t=t+dt; // dt = 0.05
                    var_M();
                    code=iter_Up();
                    if((code==0)&&(it<iopt)){iopt=it;topt=t;}
                }
            }
    }
}

```

```

printf("\nПолучены следующие оптимальные значения t и iter: %.3f; %d",topt,iOpt);
printf("\nПовторить вычисления для новых параметров итерационного процесса -
введите 0, иначе 1: ");
scanf("%d",&fl);
}
break;
case 1:
printf("\nВ матрице системы есть диагональный компонент = 0 !");
fl=1;
break;
case 2:
printf("\nМетод разошелся !");
fl=1;
break;
}
}

printf("\nДля завершения программы нажмите любую клавишу и ENTER ");
scanf("%s",s); printf("\n");
return 0;
}

//п/п реализует итерационный метод верхней релаксации
int iter_Up(){
    int i,j,k,fl=0;
    float nt,buf,norm,norm_old=0,t1=1-t;
    //нормировка системы
    for(i=0;i<n;i++){
        A(i,n+1)=A(i,n);
        if (fabs(A(i,i)<1.E-30))return 1; //есть A(i,i)=0
        A(i,n)=t*A(i,n)/A(i,i);
        for(j=0;j<n;j++)if(i!=j)A(i,j)=t*A(i,j)/A(i,i);
    }
    it=0; //число выполненных итераций
    for(k=1;k<=iter;k++){ //цикл по итерациям
        it=it+1; norm=0;
        for(i=0;i<n;i++){ //цикл по уравнениям
            buf=t1*A(i,n+1)+A(i,n);
            for(j=0;j<n;j++) if(i!=j)buf=buf-A(i,j)*A(j,n+1);
            nt=fabs(buf-A(i,n+1)); A(i,n+1)=buf;
            if(nt>norm) norm=nt;
        }
        epst=norm;
        if(norm<=eps) return 0; //все хорошо
        if(norm>norm_old)fl=fl+1;else fl=0;
        if(fl>3) return 2; //процесс расходится
        norm_old=norm;
    }
    return 0;
}

//п/п симметризует систему
void var_M(){
    int i,j,k;
}

```

```

for(i=0;i<n;i++){
    for(j=0;j<=n;j++){
        A(i,j)=0.;
        for (k=0;k<n;k++)A(i,j)=A(i,j)+B(k,i)*B(k,j);
    }
}
//п/п вычисляет невязку
void Ax_b(){
int i,j;
for(i=0;i<n;i++){
    B(i,n+1)=B(i,n);
    for(j=0;j<n;j++) B(i,n+1)=B(i,n+1)-B(i,j)*A(j,n+1);
}
}

```

Скриншоты с результатами работы программы

```

*** Программа находит решение системы методом верхней релаксации ***
Введите размерность системы: 3
Введите 0/1 - задать систему случайным образом или вручную 0
Введенная система:
0.400665 0.418091 0.338928 -0.455750
-0.136871 0.086853 0.419891 0.271912
0.071259 0.380920 -0.493042 -0.070831
Преобразованная система:
0.184344 0.182771 0.043192 -0.224867
0.182771 0.327444 -0.009638 -0.193909
0.043192 -0.009638 0.534271 -0.005370
A(n-1,n) = -0.009638
Введите число итераций,точность и параметр t: 1000 0.0000001 1
Точность, число итераций и t = 0.000000 1000 1.00
Код = 0, Число использованных итераций = 29, Точность = 0.00000000
Решение:
-1.484109 0.239563 0.114250
Невязка:
-0.000000 0.000000 0.000000
Найти оптимальное t -> Введите 0, иначе 1: 0

Получены следующие оптимальные значения t и iter: 1.250; 14
Повторить вычисления для новых параметров итерационного процесса - введите 0, иначе 1: 0
Введите число итераций,точность и параметр t: 50 0.0000001 1.25

```

Итерационные методы целесообразно применять для систем линейных уравнений с сильно разреженными нерегулярными матрицами. Подобные системы обычно возникают при решении разностными методами задач математической физики.

Рассмотрим пример использования метода Зейделя при решении задачи стационарной теплопроводности.

Словесная (верbalная) формулировка задачи: при заданной плотности источников $q(x, y)$ найти распределение температуры $T(x, y)$ в плоской прямоугольной области, две границы которой теплоизолированы, а через другие происходит теплообмен.

Математически эта задача записывается следующим образом

$$k \left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) = -q(x, y), \quad (x, y) \in [0, l_x; 0, l_y]$$

$$\frac{\partial T}{\partial x} = 0, \quad x = 0; \quad \frac{\partial T}{\partial y} = 0, \quad y = 0$$

$$k \frac{\partial T}{\partial x} = -\alpha_x (T - T_0), \quad x = l_x; \quad k \frac{\partial T}{\partial y} = -\alpha_y (T - T_0), \quad y = l_y$$

Здесь $k, T_0, \alpha_x, \alpha_y$ - коэффициент теплопроводности, температура окружающей среды и коэффициенты поверхностной теплопроводности.

Для уменьшения зависимости задачи от исходных параметров перейдем к новым величинам:

$$y \equiv \frac{y}{l_y}, \quad x \equiv \frac{x}{l_y} \Rightarrow y \in [0, 1], \quad x \in [0, l_x \equiv \frac{l_x}{l_y}]$$

$$T \equiv T - T_0, \quad q \equiv \frac{q}{k} l_y^2, \quad \alpha_x \equiv \frac{\alpha_x}{k} l_y, \quad \alpha_y \equiv \frac{\alpha_y}{k} l_y$$

С учетом этих изменений исходная задача примет вид

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = -q(x, y), \quad (x, y) \in [0, l_x; 0, 1]$$

$$\frac{\partial T}{\partial x} = 0, x = 0; \quad \frac{\partial T}{\partial y} = 0, y = 0$$

$$\frac{\partial T}{\partial x} = -\alpha_x T, x = l_x; \quad \frac{\partial T}{\partial y} = -\alpha_y T, y = 1$$

Приведенную задачу будем решать разностным методом. Для этого область покроем прямоугольной сеткой (рис. 2.6), а уравнение теплопроводности и граничные условия запишем через значения искомой функции $T(x, y)$ в узлах сетки (T_{ij})

$$\frac{T_{i-1,j} - 2T_{i,j} + T_{i+1,j}}{h_x^2} + \frac{T_{ij-1} - 2T_{ij} + T_{ij+1}}{h_y^2} = -q_{i,j}, \quad (i, j) \in [0, N_x; 0, N_y]$$

$$\frac{T_{1,j} - T_{-1,j}}{2h_x} = 0, \quad j \in [0, N_y]; \quad \frac{T_{i1} - T_{i,-1}}{2h_y} = 0, \quad i \in [0, N_x]$$

$$\frac{T_{N_x+1,j} - T_{N_x-1,j}}{2h_x} = -\alpha_x T_{N_x,j}, \quad j \in [0, N_y]; \quad \frac{T_{iN_y+1} - T_{iN_y-1}}{2h_y} = -\alpha_y T_{iN_y}, \quad i \in [0, N_x]$$

Разностная задача аппроксимирует исходную с точностью порядка $O(h_x^2, h_y^2)$. Последнее легко проверить, если в разностные аппроксимации всех производных подставить соответствующее разложение Тейлора, подобное данному

$$T_{i\pm 1,j} = T(x_i, y_j) \pm h_x \frac{\partial T}{\partial x}(x_i, y_j) + \frac{1}{2} h_x^2 \frac{\partial^2 T}{\partial x^2}(x_i, y_j) \pm \frac{1}{6} h_x^3 \frac{\partial^3 T}{\partial x^3}(x_i, y_j) + \frac{1}{24} h_x^4 \frac{\partial^4 T}{\partial x^4}(\vartheta_x, y_j),$$

где $\vartheta_x \in [x_i, x_i \pm h_x]$.

Исключив с помощью граничных условий из разностных уравнений для граничных узлов значения T_{ij} , связанные с внешними узлами, и разрешив после этого все линейные уравнения относительно диагональных элементов разреженной матрицы, получим систему уравнений в форме, соответствующей методу Зейделя.

Процедура реализация метода Зейделя для данной системы выглядит следующим образом:

- все узлы сетки обрабатываются слева-направо слой за слоем, начиная с нижнего слоя ($i = 0$);
- выход из итерационного процесса происходит по критерию $\max_{i,j} |T_{ij}^k - T_{ij}^{k-1}| < \text{eps}$, где k - номер итерации, или по исчерпание заданного числа итераций.

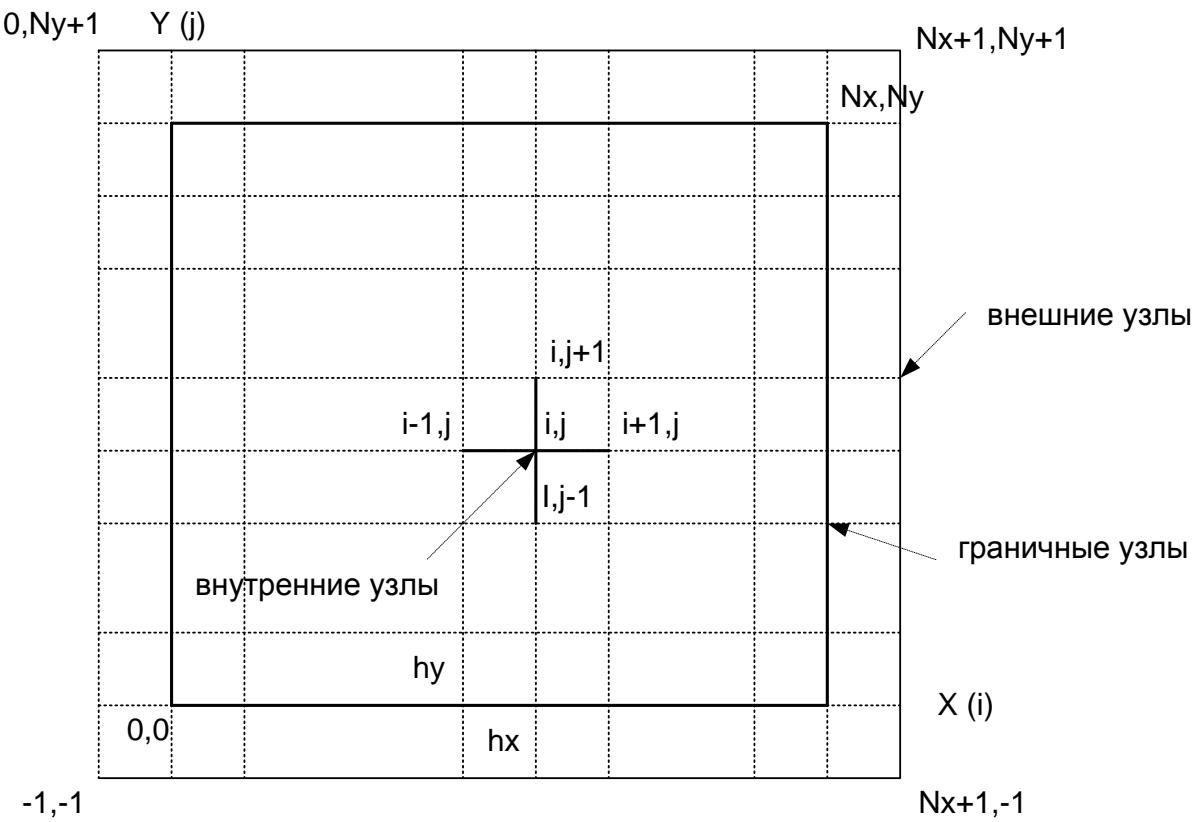


Рис. 2.6. Разностная сетка, используемая при решении задачи теплопроводности для плоской прямоугольной области.

Реализация метода Зейделя для решения приведенной задачи теплопроводности выполнена в рамках пакета Maxima. Листинг соответствующей программы и некоторые результаты ее работы приведены на рис 2.7 – 2.10. При тестировании использовались следующие исходные данные: $l_x = 2$, $h_x = h_y = 0.1$, $N_x = 20$, $N_y = 10$, $\alpha_x = \alpha_y = 1$, $q(x, y) = (2 - x)(1 - y)$.

```

kill(all);
/* подпрограмма рассчитывает коэффициенты разностной разреженной
системы */
/* линейных уравнений */
coef(hx,hy,alfx,alfy):=block
(
  M:zeromatrix(9,3),
  M[1,3]:hx^2*hy^2/(2*(hx^2+hy^2)), M[1,1]:M[1,3]*2/hx^2,
  M[1,2]:M[1,3]*2/hy^2,
  M[2,3]:M[1,3], M[2,1]:M[2,3]/hx^2, M[2,2]:M[1,2],
  M[3,3]:hx^2*hy^2/(2*(hx^2+hy^2+alfx*hx*hy^2)), M[3,1]:M[3,3]*2/hx^2,
  M[3,2]:M[3,3]*2/hy^2,
  M[4,3]:M[1,3], M[4,1]:M[1,1], M[4,2]:M[4,3]/hy^2,
  M[5,3]:M[1,3], M[5,1]:M[5,3]/hx^2, M[5,2]:M[4,2],
  M[6,3]:M[3,3], M[6,1]:M[3,1], M[6,2]:M[6,3]/hy^2,
)

```

```

M[7,3]:hx^2*hy^2/(2*(hx^2+hy^2+alfy*hy*hx^2)), M[7,1]:M[7,3]*2/hx^2,
M[7,2]:M[7,3]*2/hy^2,
M[8,3]:M[7,3], M[8,1]:M[8,3]/hx^2, M[8,2]:M[7,2],
M[9,3]:hx^2*hy^2/(2*(hx^2+hy^2+alfx*hx*hy^2+alfy*hy*hx^2)),
M[9,1]:M[9,3]*2/hx^2, M[9,2]:M[9,3]*2/hy^2
);

define(f(x,y),(2-x)*(1-y)); /* заданная функция тепловыделения */

/* подпрограмма рассчитывает значения плотности тепловыделения в узлах
сетки */
create_q(Nx,Ny,hx,hy):=block
(
  [i,j,x,y],
  q:zeromatrix(Nx+1,Ny+1),
  for i thru Nx+1 do
  (
    x:hx*(i-1), for j thru Ny+1 do (y:hy*(j-1), q[i,j]:=f(x,y))
  )
);

/* подпрограмма реализует итерационный метод Зейделя для разностной
системы */
/* линейных уравнений */
zadel(Nx,Ny,iter,eps):=block
(
  [i,j,k,buf,nt],
  it:0, T:zeromatrix(Nx+1,Ny+1),
  for k thru iter do
  (
    it:it+1, norm:0,
    /* обработка нижнего слоя сетки */
    buf:M[1,1]*T[2,1]+M[1,2]*T[1,2]+M[1,3]*q[1,1],
    nt:abs(T[1,1]-buf), T[1,1]:=buf, if nt>norm then norm:=nt,
    for i:2 thru Nx do
    (
      buf:M[2,1]*(T[i-1,1]+T[i+1,1])+M[2,2]*T[i,2]+M[2,3]*q[i,1],
      nt:abs(T[i,1]-buf), T[i,1]:=buf, if nt>norm then norm:=nt
    ),
    buf:M[3,1]*T[Nx,1]+M[3,2]*T[Nx+1,2]+M[3,3]*q[Nx+1,1],
    nt:abs(T[Nx+1,1]-buf), T[Nx+1,1]:=buf, if nt>norm then norm:=nt,
    /* обработка промежуточных слоев сетки */
    for j:2 thru Ny do
    (

```

```

buf:M[4,1]*T[2,j]+M[4,2]*(T[1,j-1]+T[1,j+1])+M[4,3]*q[1,j],
nt:abs(T[1,j]-buf), T[1,j]:buf, if nt>norm then norm:nt,
for i:2 thru Nx do
(
  buf:M[5,1]*(T[i-1,j]+T[i+1,j])+M[5,2]*(T[i,j-1]+T[i,j+1])+M[5,3]*q[i,j],
  nt:abs(T[i,j]-buf), T[i,j]:buf, if nt>norm then norm:nt
),
buf:M[6,1]*T[Nx,j]+M[6,2]*(T[Nx+1,j-1]+T[Nx+1,j+1])+M[6,3]*q[Nx+1,j],
nt:abs(T[Nx+1,j]-buf), T[Nx+1,j]:buf, if nt>norm then norm:nt
),
/* обработка последнего слоя сетки */
buf:M[7,1]*T[2,Ny+1]+M[7,2]*T[1,Ny]+M[7,3]*q[1,Ny+1],
nt:abs(T[1,Ny+1]-buf), T[1,Ny+1]:buf, if nt>norm then norm:nt,
for i:2 thru Nx do
(
  buf:M[8,1]*(T[i-1,Ny+1]+T[i+1,Ny+1])+M[8,2]*T[i,Ny]+M[8,3]*q[i,Ny+1],
  nt:abs(T[i,Ny+1]-buf), T[i,Ny+1]:buf, if nt>norm then norm:nt
),
buf:M[9,1]*T[Nx,Ny+1]+M[9,2]*T[Nx+1,Ny]+M[9,3]*q[Nx+1,Ny+1],
nt:abs(T[Nx+1,Ny+1]-buf), T[Nx+1,Ny+1]:buf, if nt>norm then norm:nt,
if norm<=eps then return(0) /* 0 - заданная точность достигнута */
)
);

```

```

/* главная программа */
numer:true;
fpprintprec:5;
hx:0.1; hy:0.1; alfx:1; alfy:1;
coef(hx,hy,alfx,alfy); M;
Nx:20; Ny:10;
create_q(Nx,Ny,hx,hy); q;
h(x,y):='(q[round(x)+1,round(y)+1]);
plot3d(h(x,y),[x,0,20],[y,0,10],[grid,20,10], [legend, "График распределение
плотности источников"]);
eps:0.00001; iter:1500; zadel(Nx,Ny,iter,eps); it;norm;
tem(x,y):='(T[round(x)+1,round(y)+1]);
plot3d(tem(x,y),[x,0,20],[y,0,10],[grid,20,10],[legend, "График распределение
температуры"]);
/* поток через верхнюю границу y=1, численное интегрирование методом
трапеций */
Ly:submatrix(T,1,2,3,4,5,6,7,8,9,10);
s1:((Ly[1,1]+Ly[21,1])/2+sum(Ly[i,1],i,2,20))*hx*alfy;
/* поток через правую границу x=2 */

```

```

Lx:submatrix(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,T);
s2:((Lx[1,1]+Lx[1,11])/2+sum(Lx[1,j],j,2,10))*hy*alfx;
print("Суммарный поток энергии через границу области =",s:s1+s2);
print("Суммарная плотность тепловыделения внутри области =",integrate(2-x,
x,0,2)*integrate(1-y,y,0,1));

```

Рис. 2.7. Листинг программы, рассчитывающей температуру над прямоугольной областью.

```

0.5   0.5   0.0025
0.25  0.5   0.0025
0.476 0.476  0.00238
0.5   0.25  0.0025
(%o12) 0.25  0.25  0.0025
0.476 0.238  0.00238
0.476 0.476  0.00238
0.238 0.476  0.00238
0.455 0.455  0.00227
(%o31) 1500
(%o32) 0.00735

```

	0.884	0.877	0.859	0.831	0.796	0.754	0.707	0.656	0.604	0.551	0.5
	0.88	0.874	0.856	0.828	0.793	0.751	0.704	0.654	0.602	0.55	0.499
	0.871	0.865	0.847	0.82	0.785	0.744	0.698	0.648	0.597	0.545	0.494
	0.857	0.851	0.833	0.807	0.773	0.733	0.687	0.639	0.589	0.538	0.488
	0.838	0.832	0.815	0.79	0.757	0.718	0.674	0.626	0.577	0.527	0.479
	0.815	0.81	0.794	0.769	0.737	0.699	0.657	0.611	0.563	0.515	0.467
	0.789	0.784	0.768	0.745	0.714	0.678	0.637	0.593	0.547	0.5	0.454
	0.76	0.755	0.741	0.718	0.689	0.654	0.615	0.573	0.529	0.484	0.439
	0.729	0.724	0.71	0.689	0.661	0.628	0.591	0.551	0.509	0.465	0.423
	0.695	0.691	0.678	0.658	0.632	0.6	0.565	0.527	0.487	0.446	0.405
(%o33)	0.66	0.656	0.644	0.625	0.6	0.571	0.538	0.502	0.464	0.425	0.386
	0.624	0.62	0.608	0.591	0.568	0.541	0.51	0.476	0.44	0.403	0.366
	0.586	0.582	0.572	0.556	0.535	0.509	0.48	0.449	0.416	0.381	0.346
	0.548	0.545	0.535	0.52	0.501	0.477	0.451	0.422	0.39	0.358	0.326
	0.51	0.507	0.498	0.485	0.467	0.445	0.421	0.394	0.365	0.335	0.305
	0.472	0.469	0.461	0.449	0.433	0.413	0.391	0.366	0.34	0.312	0.284
	0.434	0.432	0.425	0.414	0.399	0.382	0.361	0.339	0.315	0.289	0.263
	0.398	0.396	0.389	0.38	0.366	0.351	0.332	0.312	0.29	0.267	0.242
	0.362	0.36	0.355	0.346	0.335	0.32	0.304	0.285	0.266	0.244	0.222
	0.329	0.327	0.322	0.315	0.304	0.291	0.277	0.26	0.242	0.223	0.203
	0.298	0.296	0.292	0.285	0.276	0.264	0.251	0.236	0.22	0.202	0.184

Суммарный поток энергии через границу области = 1.0

Суммарная плотность тепловыделения внутри области = 1.0

Рис.2.8. Некоторые результаты выполнения программы пакетом Maxima: матрицы коэффициентов системы и рассчитанных значений температуры в узлах сетки, число использованных итераций и достигнутая точность. Было выполнено 1500 итераций методом Зейделя. При этом достигнута точность $\max_{i,j} |T_{ij}^k - T_{ij}^{k-1}| < 0,00735$, где k - номер итерации метода Зейделя.

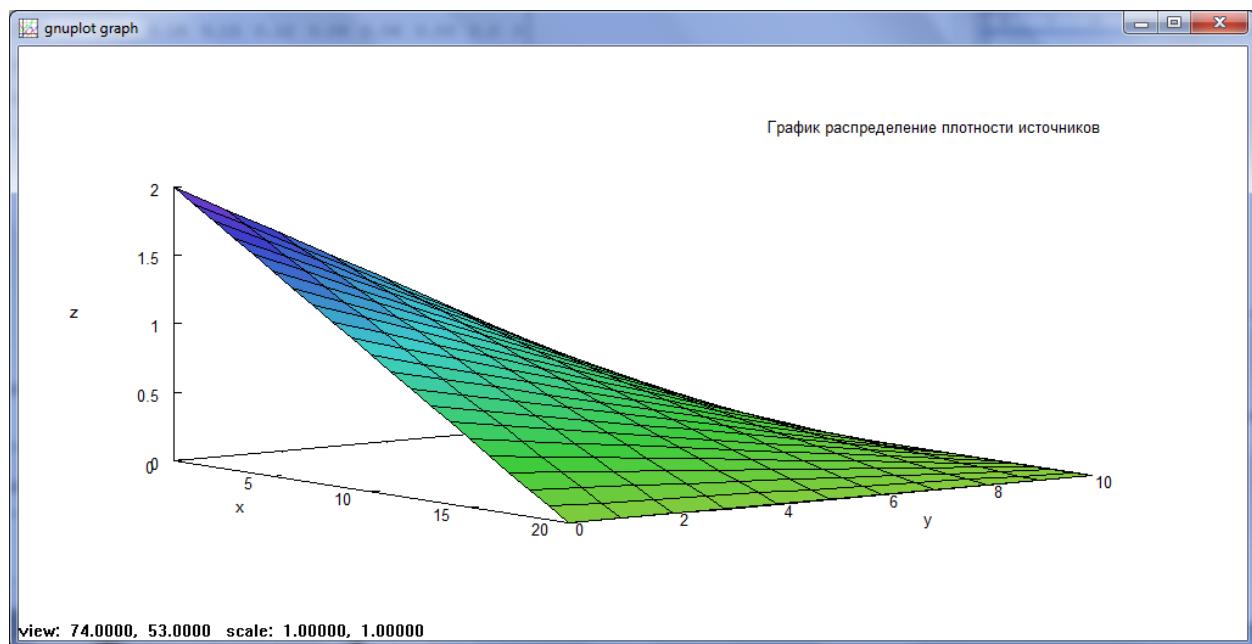


Рис. 2.9. График распределения заданной плотности тепловыделения
 $q(x, y) = (2 - x)(1 - y).$

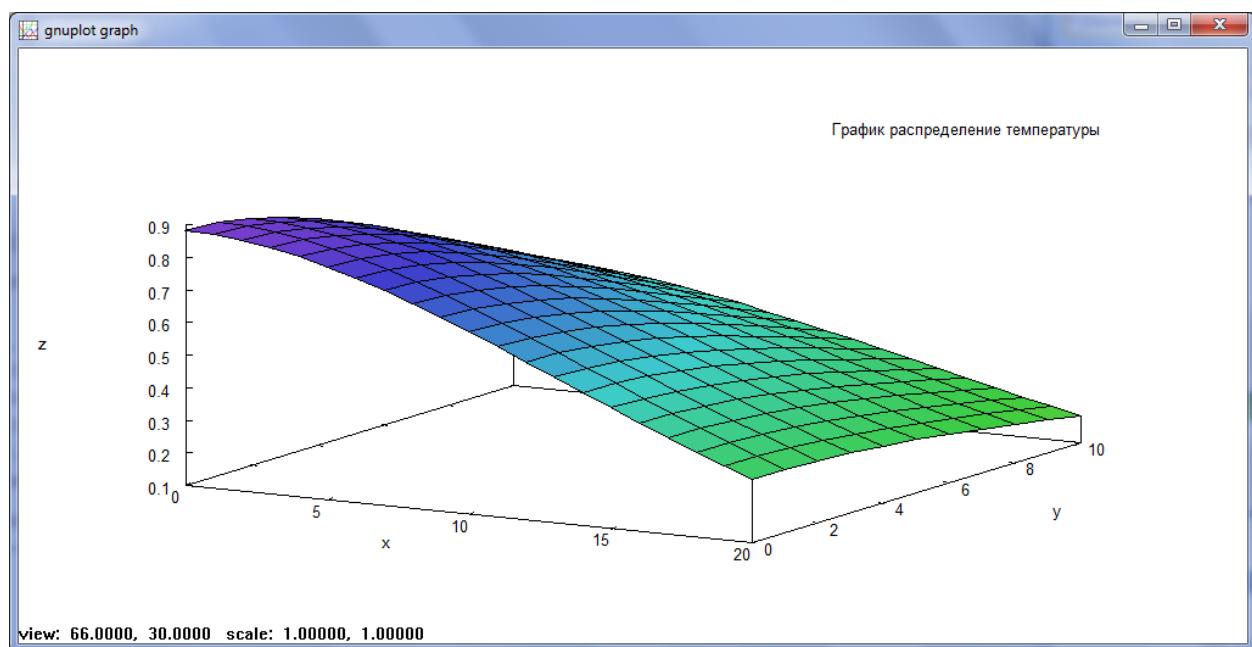


Рис. 2.10. График распределения температуры, рассчитанной методом Зейделя для разностной задачи теплопроводности.

Для оценки достоверности полученного решения дополнительно использовался закон сохранения энергии, согласно которому при стационарной теплопроводности суммарная плотность тепловыделения внутри области ($\iint_S q(x, y) dx dy$) должна равняться суммарному потоку энергии через нее

теплоизолированные участки границы области $(\alpha_y \int_0^2 T(x,1)dx + \alpha_x \int_0^1 T(2,y)dy).$

Степень выполнения этого закона в зависимости от числа использованных итераций метода Зейделя демонстрирует таблица 2.4.

Таблица 2.4. Зависимость суммарного потока энергии через границу области в зависимости от числа итераций, использованных в методе Зейделя при решении разностной задачи. Суммарное тепловыделение внутри области для $q(x,y) = (2-x)(1-y)$ равно 1.

Число итераций	500	1000	1500	2000
Суммарный поток через границу области	0.917	0.994	1	1

2.1.4. Нахождение решения линейной системы уравнений с помощью генетического алгоритма

В основе эволюции различных видов животного и растительного мира лежат два процесса: естественный отбор наиболее приспособленных к внешним условиям экземпляров и их генетические изменения от поколения к поколению.

Сегодня эти эволюционные представления широко применяется при решении практических задач в различных областях знаний.

Исследуем возможность применения эволюционного (посуте – итерационного) подхода для решения задач линейной алгебры. В качестве примера такой задачи взята система уравнений 2-го порядка

$$\begin{aligned} A_{11}x_1 + A_{12}x_2 &= b_1 \\ A_{21}x_1 + A_{22}x_2 &= b_2 \end{aligned}$$

Различные варианты эволюционного генетического алгоритма, включающие процедуры отбора и мутации, будем строить следующим образом:

1. Инициализация алгоритма.

- создается популяция объектов выбранного размера – n_pop , каждый экземпляр которой имеют три признака: x_1, x_2 (имитируют компоненты решения системы – его «гены») и $\|\delta\| = \|Ax - b\|$ (сферическая норма вектора невязки для этих значений – характеризует в какой степени этот объект можно рассматривать в качестве решения задачи). Компоненты x_1, x_2 могут задаваться случайными числами $\in [-0.5 \div 0.5]$ либо детерминированными значениями функции $\sin(x_i)$ в равномерной сетке точек $\in [0 \div \pi]$ и $\in [1 \div (1 + \pi)]$. Норма невязки для компонент x_1, x_2 вычисляется обычным образом.

- вводится вероятность мутации каждого гена $prob_mut$. Мутация возможна, если разыгрываемое случайное число $\in [0 \div 1]$ оказывается меньше этой вероятности.

- задается коэффициент мутации k_mut . Если мутация гена допускается, то его значение пересчитывается по формуле (здесь n_mut – случайное число $\in [-0.5 \div 0.5]$)

$$x_i = x_i + k_mut * n_mut$$

- задается число допустимых поколений n_pok .

2. Порядок выполнения одной итерации алгоритма (порядок построения очередного поколения популяции).

- производится сортировка объектов текущей популяции в порядке возрастания их норм (методом «вставки» или быстрая сортировка по Ч. Хоару[3]).

- отбрасывается нижняя половина отсортированных объектов и на их место помещаются новые экземпляры. Они могут формироваться двумя способами: скрещиванием или клонированием. В первом случае из верхней половины массива популяции случайным образом выбирается пара «родителей» и их гены передаются потомку. При этом для передачи генов можно выбрать один из двух вариантов: детерминированный, когда первый ген забирается от первого, второй - от второго, или случайный, когда номер родителя, от которого забирается первый ген, разыгрывается случайным образом. При клонировании нижняя часть массива популяции просто заменяется верхней половиной.

- затем, если это «допускается», над каждым геном потомка выполняется описанная выше процедура мутации.

- все это повторяется n_pok - раз.

Скриншоты приложения, которое реализует описанные эволюционные процедуры, представлены на рис. 1-3.

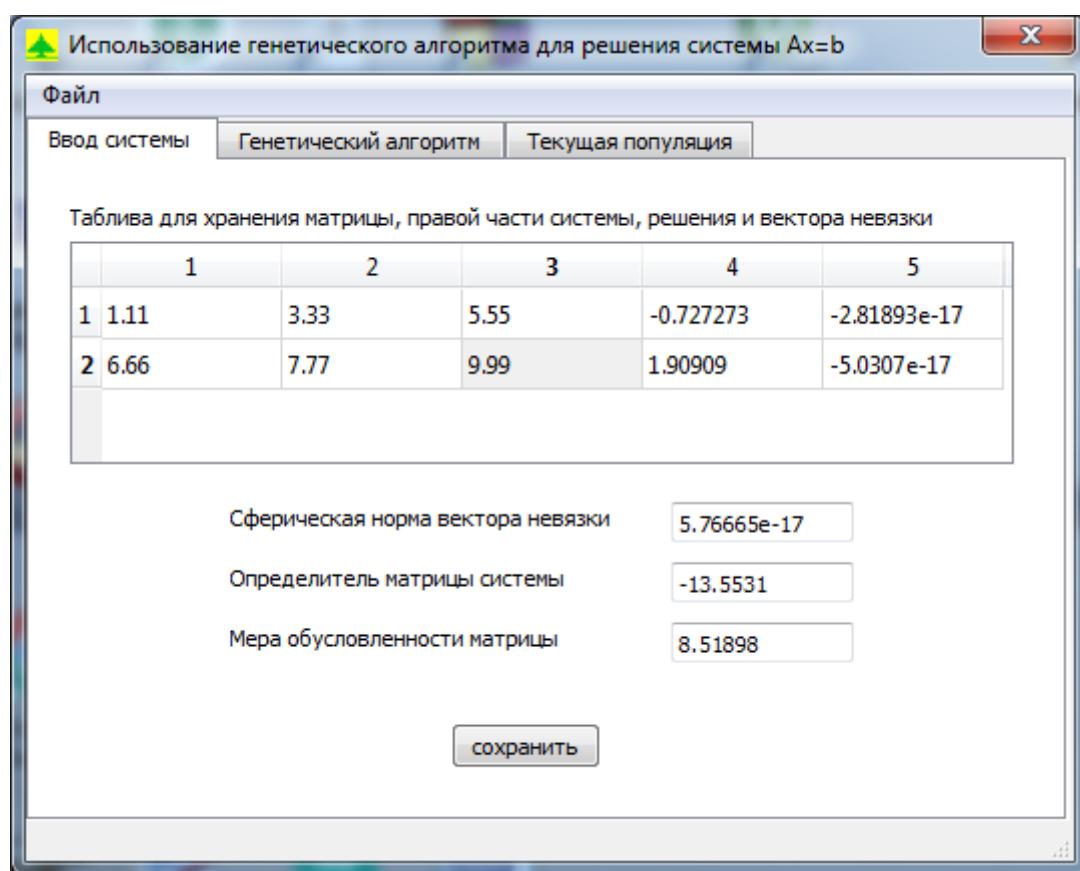


Рис. 1 Закладка для задания исследуемой системы уравнений и определения ее решения методом Крамера.

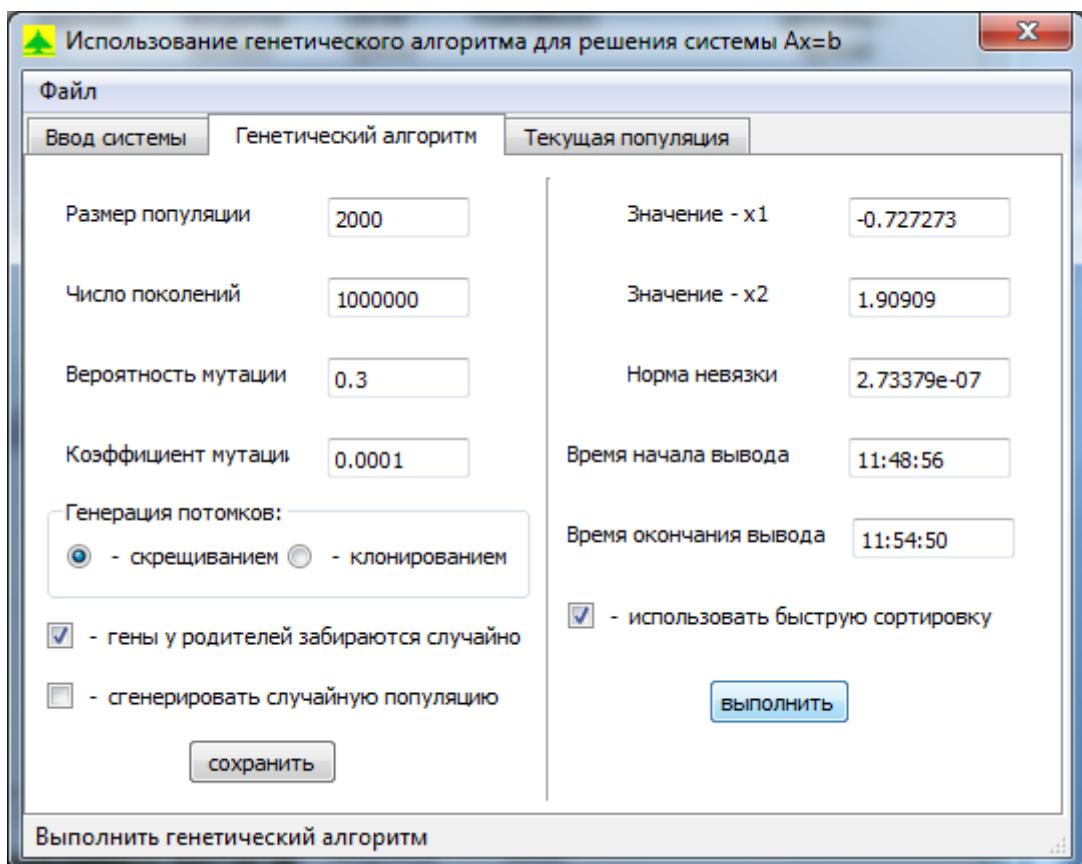


Рис. 2. Закладка для задания параметров генетического алгоритма и запуска выбранного варианта алгоритма на выполнение (с полученными результатами).

Массив популяций: вектор x и сферическая норма его невязки			
	1	2	3
1259	-0.727273	1.90909	2.73379e-07
1260	-0.727273	1.90909	2.73379e-07
1261	-0.727273	1.90909	2.73379e-07
1262	-0.727273	1.90909	2.73379e-07
1263	-0.727272	1.90909	2.65707e-06
1264	-0.727273	1.90909	3.42829e-06
1265	-0.727274	1.90909	5.64594e-06
1266	-0.727273	1.90909	5.77154e-06
1267	-0.727275	1.90909	1.70367e-05
1268	-0.727277	1.90909	4.76562e-05

показать

- выполнить сортировку массива

Вывести в таблицу массив со значениями текущей популяции

Рис. 3. Закладка, на которой выводится текущее состояние популяции. Видно, что в процессе выполнения алгоритма произошло «насыщение» популяции, поскольку число объектов с «оптимальными» конечными значениями «генов» близко к значению $n_pop(1 - prob_mut) = 1400$.

Несколько слов о составных частях приложения.

На первой закладке вводится тестовая система уравнений. По кнопке «сохранить» сохраняются ее параметры, находится решение (методом Крамера), его невязка и ее сферическая норма, определитель матрицы системы и ее мера обусловленности. В дальнейшем эти результаты можно использовать как эталонные при оценке работы генетического алгоритма.

Вторая закладка используется для ввода генетических параметров, генерации начальной популяции (кнопка «сохранить») и запуска на выполнение генетического алгоритма (кнопка «выполнить»). Алгоритм возвращает значения генов (x_1, x_2) первого объекта из последней отсортированной популяции, сферическую норму ее невязки и время начала и окончания работы генетического вывода. Пользователю предоставляется возможность выбора вариантов генерации начального поколения, отбора претендентов в очередное поколение, способа передачи и мутации генов, а также метода сортировки в процессе выполнения алгоритма.

Третья закладка позволяет открыть массив последней популяции. Таким образом можно проверить произошло ли «насыщение» популяции оптимальными объектами (с параметрами как у первого) или следует увеличить число поколений, чтобы «выжить» из заданной популяции все, на что она способна.

Некоторые результаты исследований, проведенных с помощью данного приложения, представлены в таблице 1.

Таблица 1. Результаты исследования работы алгоритма для системы уравнений, изображенной на рис. 1. Использовалась детерминированная начальная популяция, а при скрещивании потомков - случайный выбор генов родителей.

Размер популяции	Число поколений	Вероятность мутации	Коэффициент мутации	Компонент X1	Компонент X2	Норма невязки
скрещивание						
500	500000	0.2	0.01	-0.726623	1.90856	0.00107
500	1000000	0.2	0.001	-0.727283	1.9091	1.6967 E-05
1000	1000000	0.3	0.001	-0.727276	1.90909	5.4187 E-06
2000	1000000	0.3	0.0001	-0.727273	1.90909	2.7338 E-07
клонирование						
500	500000	0.2	0.01	-0.727022	1.90888	0.00042
500	1000000	0.2	0.001	-0.727248	1.90907	4.0567 E-05
1000	1000000	0.3	0.001	-0.727269	1.90909	5.8085

						E-06
2000	1000000	0.3	0.0001	-0.727273	1.90909	2.1488 E-07

Видно, что при размере популяции 2000 и числе поколений 1000000 компоненты x_1, x_2 , полученные с помощью генетического алгоритма, совпадают с решением метода Крамера (рис. 1). Причем, это не зависит от способа генерации потомков – путем скрещивания или клонированием.

Выводы, которые можно сделать после всех выполненных исследований.

1. Если число «оптимальных» объектов популяции имеет порядок $n_pop^*(1 - prob_mut)$, то дополнительные итерации по поколениям не приводят к улучшению результата, поскольку при заданной вероятности мутации генов алгоритм уже «выжил» из данной популяции все возможное.
2. Можно утверждать, что при недостаточном числе итераций на конечный результат заметно влияют, как состав начальной популяции, так и случайные характеристики генетического алгоритма.
3. Увеличение размера популяции приводит к более качественному улучшению результата, нежели при возрастании числа поколений.
4. Для повышения точности вывода следует использовать небольшой коэффициент мутации (например, 0.0001), поскольку это обеспечит более полное (до насыщения ее «оптимальными» экземплярами) использование популяции. Но при этом потребуется сгенерировать большее число поколений.
5. При больших размерах популяции и значительном числе поколений алгоритм выдает хороший результат независимо от выбора способа генерации начальной популяции и процедуры формирования поколений.
6. Работа алгоритма с включением быстрой сортировки выполняется в 3-5 раз быстрее, чем с сортировкой-вставкой (на конкретном компьютере). Последнее связано с тем, что сортировка является наиболее затратной процедурой алгоритма.
7. Подтверждена возможность использования генетических алгоритмов для решения практических задач линейной алгебры. Однако данный подход не выдерживает конкуренции с привычными численными методами по времени получения результата.
8. Генетический алгоритм *совершенно* не пригоден для решения плохо обусловленных задач линейной алгебры.

Вопросы и задания для самоконтроля

1. Почему для окончания итерационного процесса необходимо использовать два (какие) критерия?
2. Запишите выражение для оценки скорости сходимости итерационного процесса. При каком условии оно получено?
3. Почему последовательное увеличение значений нормы $\|x^{(k)} - x^{(k-1)}\|$ при трех последовательных значениях k можно считать признаком расходимости итерационного процесса?

4. Чем принципиально отличается программная реализация метода верхней релаксации от остальных методов?
5. Какие две задачи решает предварительная нормировка системы перед запуском итерационного процесса?
6. Почему по окончании итерационного процесса желательно возвращать пользователю число использованных итераций и значение нормы разности двух соседних приближений?
7. Составьте алгоритм решения системы $Ax = b$ методом Зейделя.
8. Попробуйте реализовать алгоритм метода верхней релаксации на Maxima.

2.2. Прямые методы решения системы линейных уравнений

2.2.1. Решение методом Крамера системы 5-го порядка

Это наиболее известный метод, но его практическое использование, как правило, ограничивается системами 2-го порядка. Рассмотрим все эти вопросы более подробно.

В методе Крамера компоненты вектора решения системы находят по формуле

$$x_i = \det(A \text{ с } b \text{ в } i - \text{ом столбце}) / \det(A),$$

в которой все определители вычисляются через определители миноров для компонент первой строки матрицы. Т.е. через определители, у которых порядок на единицу меньше исходного.

Листинг программы на С для среды программирования CodeBlocks, реализующей этот алгоритм для системы 5-го порядка, приведен ниже. В программе система уравнений хранится в 6-ти векторах-столбцах. Предусмотрена возможность ввода системы с клавиатуры, из заранее созданного текстового файла или ее формирование с помощью псевдослучайных чисел их диапазона [-0.5;0.5]. При вычислении определителя 5-го порядка выполняется последовательный вызов процедур для расчета определителей предыдущих порядков.

Пример наглядно демонстрирует бесперспективность применения этого метода в случае систем высокого порядка из-за сложности его программной реализации алгоритма и неустойчивости к ошибкам округления. Например, в нашем случае для нахождения решения потребовалось порядка 2100 операций (для сравнения метод исключения решает эту задачу за около $n^3/3=41$ операций).

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <time.h>
//прототипы используемых процедур
float det3(float s1[],float s2[],float s3[]);
float det4(float m1[],float m2[],float m3[],float m4[]);
float det5(float m1[],float m2[],float m3[],float m4[],float m5[]);
void not_eqv(float s1[],float s2[],float s3[],float s4[],float s5[],float s6[],float x[],float dx[]);
//главная процедура
int main() {
    float m1[5],m2[5],m3[5],m4[5],m5[5],m6[5];
    float d,x[5],dx[5];
    char str[50],*ptr,c[3];
    FILE *f;
    int fl,i;
    float nf,ni;
    int i1,i2;

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    printf("* Решаем методом Крамера систему из 5-ти линейных уравнений *\n");
    printf("Введите 1/2/3 - ввод системы с клавиатуры/из файла/задать случайным образом ");
    scanf("%d",&fl);
    switch (fl){
```

```

case 1:
    for(i=0;i<5;i++){
        printf("Введите параметры уравнения %d: ",i+1);
        scanf("%f %f %f %f %f",&m1[i],&m2[i],&m3[i],&m4[i],&m5[i],&m6[i]);
    }
    break;
case 2:
    f=fopen("eqv5.txt","r");
    if(f==NULL){
        printf("Ошибка при открытии файла !");
        goto Label;
    }
    printf("Считанная матрица:\n");
    for (i=0;i<5;i++){
        fgets(str,sizeof(str),f);
        printf("%s",str);
        m1[i]=strtod(str,&ptr);m2[i]=strtod(ptr,&ptr);m3[i]=strtod(ptr,&ptr);
        m4[i]=strtod(ptr,&ptr);m5[i]=strtod(ptr,&ptr);m6[i]=strtod(ptr,NULL);
    }
    fclose(f);
    break;
case 3:
    //инициализация датчика псевдослучайных чисел текущим временем
    srand(time(NULL));
    for(i=0;i<5;i++){
        m1[i]=0.5-rand()/(RAND_MAX+1.0);
        m2[i]=0.5-rand()/(RAND_MAX+1.0);
        m3[i]=0.5-rand()/(RAND_MAX+1.0);
        m4[i]=0.5-rand()/(RAND_MAX+1.0);
        m5[i]=0.5-rand()/(RAND_MAX+1.0);
        m6[i]=0.5-rand()/(RAND_MAX+1.0);
    }
    printf("\nСформированная случайная система:");
    for(i=0;i<5;i++){
        printf("\n%.3f %.3f %.3f %.3f %.3f",m1[i],m2[i],m3[i],m4[i],m5[i]);
    }
    break;
default:
    break;
}
d=det5(m1,m2,m3,m4,m5);
printf("\nОпределитель системы = %.3f",d);
if(fabs(d)<1.E-20){      //detA = 0
    printf("\nСистема вырожденная и не имеет решения !");
    goto Label;
    //return 1;
}
x[0]=det5(m6,m2,m3,m4,m5)/d;x[1]=det5(m1,m6,m3,m4,m5)/d;
x[2]=det5(m1,m2,m6,m4,m5)/d;x[3]=det5(m1,m2,m3,m6,m5)/d;
x[4]=det5(m1,m2,m3,m4,m6)/d;
printf("\nРешение системы: %.3f %.3f %.3f %.3f %.3f",x[0],x[1],x[2],x[3],x[4]);
not_eqv(m1,m2,m3,m4,m5,m6,x,dx);

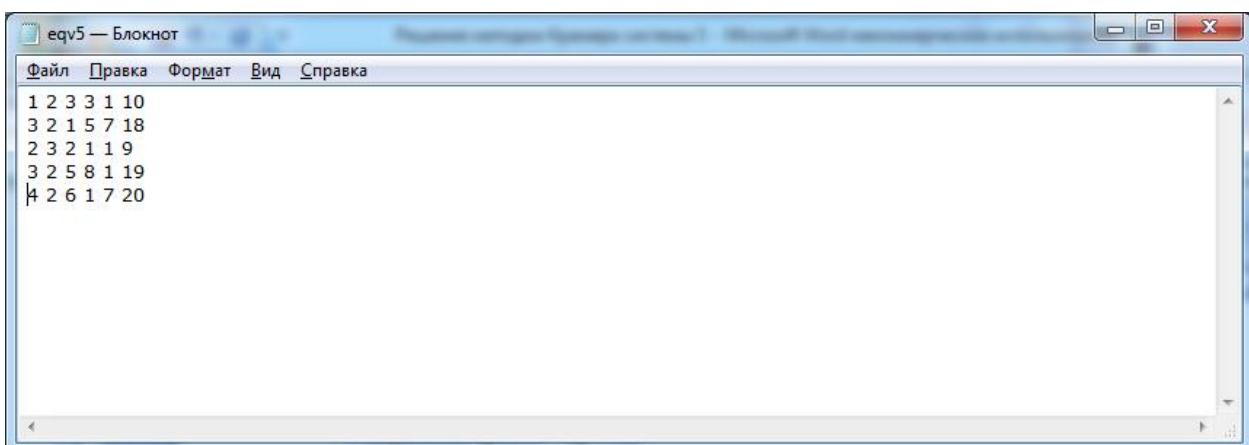
```

```

printf("\nНевязка решения: %.5f %.5f %.5f %.5f %.5f",dx[0],dx[1],dx[2],dx[3],dx[4]);
Label:
printf("\nДля выхода из программы нажмите любую клавишу и ENTER "); scanf("%s",c);
return 0;
}
//рассчитывает определитель 3-го порядка, разложением по минорам первой строки
float det3(float s1[],float s2[],float s3[]) {
    float d;
    d=s1[0]*(s2[1]*s3[2]-s2[2]*s3[1])-s2[0]*(s1[1]*s3[2]-s1[2]*s3[1])+
    s3[0]*(s1[1]*s2[2]-s1[2]*s2[1]);
    return d;
}
//рассчитывает определитель 4-го порядка, разложением по минорам первой строки
float det4(float m1[],float m2[],float m3[],float m4[]){
    float d; float *s1,*s2,*s3;
    s1=&m2[1];s2=&m3[1];s3=&m4[1]; d=m1[0]*det3(s1,s2,s3);
    s1=&m1[1];s2=&m3[1];s3=&m4[1]; d=d-m2[0]*det3(s1,s2,s3);
    s1=&m1[1];s2=&m2[1];s3=&m4[1]; d=d+m3[0]*det3(s1,s2,s3);
    s1=&m1[1];s2=&m2[1];s3=&m3[1]; d=d-m4[0]*det3(s1,s2,s3);
    return d;
}
//рассчитывает определитель 5-го порядка, разложением по минорам первой строки
float det5(float m1[],float m2[],float m3[],float m4[],float m5[]){
    float d; float *s1,*s2,*s3,*s4;
    s1=&m2[1];s2=&m3[1];s3=&m4[1];s4=&m5[1], d=m1[0]*det4(s1,s2,s3,s4);
    s1=&m1[1];s2=&m3[1];s3=&m4[1];s4=&m5[1], d=d-m2[0]*det4(s1,s2,s3,s4);
    s1=&m1[1];s2=&m2[1];s3=&m4[1];s4=&m5[1], d=d+m3[0]*det4(s1,s2,s3,s4);
    s1=&m1[1];s2=&m2[1];s3=&m3[1];s4=&m5[1], d=d-m4[0]*det4(s1,s2,s3,s4);
    s1=&m1[1];s2=&m2[1];s3=&m3[1];s4=&m4[1], d=d+m5[0]*det4(s1,s2,s3,s4);
    return d;
}
//вычисляет невязку для полученного решения  $\delta = Ax - b$ 
void not_eqv(float s1[],float s2[],float s3[],float s4[],float s5[],float s6[],float x[],float dx[]){
    int i;
    for(i=0;i<5;i++){ dx[i]=s1[i]*x[0]+s2[i]*x[1]+s3[i]*x[2]+s4[i]*x[3]+s5[i]*x[4]-s6[i]; }
}

```

Содержание файла - eqv5.txt с системой уравнений 5-го порядка



The screenshot shows a Windows Notepad window with the title 'eqv5 — Блокнот'. The menu bar includes 'Файл', 'Правка', 'Формат', 'Вид', and 'Справка'. The main text area contains the following 5x5 matrix:

```

1 2 3 3 1 10
3 2 1 5 7 18
2 3 2 1 1 9
3 2 5 8 1 19
4 2 6 1 7 20

```

Скриншоты с результатами работы программы

```
C:\Users\admin\Documents\Л_2016\Л\SYSTEM_eq5\main_file.exe
* Решаем методом Крамера систему из 5-ти линейных уравнений *
Введите 1/2/3 - ввод системы с клавиатуры/из файла/задать случайным образом 1
Введите параметры уравнения 1: 1 2 3 4 5 15
Введите параметры уравнения 2: 2 4 1 5 3 15
Введите параметры уравнения 3: 1 2 1 2 1 7
Введите параметры уравнения 4: 2 2 1 1 2 8
Введите параметры уравнения 5: 3 2 1 4 1 11

Определитель системы = 64.000
Решение системы: 1.000 1.000 1.000 1.000 1.000
Невязка решения: 0.00000 0.00000 0.00000 0.00000 0.00000
Для выхода из программы нажмите любую клавишу и ENTER
```

```
C:\Users\admin\Documents\Л_2016\Л\SYSTEM_eq5\main_file.exe
* Решаем методом Крамера систему из 5-ти линейных уравнений *
Введите 1/2/3 - ввод системы с клавиатуры/из файла/задать случайным образом 2
Считанная матрица:
1 2 3 3 1 10
3 2 1 5 7 18
2 3 2 1 1 9
3 2 5 8 1 19
4 2 6 1 7 20

Определитель системы = -1000.000
Решение системы: 1.000 1.000 1.000 1.000 1.000
Невязка решения: 0.00000 0.00000 0.00000 0.00000 0.00000
Для выхода из программы нажмите любую клавишу и ENTER
```

```
C:\Users\admin\Documents\Л_2016\Л\SYSTEM_eq5\main_file.exe
* Решаем методом Крамера систему из 5-ти линейных уравнений *
Введите 1/2/3 - ввод системы с клавиатуры/из файла/задать случайным образом 3
Сформированная случайная система:
-0.031 -0.076 0.434 -0.325 0.466 -0.436
0.301 0.180 0.224 -0.329 0.231 0.443
-0.153 -0.133 -0.431 0.306 -0.336 0.084
0.248 0.282 -0.289 0.007 0.338 0.271
0.437 0.321 0.192 -0.487 -0.256 -0.030
Определитель системы = -0.002
Решение системы: 7.471 -4.995 1.080 3.728 0.333
Невязка решения: -0.00000 0.00000 -0.00000 -0.00000 0.00000
Для выхода из программы нажмите любую клавишу и ENTER
```

2.2.1. Метод исключения Гаусса

В методе Гаусса исходная система $Ax = b$, начиная с левого столбца, путем $(n-1)$ -го последовательного преобразования приводится к системе с верхней треугольной матрицей. Решение последней находится элементарно.

Приведем формулы, по которым выполняется k -ое преобразование. К данному моменту текущая расширенная матрица исходной системы (к исходной матрице добавлен столбец, состоящий из компонентов вектора b) имеет следующий вид

$$A^{(k-1)} = \begin{bmatrix} A_{11} & - & A_{1k} & - & A_{1j} & - & A_{1n+1} \\ 0 & - & - & - & - & - & - \\ 0 & 0 & A_{kk} & - & A_{kj} & - & A_{kn+1} \\ 0 & 0 & - & - & - & - & - \\ 0 & 0 & A_{ik} & - & A_{ij} & - & A_{in+1} \\ 0 & 0 & - & - & - & - & - \\ 0 & 0 & A_{nk} & - & - & - & A_{nn+1} \end{bmatrix} \dots k \dots i \quad (2.21)$$

При k -ом преобразовании в матрице $A^{(k-1)}$ необходимо обнулить все компоненты k -го нижнего подстолбца, начиная с компонента A_{k+1k} .

Последнее осуществляется вычитанием из i -ой строки компонентов k -ой строки, умноженной на отношение $\frac{A_{ik}}{A_{kk}}$

$$A_{ij} = A_{ij} - A_{ik} \frac{A_{kj}}{A_{kk}}, \quad i = (k+1)..n, \quad j = k..(n+1)$$

Алгоритм этой операции представлен на схеме (2.22)

```

for i = (k + 1), n
{
    buf = A_{ik} / A_{kk}
    for j = k, (n + 1)                      (2.22)
    {
        A_{ij} = A_{ij} - buf * A_{kj}
    }
}

```

Замечания:

1. Легко проверить, что преобразование (2.22) эквивалентно матричной операции

$$M^{(k)} A^{(k-1)}$$

где матрица $M^{(k)}$ имеет вид

$$M^{(k)} = \begin{bmatrix} 1 & 0 & - & - & 0 \\ 0 & - & - & - & 0 \\ 0 & - & 1 & - & 0 \\ 0 & - & M_{(k+1)k}^{(k)} & - & 0 \\ 0 & - & - & - & 0 \\ 0 & - & M_{nk}^{(k)} & - & 1 \end{bmatrix}$$

$$\text{где } M_{ik}^{(k)} = -\frac{A_{ik}^{(k-1)}}{A_{kk}^{(k-1)}}.$$

Поэтому операцию приведения матрицы к верхней треугольной можно представить в матричном виде

$$A^{(n-1)} = M^{(n-1)} \dots M^{(1)} A$$

2. Поскольку нулевая нижняя подматрица в дальнейших операциях не используется, то нет необходимости ее явно обнулять. Исходя из этого, в алгоритме (2.22) внутренний цикл можно начинать со значения $k + 1$.

Поместив последовательность операций (2.22) внутрь цикла по полному числу преобразований

$$\text{for } k = 1, (n-1) \{ \text{алгоритм (2.22)} \}, \quad (2.23)$$

получим алгоритм, приводящий исходную систему уравнений к эквивалентной системе с верхней треугольной матрицей.

Для уменьшения влияния ошибок округления и повышения устойчивости работы алгоритма (2.23) в нем перед началом очередного преобразования необходимо выполнять процедуру «выбора ведущего элемента».

Задачи этой процедуры:

- поставить на место k -ой строки строку из оставшихся нижних строк с максимальным по модулю элементом в k -ом нижнем подстолбце;
- проверить, что найденный максимальный элемент не равен нулю (матрица невырожденная);
- осуществить подсчет определителя исходной матрицы.

При вычислении определителя учитывается, что:

1. определитель треугольной матрицы равен произведению ее диагональных компонент;
2. значение определителя не изменяется, если к его строке прибавить другую строку, умноженную на произвольный коэффициент:

$$\begin{vmatrix} A_{11} & A_{12} \\ A_{21} + \lambda A_{11} & A_{22} + \lambda A_{12} \end{vmatrix} = A_{11}(A_{22} + \lambda A_{12}) - A_{12}(A_{21} + \lambda A_{11}) = \\ = A_{11}A_{22} + \lambda A_{11}A_{12} - A_{12}A_{21} - \lambda A_{11}A_{12} = A_{11}A_{22} - A_{12}A_{21}$$

3. при перестановке строк (или столбцов) матрицы знак ее определителя меняется на противоположный:

$$\begin{vmatrix} A_{21} & A_{22} \\ A_{11} & A_{12} \end{vmatrix} = A_{21}A_{12} - A_{11}A_{22} = (-1)(A_{11}A_{22} - A_{21}A_{12})$$

4. умножение всех компонент строки/столбца матрицы на число приводит к умножению определителя этой матрицы на то же число:

$$\begin{vmatrix} \lambda A_{11} & \lambda A_{12} \\ A_{21} & A_{22} \end{vmatrix} = \lambda A_{11}A_{22} - \lambda A_{12}A_{21} = \lambda(A_{11}A_{22} - A_{12}A_{21})$$

Дополнительно приведем еще два свойства определителя, которые будут использоваться в дальнейшем:

1. $\det(A^*B) = \det(A)^*\det(B)$, где А и В – матрицы размерности $n*n$.

2. $\det(A^{-1}) = 1/\det(A)$, т.к. $A^{-1}A = E$ и $\det(E) = 1$.

Справедливость этих соотношений подтверждают результаты выполнения программы на языке пакета Maxima для случайных матриц размерности $5*5$.

```
kill(all);
numer:true;
fpprintprec:6;
ratprint:false;
/* п/п формирует случайную матрицу размерности n*n */
rndA(n):=block
(
  A:zeromatrix(n,n),
  for i thru n do
    for j thru n do A[i,j]:=random(1.0)
);
n:5;
rndA(n); B:copy(A); rndA(n); A; C:A.B;
dC:determinant(C); dA:determinant(A); dB:determinant(B); dAB:dA*dB;
AoBr:invert(A); dAoBr:determinant(AoBr); d:1/dAoBr;
```

(%o7)
$$\begin{bmatrix} 0.3127 & 0.172 & 0.6646 & 0.5639 & 0.8494 \\ 0.7582 & 0.7517 & 0.7489 & 0.5748 & 0.8298 \\ 0.4486 & 0.6736 & 0.7662 & 0.1527 & 0.4558 \\ 0.99 & 0.7793 & 0.3618 & 0.2534 & 0.03571 \\ 0.8776 & 0.7697 & 0.1077 & 0.542 & 0.3899 \end{bmatrix}$$
 - матрица В

(%o9)
$$\begin{bmatrix} 0.1379 & 0.8734 & 0.2847 & 0.3798 & 0.1838 \\ 0.4538 & 0.1262 & 0.3667 & 0.2681 & 0.1242 \\ 0.6826 & 0.3965 & 0.1964 & 0.04239 & 0.1281 \\ 0.2749 & 0.4481 & 0.4637 & 0.7349 & 0.7663 \\ 0.6014 & 0.7129 & 0.004008 & 0.8037 & 0.9441 \end{bmatrix}$$
 - матрица А

$$(\%o10) \begin{bmatrix} 1.37031 & 1.30946 & 1.121 & 0.8191 & 1.05683 \\ 0.7765 & 0.7245 & 0.7875 & 0.5197 & 0.7154 \\ 0.7565 & 0.6794 & 0.9302 & 0.7229 & 1.04979 \\ 2.03376 & 1.859 & 1.22192 & 1.08495 & 1.1417 \\ 2.35464 & 1.9951 & 1.32908 & 1.46492 & 1.50108 \end{bmatrix} - \text{матрица } C = A^*B$$

(%o11) $-2.221309 \cdot 10^{-4}$

(%o12) -0.04756

(%o13) 0.004671

(%o14) $-2.221309 \cdot 10^{-4}$

- значения определителей матриц C , A , B и

$\det(A)^*\det(B)$

$$(\%o15) \begin{bmatrix} -0.4408 & 1.31819 & 0.5055 & -0.9917 & 0.6488 \\ 1.11755 & -1.23212 & 0.7234 & -0.0168 & -0.1399 \\ -0.04933 & -0.5736 & 1.13179 & 2.17701 & -1.83559 \\ 0.9056 & 4.65891 & -3.86521 & -2.6194 & 1.86132 \\ -1.33383 & -3.87324 & 2.41753 & 2.86525 & -0.8253 \end{bmatrix} - \text{матрица } A^{-1}$$

(%o16) -21.0264

(%o17) -0.04756 - определитель матрицы A^{-1} и $1/\det(A^{-1})$

Алгоритм процедуры выбора ведущего элемента приведен на схеме (2.24). Здесь nul - переменная для хранения машинного нуля, используемого для проверки действительных чисел на нуль (обычно $nul \leq 10^{-10}$ и может корректироваться пользователем для фильтрации уровня ошибок округления, возникающих в процессе вычисления).

$\det = 1$ (начальная инициализация перед запуском цикла (2.23))
 -- процедура --
 $k \max = k$
 $A \max = |A_{kk}|$
 $for i = (k + 1), n$
 {
 $A \text{ mod} = |A_{ik}|$
 $if A \text{ mod} > A \max \text{ then}$
 {
 $A \max = A \text{ mod}$
 $k \max = i$
 }
 }
 $if A \max < nul (\approx 0) \text{ then}$
 {
 $\det = 0$ (матрица вырожденная)
 выход из процедуры
 }
 $if k \max \neq k \text{ then}$
 {
 $for j = k, (n + 1) \text{ (переставляем строки)}$
 {
 $buf = A_{kj}$
 $A_{kj} = A_{(k \max) j}$
 $A_{(k \max) j} = buf$
 }
 $\det = -\det$
 }
 $\det = \det^* A_{kk}$ (вычисление определителя)

Замечание. Перестановка i и j строк эквивалентна матричной операции
 $T(i, j)A$
 где

$$T(i, j) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{matrix} i \\ j \end{matrix}$$

- матрица перестановок (на примере матрицы 6x6 и $i = 3, j = 5$).

Она обладает следующими свойствами:

1. Матричная операция $T(i, j)A$ переставляет строки, например,

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{21} & A_{22} \\ A_{11} & A_{12} \end{bmatrix}$$

2. Матричная операция $AT(i, j)$ переставляет столбцы, например,

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} A_{12} & A_{11} \\ A_{22} & A_{21} \end{bmatrix}$$

3. $T(i, j)T(i, j) = E$, например,

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

После приведения системы уравнений к системе с треугольной матрицей

$$\begin{aligned} A_{11}x_1 + \dots + A_{1i}x_i + \dots + A_{1j}x_j + \dots &= A_{1(n+1)} \\ \dots &\dots \dots \dots \dots \\ A_{ii}x_i + \dots + A_{ij}x_j + \dots &= A_{i(n+1)} \\ &\dots \dots \dots \\ A_{nn}x_n &= A_{n(n+1)} \end{aligned} \tag{2.25}$$

легко записать алгоритм решения последней (учтено, что правые части уравнений хранятся в дополнительном столбце расширенной матрицы - $A_{i(n+1)}, i = 1, n$)

```

if |Ann| < nul then {det = 0, exit sub} else det = det * Ann
xn = An(n+1) / Ann
for i = (n - 1), 1 step = -1
{
    xi = 1 / Aii (Ai(n+1) - Σj=(i+1)n Aij xj)
}

```

(2.26)

Здесь первый условный оператор завершает проверку обусловленности матрицы и подсчет значения определителя.

2.2.2. Метод оптимального исключения Гаусса-Жордана

В отличие от метода Гаусса в данном методе исходная система $Ax = b$ с помощью n последовательных преобразований сводится к системе с единичной матрицей. При этом решение системы располагается на месте правого крайнего столбца расширенной матрицы преобразованной системы (добавлен столбец, состоящий из компонентов вектора b)

$$Ex \equiv LAx = Lb, \text{ где } L = L^{(n)} \dots L^{(1)}$$

Приведем формулы, по которым выполняется k -ое преобразование. К данному моменту текущая расширенная матрица исходной системы имеет следующий вид

$$A^{(k-1)} = \begin{bmatrix} 1 & - & A_{1k} & - & A_{1j} & - & A_{1n+1} \\ - & - & - & - & - & - & - \\ 0 & - & A_{kk} & - & A_{kj} & - & A_{kn+1} \\ - & - & - & - & - & - & - \\ 0 & - & A_{ik} & - & A_{ij} & - & A_{in+1} \\ - & - & - & - & - & - & - \\ 0 & - & A_{nk} & - & - & - & A_{nn+1} \end{bmatrix} \dots k \dots i$$

Первоначально компонент матрицы A_{kk} приводится к единице, путем деления k -ой правой подстроки на компонент A_{kk}

$$A_{kj} = \frac{A_{kj}}{A_{kk}}, \quad j = k .. (n + 1) \quad (2.27)$$

Затем обнуляются все остальные компоненты k -го столбца расширенной матрицы, путем вычитания из каждой i -ой строки k -ой строки, умноженной на компонент A_{ik}

$$A_{ij} = A_{ij} - A_{ik} A_{kj}, \quad i = 1..n, i \neq k; j = k .. (n + 1) \quad (2.28)$$

Замечание. Легко проверить, что преобразования матрицы (2.27) и (2.28) эквивалентно матричной операции

$$L^{(k)} A^{(k-1)}$$

где $A^{(k-1)}$ - исходная матрица после $(k-1)$ -го преобразования, а матрица $L^{(k)}$ имеет вид (отличается от единичной матрицы, размерности $n \times n$, только k -ым столбцом)

$$L^{(k)} = \begin{vmatrix} 1 & - & L_{1k}^{(k)} & - & 0 \\ - & - & - & - & - \\ 0 & - & L_{kk}^{(k)} & - & 0 \\ - & - & - & - & - \\ 0 & 0 & L_{nk}^{(k)} & - & 1 \end{vmatrix} \quad (2.29)$$

где $L_{ik}^{(k)} = -\frac{A_{ik}^{(k-1)}}{A_{kk}^{(k-1)}}, i = 1, n, i \neq k, L_{ik}^{(k)} = \frac{1}{A_{kk}^{(k-1)}}$.

Приведем алгоритм, реализующий данный метод

$\det = 1$ (*переменная для хранения значения определителя*)

for $k = 1, n$

{

вызов процедуры выбора ведущего элемента

for $j = k + 1, n + 1$

{

$$A_{kj} = \frac{A_{kj}}{A_{kk}}$$

}

for $i = 1, n$

{

if $i \neq k$ *then*

{

for $j = k + 1, n + 1$

{

$$A_{ij} = A_{ij} - A_{ik} A_{kj}$$

}

(2.30)

}

}

}

Здесь:

1. в пределах каждого k -го преобразования k -ый столбец явно не обрабатывается, поскольку он в последующих преобразованиях не используется;
2. если в процессе преобразований необходимо явно сформировать единичную матрицу, то заголовок первого цикла следует заменить на `for j = n + 1, 1 step -1`, а второй цикл начинать с $j = k$;
3. каждое очередное преобразование матрицы предваряется процедурой выбора ведущего элемента, которая приведена ниже.

```

 $k \max = k$  ,  $A \max = |A_{kk}|$ 
if  $k < n$  then
{
    for  $i = (k + 1), n$ 
    {
         $A \text{ mod} = |A_{ik}|$ 
        if  $A \text{ mod} > A \max$  then
        {
             $A \max = A \text{ mod}$  ,  $k \max = i$ 
        }
    }
}
if  $A \max < nul (\approx 0)$  then
{
    det = 0 (матрица вырожденная)
    выход из процедуры
}
if  $k \max \neq k$  then
{
    for  $j = k, (n + 1)$  (переставляем строки)
    {
        buf =  $A_{kj}$  ,  $A_{kj} = A_{(k \max) j}$ 
         $A_{(k \max) j} = buf$ 
    }
    det = -det
}
det = det *  $A_{kk}$  (вычисление определителя)

```

Переменная *det* используется процедурой выбора ведущего элемента для хранения значения определителя, а *nul*- для хранения машинного нуля, используемого для проверки действительных чисел на нуль (обычно $nul \leq 10^{-10}$ и может корректироваться пользователем для фильтрации уровня ошибок округления, возникающих в процессе вычисления).

Программа для данного метода на С, написанная в среде CodeBlocks, приведена ниже

```
#include <stdio.h>
```

```

#include <stdlib.h>
#include <windows.h>
#include <math.h>
#include <time.h>
#define B(i,j) MM[i*(n+1)+j] //исходная система
#define A(i,j) M[i*(n+1)+j] //приводимая система

float *M,*MM,det;
int n,m;      //размерность и число перестановок
int n1,n2;    //вспомогательные переменные

void metod_G_J();
void select_Amax(int k);
void b_Ax();

int main()
{
    int i,j,c;
    char s[2];
    SetConsoleCP(1251);           //чтобы воспринимала русские буквы
    SetConsoleOutputCP(1251);
    printf("*** Программа находит решение системы Ax=b методом Гаусса-Жордано ***");
    printf("\nВведите размерность системы: ");
    scanf("%d",&n); n1=n-1; n2=n+1;
    M=malloc(n*(n+1)*sizeof(float)); //расширенная матрица - (A+b)
    MM=malloc(n*(n+1)*sizeof(float));
    printf("Введите 0/1 - задать систему случайным образом или вручную ");
    scanf("%d",&c);
    switch (c){
        case 0:
            //инициализация датчика п.с. чисел текущим временем
            srand(time(NULL));
            for(i=0;i<n;i++)for(j=0;j<n+1;j++)B(i,j)=0.5-rand()/(RAND_MAX+1.0); //[-0.5;0.5]
            break;
        default:
            for (i=0;i<n;i++){
                for (j=0;j<n;j++){
                    printf("Введите A(%d,%d): ",i,j);
                    scanf("%f",&B(i,j));
                }
                printf("Введите b(%d): ",i);
                scanf("%f",&B(i,n));
            }
    }
    printf("Введенная система:\n");
    for (i=0;i<n;i++){
        for (j=0;j<n;j++){
            printf("%.5f ",B(i,j));
            A(i,j)=B(i,j);
        }
        printf("%.5f\n",B(i,n));
        A(i,n)=B(i,n);
    }
}

```

```

}
metod_G_J();
printf("Определитель матрицы системы = %.5f",det);
if (fabs(det)<1.E-10){
    printf("\nМатрица системы ВЫРОЖДЕННАЯ !");
}
else{
    printf("\nРешение системы:\n");
    for(i=0;i<n;i++){
        printf("%.5f ",A(i,n));
    }
    printf("\nЧисло переставленных строк - %d",m);
    b_Ax();
    printf("\nНевязка для полученного решения:\n");
    for(i=0;i<n;i++){
        printf("%.5f ",B(i,n));
    }
}
printf("\nДля завершения программы введите любой символ и ENTER");
scanf("%s",&s);
return 0;
}
//п/п вычисляет невязку для полученного решения
void b_Ax(){
int i,j;
for(i=0;i<n;i++){
    for(j=0;j<n;j++) B(i,n)=B(i,n)-B(i,j)*A(j,n);
}
}
//п/п реализует метод Гаусса-Жордано
void metod_G_J(){
int i,j,k;
float buf;
det=1;m=0;
//приводим матрицу к единичному виду
for (k=0;k<n;k++) //цикл по преобразованиям
{
    select_Amax(k);
    if (fabs(det)<1.E-10){return;}
    for (j=n;j>k-1;j--){
        A(k,j)=A(k,j)/A(k,k);
    }
/*
    buf=A(k,k); //можно и так
    for (j=k;j<n+1;j++) A(k,j)=A(k,j)/buf;
*/
    for (i=0;i<n;i++){
        if (i!=k){
            for (j=n;j>k-1;j--){
                A(i,j)=A(i,j)-A(i,k)*A(k,j);
            }
        }
    }
}
}

```

```

    }
}

printf("Система преобразована к виду; \n");
for (i=0;i<n;i++){
    for (j=0;j<n;j++){
        printf("%.5f ",A(i,j));
    }
    printf("%.5f\n",A(i,n));
}
//процедура выбора ведущего элемента в k-ом подстолбце
void select_Amax(int k){
    int i,kmax,j,k1=k+1;
    float Amod,Amax,buf;
    Amax=fabs(A(k,k)); kmax=k;
    if (k<n-1){
        for (i=k1;i<n;i++){
            Amod=fabs(A(i,k));
            if (Amod>Amax){Amax=Amod; kmax=i;}
        }
    }
    if (Amax<1.E-10){ //матрица системы вырожденная
        det=0;return;
    }
    if (kmax!=k){
        for (j=k;j<n+1;j++){ //переставляем строки
            buf=A(k,j);
            A(k,j)=A(kmax,j);
            A(kmax,j)=buf;
        }
        //printf("TECT => A(k,k), A(k,n) = %.3f %.3f\n",A(k,k),A(k,n));
        det=-det;
        m++; //число перестановок строк
    }
    det=det*A(k,k);
}

```

Скриншот с результатами ее работы

```

*** Программа находит решение системы Ax=b методом Гаусса-Жордано ***
Введите размерность системы: 3
Введите 0/1 - задать систему случайным образом или вручную 0
Введенная система:
-0.29910 0.41681 -0.23291 0.11920
-0.33704 0.30399 -0.15411 0.48016
0.33295 0.40079 0.13953 0.13416
Система преобразована к виду;
1.00000 0.00000 0.00000 -2.69147
0.00000 1.00000 0.00000 0.95226
0.00000 0.00000 1.00000 4.64872
Определитель матрицы системы = 0.02209
Решение системы:
-2.69147 0.95226 4.64872
Число переставленных строк - 2
Невязка для полученного решения:
0.00000 -0.00000 -0.00000
Для завершения программы введите любой символ и ENTER

```

2.2.3. Нахождение обратной матрицы методом оптимального исключения Гаусса-Жордана

Полное преобразование матрицы методом оптимального исключения может быть записано в матричном виде

$$L^{(n)}T^{(n)} \dots L^{(1)}T^{(1)}A = E,$$

из которого следует матричная форма нахождения обратной матрицы (т.к. по определению $A^{-1}A = E$)

$$A^{-1} = L^{(n)}T^{(n)} \dots L^{(1)}T^{(1)}E \quad (2.31)$$

где $T^{(i)}$ - матрица перестановок при i - ом преобразовании, а $L^{(i)}$ задано (2.29).

Согласно выражению (2.31), если исходную матрицу A дополнить справа единичной матрицей E размерности $n \times n$, то в качестве алгоритма нахождения обратной матрицы можно использовать алгоритм (2.30), заменив в нем правые пределы циклов со значения $n+1$ на $n+n$. При этом ту же замену необходимо выполнить и в процедуре выбора ведущего элемента (2.24) при перестановке строк. После n преобразований на месте единичной матрицы будет располагаться обратная матрица.

В противовес сказанному, можно предложить более экономичный (по размеру используемой памяти и количеству операций) алгоритм нахождения обратной матрицы. В нем выбор ведущего элемента осуществляется в k -ой правой подстроке с перестановкой соответствующих столбцов. В этом случае полное преобразование исходной матрицы будет записываться в следующем матричном виде

$$L^{(n)} \dots L^{(1)}AT^{(1)} \dots T^{(n)} = E$$

Отсюда последовательно получим

$$AT^{(1)} \dots T^{(n)} = (L^{(n)} \dots L^{(1)})^{-1} E \Rightarrow (AT^{(1)} \dots T^{(n)})^{-1} = (L^{(n)} \dots L^{(1)})E \Rightarrow$$

$$(T^{(1)} \dots T^{(n)})^{-1} A^{-1} = (L^{(n)} \dots L^{(1)})E \Rightarrow A^{-1} = (T^{(1)} \dots T^{(n)})(L^{(n)} \dots L^{(1)})E$$

Таким образом, если единичную матрицу подвергнуть тем же преобразованиям (кроме перестановки столбцов), что и матрицу A, то для окончательного построения обратной матрицы из преобразованной единичной матрицы необходимо в последней переставить между собой строки с номерами, которые соответствуют номерам переставленных в процессе преобразования столбцов, но в обратной последовательности. Естественно, номера переставляемых столбцов предварительно должны быть сохранены в двумерном вспомогательном массиве.

Обсудим детали оптимального алгоритма расчета обратной матрицы. Обозначим соответствующие матрицы после $(k-1)$ -ого преобразования

$$A^{(k-1)} = L^{(k-1)} \dots L^{(1)} AT^{(1)} \dots T^{(k-1)}$$

$$B^{(k-1)} = L^{(k-1)} \dots L^{(1)} E$$

Они имеют вид

$$A^{(k-1)} = \begin{bmatrix} 1 & - & 0 & A_{1k}^{(k-1)} & - & A_{1n}^{(k-1)} \\ - & - & - & - & - & - \\ 0 & - & 1 & A_{k-1k}^{(k-1)} & - & A_{k-1n}^{(k-1)} \\ 0 & - & 0 & A_{kk}^{(k-1)} & - & A_{kn}^{(k-1)} \\ - & - & - & - & - & - \\ 0 & - & 0 & A_{nk}^{(k-1)} & - & A_{nn}^{(k-1)} \end{bmatrix}$$

$$B^{(k-1)} = \begin{bmatrix} B_{11}^{(k-1)} & - & B_{1k-1}^{(k-1)} & 0 & - & 0 \\ - & - & - & - & - & - \\ B_{k-11}^{(k-1)} & - & B_{k-1k-1}^{(k-1)} & 0 & - & 0 \\ B_{1k}^{(k-1)} & - & B_{kk-1}^{(k-1)} & 1 & - & 0 \\ - & - & - & - & - & - \\ B_{n1}^{(k-1)} & - & B_{nk-1}^{(k-1)} & 0 & - & 1 \end{bmatrix}$$

Обратите внимание на следующие обстоятельства:

1. В первой матрице левая часть, а во второй правая совпадают с фрагментом единичной матрицы, причем эти фрагменты таковы, что дополняют друг друга до полной единичной матрицы.
2. В процессе k -ого преобразования в первой матрице k -ый столбец переходит в k -ый столбец единичной матрицы (поскольку он в дальнейшем не используется, то его явно можно не обрабатывать), а во второй матрице появляется k -ый столбец, отличный от k -ого столбца единичной матрицы.

Все это позволяет сделать вывод, что обе матрицы можно хранить на месте исходной и во время k -ого преобразования обрабатывать по единым формулам (2.32), которые получены в результате обобщения матричных операций $A^{(k)} = L^{(k)}(A^{(k-1)}T^{(k)})$, $B^{(k)} = L^{(k)}B^{(k-1)}$.

выбор ведущего элемента в k -ой правой подстроке ($\neq 0$)

$$\begin{aligned} A_{ij} &= A_{ij} - \frac{A_{ik}A_{kj}}{A_{kk}} \quad i, j = 1, n; i \neq k, j \neq k \\ A_{ik} &= -\frac{A_{ik}}{A_{kk}} \quad i = 1, n; i \neq k \\ A_{kj} &= \frac{A_{kj}}{A_{kk}} \quad j = 1, n; j \neq k \\ A_{kk} &= \frac{1}{A_{kk}} \end{aligned} \tag{2.32}$$

В заключении приведем оптимальный алгоритм в развернутом виде

$\det = 1, m = 0$ (инициализация переменных)

for $k = 1, n$

{

вызов процедуры выбора ведущего элемента в подстроке (см. ниже)

for $i = 1, n$

{

if $i \neq k$ then

{

$$A_{ik} = -\frac{A_{ik}}{A_{kk}}$$

for $j = 1, n$ { if $j \neq k$ then { $A_{ij} = A_{ij} + A_{ik}A_{kj}$ } }

}

}

for $j = 1, n$

{

$$\text{if } j \neq k \text{ then } \{ A_{kj} = \frac{A_{kj}}{A_{kk}} \}$$

}

$$A_{kk} = \frac{1}{A_{kk}}$$

}

вызов процедуры, переставляющей строки в обработанной матрице (см. ниже)

Здесь, если возвращается нулевое значение определителя матрицы, то матрица – вырожденная или плохо обусловлена.

Алгоритм процедуры выбора ведущего элемента. Здесь для хранения номеров переставляемых столбцов используется массив M_{ij} $i = 1, 2; j = 1, n$, а в переменной m хранится число выполненных перестановок.

```

 $k \max = k, A \max = |A_{kk}|$ 
if  $k < n$  then
{
    for  $j = (k + 1), n$ 
    {
         $A \text{mod} = |A_{kj}|$ 
        if  $A \text{mod} > A \max$  then {  $A \max = A \text{mod}, k \max = j$  }
    }
}
if  $A \max < \text{nul}$  ( $= 1E^{-30} \approx 0$ ) then
{
     $\det = 0$  (матрица вырожденная), выход из процедуры
}
if  $k \max \neq k$  then
{
    for  $i = 1, n$  (переставляем столбцы)
    {
         $buf = A_{ik}, A_{ik} = A_{ik \max}, A_{ik \max} = buf$ 
    }
     $\det = -\det, m = m + 1, M_{1m} = k, M_{2m} = k \max$ 
}
 $\det = \det * A_{kk}$  (вычисление определителя)

```

Процедура, переставляющая строки в преобразованной матрице

```

if  $m = 0$  then выход из процедуры
for  $l = m, 1$  step  $= -1$  (переставляем строки в обратном порядке)
{
     $m1 = M_{1l}$ 
     $m2 = M_{2l}$ 
    for  $j = 1, n$ 
    {
         $buf = A_{m1j}$ 
         $A_{m1j} = A_{m2j}$ 
         $A_{m2j} = buf$ 
    }
}

```

В заключении, в качестве примера, приведем экранную форму (рис. 2.11) программы, написанной на языке C^{++} с использованием библиотеки Qt, которая для введенной матрицы рассчитывает (использованы описанные выше алгоритмы) ее определитель и обратную матрицу. Пользователь последовательно задает размерность матрицы и вводит значения ее компонент или формирует ее случайным образом.

После «кликания» управляющей кнопки «найти обратную матрицу» программа на месте исходной матрицы формирует рассчитанную обратную матрицу. Одновременно выдается значение определителя исходной матрицы и число использованных перестановок столбцов.

Далее полученную матрицу можно протестировать, проверив выполнение соотношения $A(\text{обр})^*A = E$ (рис. 2.12).

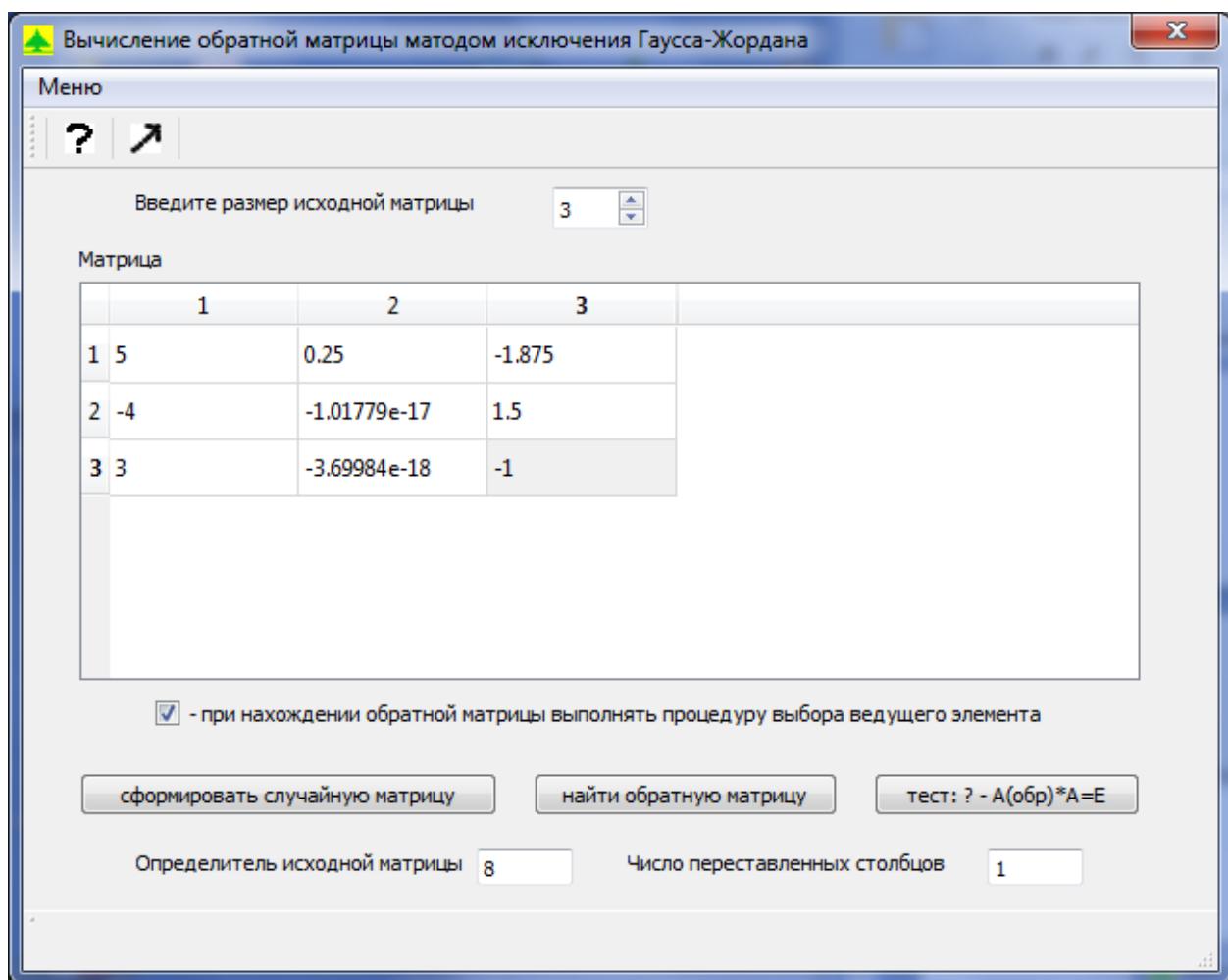


Рис. 2.11. Экранная форма программы, на которой сформирована обратная матрица для исходной матрицы $\begin{bmatrix} 0 & 2 & 3 \\ 4 & 5 & 0 \\ 0 & 6 & 8 \end{bmatrix}$. Заметим, что если с помощью переключателя отключить процедуру выбора ведущего элемента, то определить обратную матрицу не удастся. Последнее подтверждает необходимость использования указанной процедуры в рассмотренных выше алгоритмах.

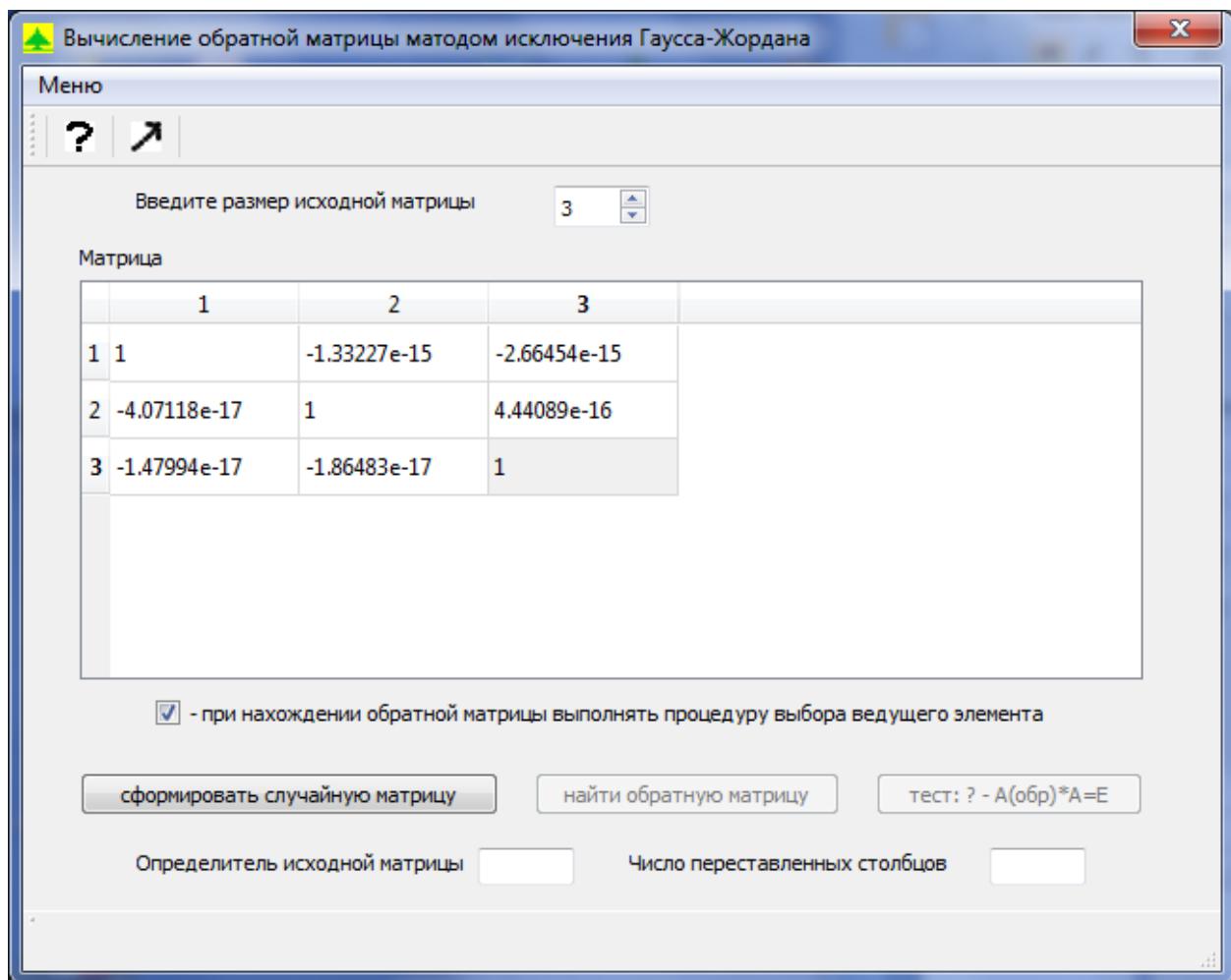


Рис. 2.12. Результаты тестирования полученной обратной матрицы.

Ниже представлены листинги программ данного приложения, которые запускаются при нажатии кнопок «сформировать случайную матрицу», «найти обратную матрицу» и «тест ? – $A(\text{обр})^*A=E$ ».

```
// слот -> формируем случайную матрицу
void MainWindow::on_pushButton_2_clicked()
{
    int i,j;
    double num;
    for (i=0; i<n; i++)
    {
        for (j=0; j<n; j++)
        {
            num=(double(rand()) / RAND_MAX - 0.5) * 2; // формируем
            случайное число из диапазона (-1,1)
            QTableWidgetItem *item=new QTableWidgetItem;
            item->setText(QString("%1").arg(num));
            ui->tableWidget->setItem(i, j, item);
        }
    }
    ui->lineEdit->clear();
}
```

```

ui->lineEdit_2->clear();
//QMessageBox::information(0,tr("Сообщение"),tr("Готово"));
ui->statusBar->showMessage(tr("!!! - СЛУЧАЙНАЯ МАТРИЦА
СФОРМИРОВАНА - !!!"),2000);
ui->pushButton->setEnabled(true); //делаем кнопку доступной
ui->pushButton_3->setEnabled(false); //недоступной
n_n=0;
fl_enb=1;
}

// нахождение обратной матрицы методом Гаусса-Жордана
void MainWindow::on_pushButton_clicked()
{
    int i,j,k,m0,m1,rang;
    det=1; m=0; rang=n;
    for (i=0; i<n; i++) // Инициируем матрицу из таблицы
    {
        for (j=0; j<n; j++)
        {
            A[i][j]=ui->tableWidget->item(i, j)-
>text().toDouble(&ok);
            if (ok==false)
            {
                Int warn
                =QMessageBox::warning(0,tr("Предупреждение"),
tr("Ошибка в задании компонента/ов матрицы"));
                return;
            }
            B[i][j]=A[i][j]; //дублируем исходную матрицу
        }
    }
    for (k=0;k<n;k++)
    {
        if (fl_check==1)
        {
            select_Amax(k); // выбор ведущего элемента
        }
        else
        {
            if (fabs(A[k][k])<pow(10,-30))
            {
                det=0;
                int warn=QMessageBox::warning(0,tr("Предупреждение"),
tr("У матрицы встретился диагональный элемент < 10 E-30. Процесс
нахождения обратной матрицы прерван!"));
                return;
            }
        };
        if (det==0) {rang=k; break;}
        for (i=0;i<n;i++)
        {
            if (i!=k)
            {

```

```

        A[i][k]=-A[i][k]/A[k][k];
        for (j=0;j<n;j++)
        {
            if (j!=k)
            {
                A[i][j]=A[i][j]+A[i][k]*A[k][j];
            }
        }
    }
    for (j=0;j<n;j++)
    {
        if (j!=k)
        {
            A[k][j]=A[k][j]/A[k][k];
        }
    }
    A[k][k]=1/A[k][k];
}
if (m>0)           // переставляем строки
{
    for (i=(m-1);i>=0;i--)
    {
        m0=L[0][i];
        m1=L[1][i];
        for (j=0;j<n;j++)
        {
            buf=A[m0][j];
            A[m0][j]=A[m1][j];
            A[m1][j]=buf;
        }
    }
}
ui->lineEdit->setText(QString("%1").arg(det));
ui->lineEdit_2->setText(QString("%1").arg(m));
ui->lineEdit_3->setText(QString("%1").arg(rang));
if (det!=0)
{
    for (i=0; i<n;i++)      // Вставляем обратную матрицу в таблицу
    {
        for (j=0; j<n;j++)
        {
            QTableWidgetItem *item=new QTableWidgetItem;
            item->setText(QString("%1").arg(A[i][j]));
            ui->tableWidget->setItem(i,j,item);
        }
    }
    ui->pushButton_3->setEnabled(true);      // делаем кнопку
    "тест.." доступной
    ui->statusBar->showMessage(tr("!!! - ОБРАТНАЯ МАТРИЦА
НАЙДЕНА - !!!"),2000); //текст отображается в течение 2 сек.
}
else

```

```

    {
        int
warn=QMessageBox::warning(0,tr("Предупреждение"),tr("Матрица
ВЫРОЖДЕННАЯ"));
    }
}

// процедура выбора ведущего элемента
void select_Amax(int k)
{
    int i,kmax,j;
    double Amod,Amax;
    if(k<(n-1))
    {
        Amax=fabs(A[k][k]); kmax=k;
        for (j=(k+1); j<n; j++)
        {
            Amod=fabs(A[k][j]);
            if (Amod>Amax)
            {
                Amax=Amod; kmax=j;
            }
        }
        if (Amax<pow(10,-30)) // матрицы вырожденная
        {
            det=0;
            return;
        }
        if (kmax!=k)
        {
            for (i=0;i<n;i++) // переставляем столбцы
            {
                buf=A[i][k];
                A[i][k]=A[i][kmax];
                A[i][kmax]=buf;
            }
            det=-det;
            L[0][m]=k;
            L[1][m]=kmax;
            m++;
        }
    }
    det=det*A[k][k];
}

// расчитываем произведение A(обр) *A
void MainWindow::on_pushButton_3_clicked()
{
    int i,j,k;
    for (i=0; i<n; i++)
    {
        for (j=0; j<n; j++)
        {

```

```

E[i][j]=0;
for (k=0; k<n; k++)
{
    E[i][j]=E[i][j]+A[i][k]*B[k][j];
}
}
for (i=0; i<n; i++) // Вставляем "единичную" матрицу в
таблицу
{
    for (j=0; j<n; j++)
    {
        QTableWidgetItem *item=new QTableWidgetItem;
        item->setText(QString("%1").arg(E[i][j]));
        ui->tableView->setItem(i, j, item);
    }
}
ui->lineEdit->clear();
ui->lineEdit_2->clear();
ui->pushButton->setEnabled(false); //делаем кнопки
недоступными
ui->pushButton_3->setEnabled(false);
ui->statusBar->showMessage(tr("!!! - ТЕСТ ВЫПОЛНЕН -
!!!"), 2000);
n_n=0;
fl_enb=0;
}

```

Консольный аналог подобной программы на языке С в среде программирования CodeBlocks имеет вид

```

#include <math.h>
#include <time.h>
#define B(i,j) MM[i*n+j] //исходная система
#define A(i,j) M[i*n+j] //приводимая система
#define E(i,j) ME[i*n+j] //под единичную матрицу
#define L(i,j) LL[i*n+j] //массив для номеров переставленных столбцов

float *M,*MM,*ME,*LL,det=1;
int n,m; //размерность и число перестановок
int n1,n2;
void fine();
void select_Amax(int k);
void test();

int main()
{
    int i,j,c;
    char s[2];
    SetConsoleCP(1251); //чтобы воспринимала русские буквы
    SetConsoleOutputCP(1251);
    printf(" *** Программа находит обратную матрицу методом Гаусса-Жордано *** ");

```

```

printf("\nВведите размерность исходной матрицы: ");
scanf("%d",&n);
M=malloc(n*n*sizeof(float)); //матрица - A
MM=malloc(n*n*sizeof(float)); //копия A - B
ME=malloc(n*n*sizeof(float)); //для теста единичной матрицы
LL=malloc(2*n*sizeof(float)); //хранит номера столбцов
n1=n-1; //чтобы обращаться к компонентам матриц A, B, L
printf("Введите 0/1 - задать систему случайным образом или вручную ");
scanf("%d",&c);
switch (c){
    case 0:
        //инициализация датчика п.с. чисел текущим временем
        srand(time(NULL));
        for(i=0;i<n;i++)for(j=0;j<n;j++)B(i,j)=0.5-rand()/(RAND_MAX+1.0); //[-0.5;0.5]
        break;
    default:
        for (i=0;i<n;i++){
            for (j=0;j<n;j++){
                printf("Введите A(%d,%d): ",i,j);
                scanf("%f",&B(i,j));
            }
        }
    }
printf("Введенная матрица:\n");
for (i=0;i<n;i++){
    for (j=0;j<n;j++){
        printf("%.5f ",B(i,j));
        A(i,j)=B(i,j);
    }
    printf("\n");
}
//printf("A(n-1,n-1) = %.3f\n",A(n1,n1));
fine();
printf("Определитель матрицы системы = %.5f",det);
if (fabs(det)<1.E-10){
    printf("\nМатрица системы ВЫРОЖДЕННАЯ !");
}
else{
    printf("\nОбратная матрица:\n");
    for (i=0;i<n;i++){
        for (j=0;j<n;j++){
            printf("%.5f ",A(i,j));
        }
        printf("\n");
    }
    printf("\nЧисло переставленных строк - %d",m);
    test();
    printf("\nПроизведение обратной матрицы на исходную:\n");
    for (i=0;i<n;i++){
        for (j=0;j<n;j++){
            printf("%.5f ",E(i,j));
        }
    }
}

```

```

        printf("\n");
    }
}
printf("\nДля завершения программы введите любой символ и ENTER");
scanf("%s",&s);
return 0;
}
//вычисляет произведение обратной матрицы на исходную
void test(){
int i,j,k;
for(i=0;i<n;i++){
for(j=0;j<n;j++){
E(i,j)=0;
for(k=0;k<n;k++){
E(i,j)=E(i,j)+A(i,k)*B(k,j);
}
}
}
}
//процедура выбора ведущего элемента в k-ой подстроке
void select_Amax(int k){
int i,kmax,j;
float Amod,Amax,buf;
if(k<(n-1))
{
Amax=fabs(A(k,k)); kmax=k;
for (j=(k+1);j<n;j++)
{
Amod=fabs(A(k,j));
if (Amod>Amax)
{
Amax=Amod; kmax=j;
}
}
if (Amax<1.E-10) // матрицы вырожденная
{
det=0;
return;
}
if (kmax!=k)
{
for (i=0;i<n;i++) // переставляем столбцы
{
buf=A(i,k);
A(i,k)=A(i,kmax);
A(i,kmax)=buf;
}
det=-det;
L(0,m)=k; //сохраняем номера переставленных столбцов
L(1,m)=kmax;
m++;
}
}

```

```

        }
        det=det*A(k,k);
    }
// нахождение обратной матрицы методом Гаусса-Жордана
void fine()
{
    int i,j,k,m0,m1;
    float buf;
    det=1; m=0;

    for (k=0;k<n;k++)
    {
        select_Amax(k); // выбор ведущего элемента
        if (det==0) return;
        for (i=0;i<n;i++)
        {
            if (i!=k)
            {
                A(i,k)=-A(i,k)/A(k,k);
                for (j=0;j<n;j++)
                {
                    if (j!=k)
                    {
                        A(i,j)=A(i,j)+A(i,k)*A(k,j);
                    }
                }
            }
        }
        for (j=0;j<n;j++)
        {
            if (j!=k)
            {
                A(k,j)=A(k,j)/A(k,k);
            }
        }
        A(k,k)=1/A(k,k);
    }
    if (m>0)      // переставляем строки в обратной матрице
    {
        for (i=(m-1);i>=0;i--)
        {
            m0=L(0,i);
            m1=L(1,i);
            for (j=0;j<n;j++)
            {
                buf=A(m0,j);
                A(m0,j)=A(m1,j);
                A(m1,j)=buf;
            }
        }
    }
}

```

А это результаты ее работы

```
Ведите размерность исходной матрицы: 5
Ведите 0/1 - задать систему случайным образом или вручную 0
Введенная матрица:
0.02048 -0.49951 -0.11264 -0.35117 -0.11053
0.21796 -0.29504 0.45590 0.17920 -0.25458
0.34467 -0.36581 0.06110 -0.27759 -0.07358
0.20813 -0.19092 -0.21429 -0.33566 -0.33795
0.09088 0.07028 0.23901 0.19775 0.07004
Определитель матрицы системы = -0.00005
Обратная матрица:
92.62115 -54.90818 -76.79281 76.89443 236.95479
-113.47365 63.70029 93.75261 -89.06085 -278.79568
-188.33047 109.29766 158.32056 -152.69626 -470.42242
216.26018 -123.78255 -182.52496 174.37308 541.01971
25.77012 -16.16280 -19.35870 18.34210 64.36771

Число переставленных строк - 3
Произведение обратной матрицы на исходную:
1.00000 0.00000 -0.00000 0.00000 0.00000
0.00000 1.00000 0.00001 -0.00000 0.00000
0.00000 -0.00000 1.00000 -0.00001 -0.00000
-0.00001 0.00001 -0.00000 1.00002 0.00000
-0.00000 -0.00000 -0.00000 0.00000 1.00000
```

```
*** Программа находит обратную матрицу методом Гаусса-Жордано ***
Ведите размерность исходной матрицы: 2
Ведите 0/1 - задать систему случайным образом или вручную 1
Ведите A(0,0): 1
Ведите A(0,1): 2
Ведите A(1,0): 2
Ведите A(1,1): 4
Введенная матрица:
1.00000 2.00000
2.00000 4.00000
Определитель матрицы системы = -0.00000
Матрица системы ВЫРОЖДЕННАЯ !
Для завершения программы введите любой символ и ENTER
```

Далее для сравнения приведены листинг (рис. 2.13) аналогичной консольной программы на языке Python и результаты ее работы (рис. 2.14). Обратите внимание, что в языке Python для объединения команд в блоки используется одинаковое число (обычно четыре) левых пробелов. Кстате, это соответствует правилу «хорошего тона» в программировании.

```
# -*- coding: cp1251 -*-
import random
# --- процедура выбора ведущего элемента ---
```

```

def select_Amax(k):
    global m,det,n
    global A
    #print m,k
    if k<(m-1):
        Amax=abs(A[k][k])
        kmax=k
        for j in xrange(k+1,m):
            Amod=abs(A[k][j])
            if (Amod>Amax):
                Amax=Amod
                kmax=j
        if (Amax<0.0000000001): # матрицы вырожденная
            det=0
            return
        if (kmax!=k):
            for i in xrange(0,m): # переставляем столбцы
                buf=A[i][k]
                A[i][k]=A[i][kmax]
                A[i][kmax]=buf
                det=-det
            L[0][n]=k
            L[1][n]=kmax
            n+=1
        det=det*A[k][k]
    return

```

```

# --- нахождение обратной матрицы методом Гаусса-Жордана ---
def convertM():
    global det,n,m
    global A
    for k in xrange(0,m):
        select_Amax(k)      # выбор ведущего элемента
        if (det==0):
            break
        for i in xrange(0,m):
            if (i!=k):
                A[i][k]=-A[i][k]/A[k][k]
                for j in xrange(0,m):
                    if (j!=k):
                        A[i][j]=A[i][j]+A[i][k]*A[k][j]
        for j in xrange(0,m):
            if (j!=k):
                A[k][j]=A[k][j]/A[k][k]

```

```

A[k][k]=1/A[k][k]
if (n>0):           # переставляем строки
    for i in xrange((n-1),-1,-1):
        n0=L[0][i]
        n1=L[1][i]
        for j in xrange(0,m):
            buf=A[n0][j];
            A[n0][j]=A[n1][j];
            A[n1][j]=buf;
print "Определитель исходной матрицы = ", "%.5f" % det
print "Число использованных перестановок столбцов = ",n
if (det!=0):
    print "Получена следующая обратная матрица A(обр):"
    for i in xrange(0,m):
        for j in xrange(0,m):
            print "%.5f" % A[i][j],   #формат - вещественное, 5 знаков после
точки
        print
else:
    print "Матрица ВЫРОЖДЕННАЯ"

# --- Главная программа ---
print "Нахождение обратной матрицы методом Гаусса-Жордано с выбором
ведущего элемента"
fl_end=1
while fl_end:
    A=[]  #исходная матрица - !!! это надо делать при каждом повторении
    B=[]  #для хранения копии матрицы
    E=[]  #используется при тестировании точности обратной матрицы
    (буфер для единичной матрицы)
    m=raw_input("Введите размерность матрицы A: ")      #завершается
нажатием - ENTER
    m=int(m)          #переводим символьное число в число типа INT
    ans=raw_input("Компоненты матрицы задать случайным образом (1-ДА/2-
НЕТ)? ")
    if ans=="1":      #формирование случайной матрицы
        for i in xrange(0,m):
            ar1=[]
            ar2=[]
            ar3=[]
            for j in xrange(0,m):
                a=random.random()
                ar1.append(a)      #добавляем число в вектор
                ar2.append(a)

```

```

ar3.append(a)
A.append(ar1)      #добавляем вектор в матрицу
B.append(ar2)
E.append(ar3)
else:
    for i in xrange(0,m):      #ввод произвольной матрицы пользователем
        ar1=[]
        ar2=[]
        ar3=[]
        for j in xrange(0,m):
            lp=1
            while lp:
                str1="Введите компонент матрицы - A("+str(i)+","+str(j)+"):"
                a=raw_input(str1)
                try:                      #обработка исключения
                    a=float(a)          #проверяемая операция
                    lp=0
                except ValueError:      #тип исключения и реакция на него
                    print "Ошибка в задании компонента"
                ar1.append(a)
                ar2.append(a)
                ar3.append(a)
            A.append(ar1)
            B.append(ar2)
            E.append(ar3)
ans=raw_input("Показать сформированную матрицу (1-ДА/2-НЕТ)? ")
if ans=="1":
    print "Сформирована следующая матрица A:"
    for i in xrange(0,m):
        for j in xrange(0,m):
            #print A[i][j],
            print "% .5f" % A[i][j],      #формат - вещественное, 5 знаков после
точки
            print                         #перевод строки
det=1
n=0
L=[]      #задание вспомогательного массива для хранения номеров
переставляемых столбцов
for i in xrange(0,2):
    ar1=[]
    for j in xrange(0,m):
        ar1.append(-1)
    L.append(ar1)
convertM()           #находим обратную матрицу

```

```

if det!=0:                      #если матрица не вырожденная
    ans=raw_input("Протестировать точность обратной матрицы [т.е. ? - 
A(обр)*A=E] (1-ДА/2-НЕТ)? ")
    if ans=="1":
        print "Произведение матриц A(обр)*A (с точностью до 7-го знака):"
        for i in xrange(0,m):          # расчитываем произведение A(обр)*A
            for j in xrange(0,m):
                E[i][j]=0
                for k in xrange(0,m):
                    E[i][j]+=A[i][k]*B[k][j]
                print "%.7f" % E[i][j],      #формат - вещественное, 7 знаков после
точки
                print
ans=raw_input("Выйти из программы (1-ДА/2-НЕТ)? ")
if ans=="1": fl_end=0

```

Рис. 2.13. Листинг консольной программы на языке Python.

Нахождение обратной матрицы методом Гаусса-Жордано с выбором ведущего элемента

Введите размерность матрицы A: 3

Компоненты матрицы задать случайным образом (1-ДА/2-НЕТ)? 2

Введите компонент матрицы - A(0,0): 0

Введите компонент матрицы - A(0,1): 2

Введите компонент матрицы - A(0,2): 3

Введите компонент матрицы - A(1,0): 4

Введите компонент матрицы - A(1,1): 5

Введите компонент матрицы - A(1,2): 0

Введите компонент матрицы - A(2,0): 0

Введите компонент матрицы - A(2,1): 6

Введите компонент матрицы - A(2,2): 8

Показать сформированную матрицу (1-ДА/2-НЕТ)? 1

Сформирована следующая матрица A:

0.00000 2.00000 3.00000

4.00000 5.00000 0.00000

0.00000 6.00000 8.00000

Определитель исходной матрицы = 8.00000

Число использованных перестановок столбцов = 1

Получена следующая обратная матрица A(обр):

5.00000 0.25000 -1.87500

-4.00000 0.00000 1.50000

3.00000 0.00000 -1.00000

Протестировать точность обратной матрицы [т.е. ? - A(обр)*A=E] (1-ДА/2-НЕТ)? 1

Произведение матриц A(обр)*A (с точностью до 7-го знака):
 1.0000000 -0.0000000 -0.0000000
 0.0000000 1.0000000 0.0000000
 0.0000000 0.0000000 1.0000000
 Выйти из программы (1-ДА/2-НЕТ)? 1

Рис.2.14. Результаты работы консольной программы.

Наконец, аналогичная работа выполнена в пакете Maxima (рис. 2.15 и 2.16), Здесь обратная матрица вычисляется с помощью соответствующей встроенной функции.

```
numer:true;      /* выдавать числа в десятичном виде */  

fpprintprec:5;   /* в выводимом числе 5 цифр, не считая точки */  
  

M:matrix([0,2,3],[4,5,0],[0,6,8]); /* задаем исходную матрицу */  

OM:invert(M);        /* находим обратную матрицу – встроенная функция */  

mulMatr(A,B):=block /* подпрограмма перемножает матрицы A*B */  

(  

 [i,j,k,n],           /* локальные переменные подпрограммы */  

 n:length(A),          /* размерность матриц */  

 E:zeromatrix(n,n),    /* генерируем нулевую «глобальную» матрицу */  

 for i thru n do      /* три вложенных цикла */  

   for j thru n do  

     for k thru n do E[i,j]:=E[i,j]+A[i,k]*B[k,j]  

);  

mulMatr(OM,M);      /* вызов подпрограммы */  

E;                   /* выдаем результат тестирования */
```

Рис. 2.15. Листинг команд пакета Maxima, выполняющих расчет обратной матрицы и ее тестирование.

```
(%o3) 
$$\begin{bmatrix} 0 & 2 & 3 \\ 4 & 5 & 0 \\ 0 & 6 & 8 \end{bmatrix}$$
  

(%o4) 
$$\begin{bmatrix} 5.0 & 0.25 & -1.875 \\ -4.0 & 0 & 1.5 \\ 3.0 & 0 & -1.0 \end{bmatrix}$$
  

- исходная и обратная матрицы  

(%o7) 
$$\begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0 & 1.0 & 0.0 \\ 0 & 0.0 & 1.0 \end{bmatrix}$$
  

- результат тестирования E = A(обр)*A
```

Рис.2.16. Результаты выполнения команд, представленных на рис. 2.10.
Сравните их с информацией на рис. 2.6, 2.7 и 2.9.

Вопросы и задания для самоконтроля

1. В чем суть метода исключения Гаусса?
2. Для каких целей в прямых методах используется процедура «выбора ведущего элемента»?
3. Запишите алгоритм для k -го преобразования исходной матрицы в методе Гаусса.
4. Запишите алгоритм нахождения решения системы с нижней треугольной матрицей.
5. Чем метод Гаусса отличается от оптимального метода исключения Гаусса-Жордана?
6. Что такое матрица перестановок и ее свойства?
7. Напишите алгоритм, который меняет местами i -ую и k -ую строки матрицы A .
8. Каким образом выбирается ведущий элемент при нахождении обратной матрицы методом Гаусса-Жордана?
9. Почему при нахождении обратной матрицы необходимо запоминать номера переставляемых столбцов при выборе ведущего элемента?

2.2.4. Первый метод ортогонализации.

Представим систему $Ax = b$ в следующем виде

$$\sum_{i=1}^n A_{ij}x_j + A_{in+1} = 0, \quad i = 1, n \quad (2.33)$$

где $A_{in+1} = -b_i$.

Если ввести $(n+1)$ -мерные вектора $A^i = (A_{i1}, \dots, A_{in}, A_{in+1})$, $i = 1 \dots n$ и $y = (x_1, \dots, x_n, 1)$, то эти уравнения можно заменить системой скалярных произведений

$$(A^i, y) = 0, \quad i = 1, n \quad (2.34)$$

Соотношения (2.34) позволяют дать другую формулировку для исходной задачи: в $(n+1)$ -мерном пространства R^{n+1} найти $(n+1)$ -мерный вектор y с компонентом $y_{n+1} = 1$, перпендикулярный n -мерной гиперплоскости, «натянутой» на вектора A^i , $i = 1, n$. Если удастся построить такой вектор, то первые n компонент этого вектора будут являться компонентами решения исходной системы уравнений.

Предлагается следующая схема решения этой задачи:

- К исходной расширенной матрице добавляем снизу строку с компонентами $(0, 0, \dots, 1)$. Если матрица исходной системы невырожденная, то строки этой новой матрицы в пространстве R^{n+1} образуют полную линейно-независимую систему векторов. Это следует из того обстоятельства, что ее определитель равен определителю исходной матрицы, который для невырожденной системы отличен от нуля.
- Применяя к указанной полной линейно-независимой системе векторов процедуру ортогонализации Грама-Шмидта (2.35), строим ортонормированный базис для пространства R^{n+1} (см. рис. 2.17).

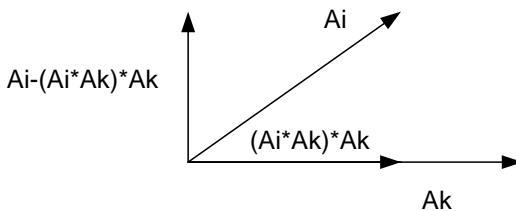


Рис. 2.17. Графическое пояснение операции процедуры Грамма-Шмидта

для $k = 1, n$

$$A^k = \frac{A^k}{\|A^k\|}, \quad \text{где } \|A^k\| = \sqrt{(A^k, A^k)}$$

для $i = k + 1, n + 1$

$$A^i = A^i - (A^i, A^k) A^k \quad (2.35)$$

конец i

конец k

$$A^{n+1} = \frac{A^{n+1}}{\|A^{n+1}\|}$$

Последний орт этой системы (последняя строка расширенной матрицы) будет перпендикулярен n -мерной гиперплоскости, «натянутой» на вектора $A^i, i = 1, n$.

Детализируем отдельные операции алгоритма процедуры ортогонализации:

- расчет скалярного произведения $sk(i, k) = (A^i, A^k) = \sum_{j=1}^{n+1} A_{ij} A_{kj}$

$s = 0$
 для $j = 1, n + 1$
 $s = s + A_{ij}A_{kj}$
 конец j
 $sk = s$

- нормировка вектора $A^k = \frac{A^k}{\|A^k\|}$

$norm = sk(k, k)$

если $norm \approx 0$, то "матрица вырожденная", иначе $norm = \sqrt{norm}$
 для $j = 1, n + 1$

$$A_{kj} = \frac{A_{kj}}{norm}$$

конец j

Нулевое значение подкоренного выражения означает линейную зависимость системы исходных векторов, т.е. вырожденность матрицы системы $Ax = b$.

- корректировка вектора $A^i = A^i - (A^i, A^k)A^k$

$sk_ik = sk(i, k)$
 для $j = 1, n + 1$
 $A_{ij} = A_{ij} - sk_ik A_{kj}$
 конец j

3. Поделив последнюю строку матрицы на ее последнюю компоненту, получим решение исходной системы

$$\begin{aligned}
 & \text{для } k = 1, n \\
 & x_k = \frac{A_{n+1k}}{A_{n+1n+1}} \quad (2.36) \\
 & \text{конец } k
 \end{aligned}$$

Замечание. Алгоритм (2.36) необходимо поставить вместо последнего оператора в схеме (2.35), которая является общей записью алгоритма ортогонализации.

На рис. 2.18 приведены экранные формы программы (основное окно и окно справки), которая с помощью рассмотренных выше алгоритмов (параграфы

2.2.2 и 2.2.4) находит решение и компоненты вектора невязки ($\delta_i = b_i - \sum_{j=1}^n A_{ij}x_j$) для заданной системы уравнений $Ax=b$.

Здесь предусмотрены возможность корректировки значения машинного нуля (переменная *nul*, используемая для сранения действительных чисел с нулем), а также включение и отключение дополнительного критерия плохо обусловленной матрицы $\|\delta\|_{куб} < nul$ (наряду с критериями $|A_{kk}| < nul$ или $norm < nul$, соответственно, для методов Гаусса-Жордана или метода ортогонализации).

Решение системы Ax=b методом Гаусса-Жордана или первым методом ортогонализации

справка выход

Размерность системы (>1) <input type="text" value="3"/>						
Матрица, правая часть системы Ax=b, ее решение и невязка для полученного решения						
i \ j	1	2	3	b	x	невязка
1		1	2	3	2,75	0
2	4	5		6	-1,	1E-15
3		7	8	9	2,	-2E-15

Строка, столбец и значение компоненты:

Определитель матрицы системы Число использованных перестановок строк

Выберите метод решения	Машинный нуль для действительных чисел <input type="text" value="1E-10"/>
<input checked="" type="radio"/> метод оптимального исключения	<input checked="" type="checkbox"/> учитывать норму невязки при оценки точности решения -да/нет
<input type="radio"/> первый метод ортогонализации	<input type="button" value="найти решение системы"/>

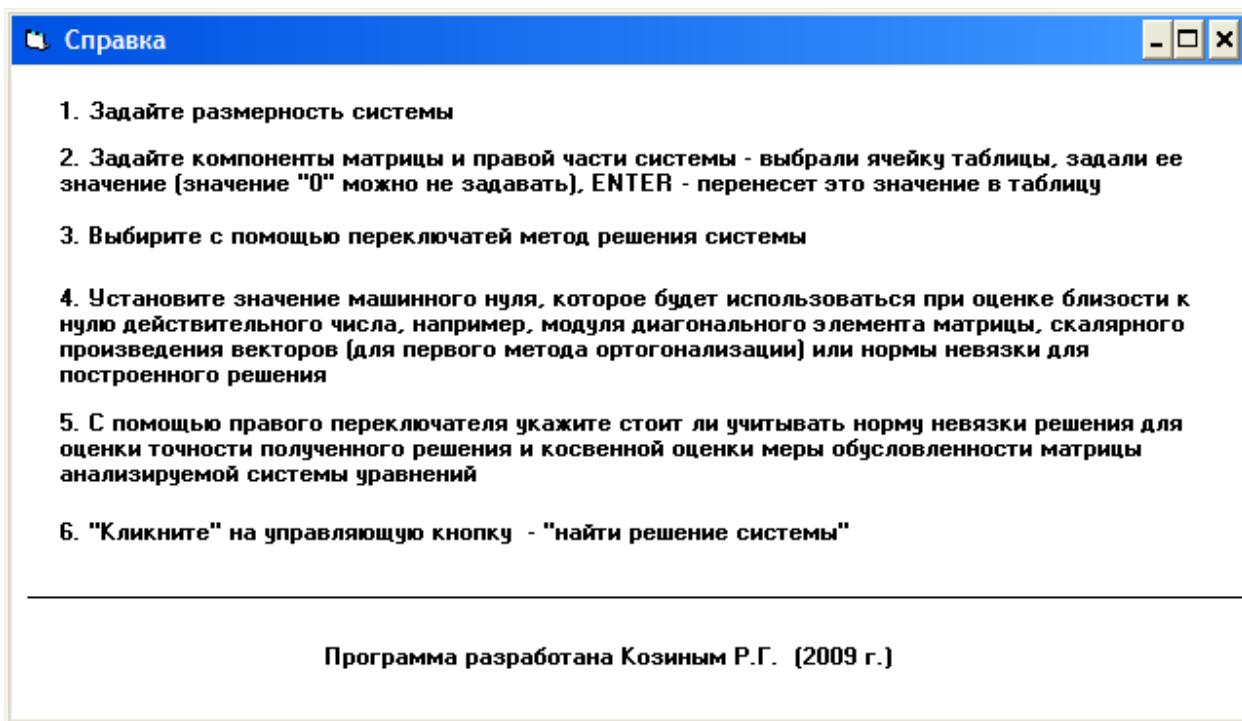


Рис. 2.18. Экранные формы программы, которая находит решения произвольной системы $Ax = b$

На рис.2.19 и 2.20 приведены результаты использования этих дополнительных возможностей для системы уравнений с вырожденной матрицей. Сравнение этих результатов наглядно показывает, к каким существенным изменениям в решении x приводит незначительная входная погрешность в векторе правой части системы b .

Решение системы $Ax=b$ методом Гаусса-Жордана или первым методом ортогонализации

справка выход

Размерность системы (>1)

Матрица, правая часть системы $Ax=b$, ее решение и невязка для полученного решения

i \ j	1	2	3	b	x	невязка
1	1	2	3	6	0	0
2	4	5	6	15	3,	0
3	7	8	9	24	0	0

Строка, столбец и значение компоненты:

Определитель матрицы системы Число использованных перестановок строк

Выберите метод решения

метод оптимального исключения

первый метод ортогонализации

Машинный нуль для действительных чисел

учитывать норму невязки при оценки точности решения -да/нет

найти решение системы

Рис. 2.19. Применение метода оптимального исключения к системе с вырожденной матрицей (ее определитель равен нулю) при $nul = 1E^{-30}$. Благодаря «пропуску» ошибок округления, удалось получить для данной системы одно из частных ее решений $x=(0,3,0)$ (например, другое - $x = (1, 1, 1)$ и т.д.).

Решение системы $Ax=b$ методом Гаусса-Жордана или первым методом ортогонализации

справка выход

Размерность системы (>1)

Матрица, правая часть системы $Ax=b$, ее решение и невязка для полученного решения

i \ j	1	2	3	b	x	невязка
1	1	2	3	6,0001	630485476	-2E-5
2	4	5	6	15	-126097095	-5E-5
3	7	8	9	24	630485476	5E-5

Строка, столбец и значение компоненты:

Определитель матрицы системы Число использованных перестановок строк

Выберите метод решения

метод оптимального исключения

первый метод ортогонализации

Машинный нуль для действительных чисел

учитывать норму невязки при оценки точности решения -да/нет

найти решение системы

Рис.2.20. Результат применения метода исключения к системе с вырожденной матрицей и отключенным вторым критерием по невязке. Небольшое изменение правой части системы b привело к полной перестройке решения системы x (сравните с рис.2.9).

В заключении для сравнения приведено решение предыдущих задач в рамках пакета Maxima (рис.2.21 и 2.22).

```

numer:true;
fpprintprec:5;
A:matrix([0,1,2],[4,5,0],[0,7,8]); /* "хорошая" матрица системы */
determinant(A); /* расчет определителя матрицы */
/* находим решение "хорошей" системы */
solve([x2+2*x3=3,4*x1+5*x2=6,7*x2+8*x3=9],[x1,x2,x3]);

B:matrix([1,2,3],[4,5,6],[7,8,9]); /* "плохая" матрица системы */
determinant(B);
/* находим решение "плохой" системы */
solve([x1+2*x2+3*x3=6,4*x1+5*x2+6*x3=15,7*x1+8*x2+9*x3=24],[x1,x2,x3])
);

```

Рис. 2.21. Листинг программы для Maxima.

```
(%o20) true
(%o21) 5
(%o22)

$$\begin{bmatrix} 0 & 1 & 2 \\ 4 & 5 & 0 \\ 0 & 7 & 8 \end{bmatrix}$$

(%o23) 24
(%o24) [[x1=2.75, x2=-1, x3=2]]
(%o25)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

(%o26) 0
solve: dependent equations eliminated: (3)
(%o27) [[x1 = %r4, x2 = 3 - 2 %r4, x3 = %r4]] - общее решение вырожденной
системы
```

Рис. 2.22. Полученные результаты. Из общего решения вырожденной системы легко получить любое частное решение, например: при %r4=0 x=(0,3,0), при %r4=1 x=(1,1,1) и т.д. Сравните с рис.2.19.

Наконец ниже представлен листинг программы на C для среды CodeBlocks, который реализует лишь алгоритм первого метода ортогонализации

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <math.h>
#include <time.h>
#define B(i,j) MM[i*(n+1)+j] //исходная система
#define A(i,j) M[i*(n+1)+j] //приводимая система

float *M,*MM;
intn;      //размерность

intmetod_Ort();
float sp(inti,int k);
intnorm(int k);
void b_Ax();

intmain()
{
inti,j,c;
char s1[2];
SetConsoleCP(1251);           //чтобы воспринимала русские буквы
```

```

SetConsoleOutputCP(1251);
printf("/** Программа находит решение системы Ax=b методом ортогонализации **");
printf("\nВведите размерность системы: ");
scanf("%d",&n);
M=malloc((n+1)*(n+1)*sizeof(float)); //расширенная матрица - (A+b)+строка(0,..,1)
MM=malloc(n*(n+1)*sizeof(float));
printf("Введите 0/1 - задать систему случайным образом или вручную ");
scanf("%d",&c);
switch (c){
    case 0:
        //инициализация датчика п.с. чисел текущим временем
        srand(time(NULL));
        for(i=0;i<n;i++)for(j=0;j<n+1;j++)B(i,j)=0.5-rand()/(RAND_MAX+1.0); //[-0.5;0.5]
        break;
    default:
        for (i=0;i<n;i++){
            for (j=0;j<n;j++){
                printf("Введите A(%d,%d): ",i,j);
                scanf("%f",&B(i,j));
            }
            printf("Введите b(%d): ",i);
            scanf("%f",&B(i,n));
        }
    }
printf("Введенная система:\n");
for (i=0;i<n;i++){
    for (j=0;j<n;j++){
        printf("%.5f ",B(i,j));
        A(i,j)=B(i,j);
    }
    printf("%.5f\n",B(i,n));
    A(i,n)=-B(i,n);
    A(n,i)=0;
}
A(n,n)=1;
//!!! тест
printf("Преобразуемая матрица\n");
for (i=0;i<n+1;i++){
    for (j=0;j<n+1;j++){
        printf("%.5f ",A(i,j));
    }
    printf("\n");
}
//!!! тест
c=metod_Ort();
if (c==1){
    printf("\nМатрица системы ВЫРОЖДЕННАЯ !");
}
else{
    printf("\nРешение системы:\n");
    for(j=0;j<n;j++){
        printf("%.5f ",A(n,j));
    }
}

```

```

    }
    b_Ax();
    printf("\nНевязка для полученного решения:\n");
    for(i=0;i<n;i++){
        printf("%.5f ",B(i,n));
    }
}
printf("\nДля завершения программы введите любой символ и ENTER");
scanf("%s",&s1);
return 0;
}
//п/п вычисляет невязку для полученного решения
void b_Ax(){
int i,j;
for(i=0;i<n;i++){
    for(j=0;j<n;j++) B(i,n)=B(i,n)-B(i,j)*A(n,j);
}
}
//п/п реализует 1-ый метод ортогонализации
Int metod_Ort(){
int i,j,k,cod;
float s;
//процедура Грамма-Шмидта
for (k=0;k<n;k++){
    cod=norm(k);
    if (cod==1) {return 1;}
    for (i=k+1;i<n+1;i++){
        s=sp(i,k);
        for (j=0;j<n+1;j++){A(i,j)=A(i,j)-s*A(k,j);}
    }
    for (j=0;j<n+1;j++) A(n,j)=A(n,j)/A(n,n); //получаем решение
}
//!!! тест
printf("\nПолученная система\n");
for (i=0;i<n+1;i++){
    for (j=0;j<n+1;j++){
        printf("%.5f ",A(i,j));
    }
    printf("\n");
}
//!!! тест
return 0;
}
//п/п вычисляет скалярное произведение строк A(i,_) и A(k,_)
float sp(int i,int k){
int j;
float s;
s=0;
for (j=0;j<n+1;j++){
    s=s+A(i,j)*A(k,j);
}
return s;
}

```

```

}

//п/п нормирует k-ый вектор, возвращает 1 - матрица вырожденная, иначе - 0
Int norm(int k){
    int j;
    float norma;
    norma=sp(k,k);
    if (norma<1.E-10){return 1;}
    norma=sqrt(norma);
    for (j=0;j<n+1;j++){
        A(k,j)=A(k,j)/norma;
    }
    return 0;
}

```

Это скриншоты с результатами ее работы

```

C:\Users\admin\Documents\Л_Л_2016\+ + +\x\юф_1_+ ё\юу\ёрышчрўш\main_1_Metod_...
*** Программа находит решение системы Ax=b методом ортогонализации ***
Введите размерность системы: 3
Введите 0/1 - задать систему случайным образом или вручную 0
Введенная система:
0.36307 -0.47955 -0.15430 -0.26596
0.20782 -0.32257 0.17426 -0.26920
0.12762 -0.38458 -0.01501 0.27435
Преобразуемая матрица
0.36307 -0.47955 -0.15430 0.26596
0.20782 -0.32257 0.17426 0.26920
0.12762 -0.38458 -0.01501 -0.27435
0.00000 0.00000 0.00000 1.00000

Полученная система
0.53746 -0.70990 -0.22841 0.39371
-0.03733 -0.11602 0.91901 0.37493
-0.00625 -0.53444 0.26032 -0.80409
-3.50245 -1.84510 -0.78319 1.00000

Решение системы:
-3.50245 -1.84510 -0.78319
Невязка для полученного решения:
-0.00000 -0.00000 0.00000
Для завершения программы введите любой символ и ENTER

```

```

C:\Users\admin\Documents\Л_Л_2016\+|+|xЕюф_1_+ЕЕюуэрышчрУш\main_1_Metod...
*** Программа находит решение системы Ax=b методом ортогонализации ***
Введите размерность системы: 2
Введите 0/1 - задать систему случайным образом или вручную 1
Введите A(0,0): 1
Введите A(0,1): 2
Введите b(0): 3
Введите A(1,0): 2
Введите A(1,1): 4
Введите b(1): 6
Введенная система:
1.00000 2.00000 3.00000
2.00000 4.00000 6.00000
Преобразуемая матрица
1.00000 2.00000 -3.00000
2.00000 4.00000 -6.00000
0.00000 0.00000 1.00000

Матрица системы ВЫРОЖДЕННАЯ !
Для завершения программы введите любой символ и ENTER

```

2.2.5. Второй метод ортогонализации.

Этот метод применим к системам с симметричной положительно определенной матрицей. Поскольку только с помощью такой матрицы можно определить следующее скалярное произведение (x, Ay) , удовлетворяющее всем свойствам, присущим скалярному произведению:

1. $(x, Ay) = (y, Ax)$;
2. $(x + z, Ay) = (x, Ay) + (z, Ay)$;
3. $(\lambda x, Ay) = \lambda(x, Ay)$;
4. $(x, Ax) > 0$, при $\|x\| > 0$

Например, проверим свойство 1, т.к. выполнение остальных свойств очевидно:

$$(x, Ay) = \sum_i x_i \sum_k A_{ik} y_k = \sum_i \sum_k A_{ik} x_i y_k$$

$$(y, Ax) = \sum_k y_k \sum_i A_{ki} x_i = \sum_i \sum_k A_{ki} x_i y_k$$

Видно, что для симметричной матрицы правые, а, следовательно, и левые части этих выражений равны.

Теперь рассмотрим, как строится решение системы $Ax = b$ данным методом.

1. Берем в пространстве R^n полную линейно-независимую систему векторов E^j , $j = 1, n$ в виде столбцов единичной матрицы (ее определитель равен 1)

$$E = \begin{vmatrix} E^1 & \dots & E^n \\ \hline 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{vmatrix}$$

2. Применяя процедуру ортогонализации Грама-Шмидта к данной системе векторов, строим в пространстве R^n ортонормированный базис в смысле скалярного произведения (x, Ay)

для k = 1, n - 1

$$E^k = \frac{E^k}{\|E^k\|}, \quad \text{где } \|E^k\| = \sqrt{(E^k, AE^k)}$$

для j = k + 1, n

$$E^j = E^j - (E^j, AE^k)E^k$$

конец j

конец k

$$E^n = \frac{E^n}{\|E^n\|}$$

При реализации алгоритма ортогонализации все матричные операции следует заменить циклами по компонентам векторов, например,

$$E^j = E^j - (E^j, AE^k)E^k \Rightarrow \text{для } i = 1, n \{ E_{ij} = E_{ij} - (E^j, AE^k)E_{ik} \}),$$

а перед вычислением корня, необходимо проверять, чтобы подкоренное выражение было положительное. Невыполнение этого условия означает, что матрица системы $Ax = b$ не является положительно определенной.

Приведем алгоритм расчета скалярного произведения $sk(j, k) \equiv (E^j, AE^k) = \sum_{i=1}^n E_{ij} \sum_{m=1}^n A_{im} E_{mk}$ с учетом хранения векторов в виде столбцов единичной матрицы

```

 $s = 0$ 
для  $i = 1, n$ 
 $buf = 0$ 
для  $m = 1, n$ 
 $buf = buf + A_{im}E_{mk}$ 
конец  $m$ 
 $s = s + E_{ij}buf$ 
конец  $i$ 
 $sk = s$ 

```

3. Ищем решение системы в виде разложения по векторам построенного базиса, который сформирован на месте единичной матрицы

$$x = \sum_{j=1}^n c_j E^j \quad (2.37)$$

или через компоненты вектора x

$$x_i = \sum_{j=1}^n E_{ij}c_j, \quad i = 1, n \quad (2.38)$$

Для определения неизвестных коэффициентов разложения подставим разложение (2.37) в исходную систему $Ax = b$, а затем умножим ее скалярно последовательно на вектора $E^i, i = 1, n$

$$\sum_{j=1}^n c_j (E^i, AE^j) = (E^i, b), \quad i = 1, n \quad (2.39)$$

Ввиду ортонормированности нового базиса в левой части уравнения (2.39) останется только одно слагаемое с $j = i$, равное c_i . Отсюда следует, что

$$c_i = (E^i, b) = \sum_{k=1}^n E_{ki}b_k \quad (2.40)$$

Очевидно, метод потребует значительного количества операций, но в методическом плане он представляет определенный интерес. По этой причине с ним полезно познакомиться.

2.2.6. Метод квадратного корня (метод Холецкого)

Метод применим только для систем с симметричной положительно определенной матрицей.

Согласно этому методу исходная матрица представляется в виде произведения двух матриц

$$A = BB^T \quad (2.41)$$

где B - нижняя треугольная матрица.

Такое разложение возможно только для симметричной положительно определенной матрицы:

$$1. \ A^T = (BB^T)^T = (B^T)^T B^T = BB^T = A$$

$$2. \ (Ax, x) = (BB^T x, x) = (B^T x, B^T x) \geq 0$$

Чтобы найти рекуррентные формулы для определения компонент матрицы B , запишем матричное соотношение (2.41) для четырехмерного случая

$$\begin{vmatrix} A_{11} \\ A_{21} A_{22} \\ A_{31} A_{32} A_{33} \\ A_{41} A_{42} A_{43} A_{44} \end{vmatrix} = \begin{vmatrix} B_{11} \\ B_{21} B_{22} \\ B_{31} B_{32} B_{33} \\ B_{41} B_{42} B_{43} B_{44} \end{vmatrix} * \begin{vmatrix} B_{11} B_{21} B_{31} B_{41} \\ B_{22} B_{32} B_{42} \\ B_{33} B_{43} \\ B_{44} \end{vmatrix} =$$

$$= \begin{vmatrix} B_{11}^2 \\ B_{11} B_{21} \ B_{21}^2 + B_{22}^2 \\ B_{11} B_{31} \ B_{31} B_{21} + B_{32} B_{22} \ B_{31}^2 + B_{32}^2 + B_{33}^2 \\ B_{11} B_{41} \ B_{41} B_{21} + B_{42} B_{22} \ B_{41} B_{31} + B_{42} B_{32} + B_{43} B_{33} \ B_{41}^2 + B_{42}^2 + B_{43}^2 + B_{44}^2 \end{vmatrix}$$

Отсюда, сравнивая компоненты левой и правой матриц, получим

$$B_{11} = \sqrt{A_{11}}$$

$$B_{21} = \frac{A_{21}}{B_{11}}, \quad B_{22} = \sqrt{A_{22} - B_{21}^2} \left(= \sqrt{\frac{A_{11}A_{22} - A_{21}^2}{A_{11}}} \right)$$

$$B_{31} = \frac{A_{31}}{B_{11}}, \quad B_{32} = \frac{A_{32} - B_{31}B_{21}}{B_{22}}, \quad B_{33} = \sqrt{A_{33} - B_{31}^2 - B_{32}^2}$$

$$B_{41} = \frac{A_{41}}{B_{11}}, \quad B_{42} = \frac{A_{42} - B_{41}B_{21}}{B_{22}}, \quad B_{43} = \frac{A_{43} - B_{41}B_{31} - B_{42}B_{32}}{B_{33}},$$

$$B_{44} = \sqrt{A_{44} - B_{41}^2 - B_{42}^2 - B_{43}^2}$$

Здесь в двух верхних строчках под корнем явно присутствуют определители первых миноров исходной матрицы, а согласно критерию Сильвестра для положительно определенной матрицы они положительны. Последнее утверждение справедливо и для остальных подкоренных выражений. Это еще раз подтверждает необходимость того, чтобы матрица была положительно определенной.

Эти формулы по индукции позволяют записать следующий алгоритм расчета всех компонент матрицы B для n -мерного случая

$$\begin{aligned}
B_{11} &= \sqrt{A_{11}} \\
B_{21} &= \frac{A_{21}}{B_{11}}, \quad B_{22} = \sqrt{A_{22} - B_{21}^2} \\
\text{для } k &= 3, n \\
&\quad \{ \\
&\quad \quad B_{k1} = \frac{A_{k1}}{B_{11}} \\
&\quad \quad \text{для } i = 2, k-1 \\
&\quad \quad \quad \{ \\
&\quad \quad \quad \quad B_{ki} = \frac{A_{ki} - \sum_{j=1}^{i-1} B_{kj} B_{ij}}{B_{ii}} \\
&\quad \quad \quad \} \\
&\quad \quad \quad B_{kk} = \sqrt{A_{kk} - \sum_{j=1}^{k-1} B_{kj}^2} \\
&\quad \}
\end{aligned} \tag{2.42}$$

При программной реализации этого алгоритма перед вычислением корня необходимо предварительно проверять является ли подкоренное выражение положительным. Если это условие не выполняется, то исходная матрица не положительно определенная. Отметим, что этот алгоритм можно использовать для проверки положительной определенности произвольной симметричной матрицы.

После того как разложение матрицы A выполнено система $Ax = b$ распадается на две системы с треугольными матрицами

$$Ax = b \Rightarrow BB^T x = b \Rightarrow By = b, \quad B^T x = y$$

или в покомпонентной записи

$$\begin{aligned}
B_{11}y_1 &= b_1 \\
B_{21}y_1 + B_{22}y_2 &= b_2 \\
\cdots & \\
B_{i1}y_1 + B_{i2}y_2 + \dots + B_{ii}y_i &= b_i \\
\cdots & \\
\cdots & \\
B_{ii}x_i + B_{i+1i}x_{i+1} + \dots + B_{ni}x_n &= y_i \\
\cdots & \\
B_{n-1n-1}x_{n-1} + B_{nn-1}x_n &= y_{n-1} \\
B_{nn}x_n &= y_n
\end{aligned}$$

Их решение рассчитывается элементарно

$$y_1 = \frac{b_1}{B_{11}}, \text{ для } i = 2, n \quad \{ y_i = \frac{b_i - \sum_{j=1}^{i-1} B_{ij} y_j}{B_{ii}} \}$$

$$x_n = \frac{y_n}{B_{nn}}, \text{ для } i = (n-1), 1 \quad \{ x_i = \frac{y_i - \sum_{j=i+1}^n B_{ji} x_j}{B_{ii}} \}$$

Последовательное вычисление компонент промежуточного вектора y можно выполнять в алгоритме (2.42) после определения компонент соответствующей строки матрицы B (см. приведенный ранее алгоритм). При этом для экономии памяти матрицу B можно хранить на месте исходной матрицы A , а вектор y формировать на месте вектора x . Необходимо помнить, что все суммы, встречающиеся в алгоритмах, вычисляются путем введения соответствующих внутренних циклов.

В заключении привем листинг на языке Visual Basic процедуры-функции, реализующей метод квадратного корня с использованием приведенных алгоритмов.

```

Function kvroot () As Integer
Dim i%, j%, k%
kvroot = 0      'матрица положительно-определенная
If Not (a(0, 0) > 0) Then
    kvroot = 1  'матрица не положительно-определенная
    Exit Function
End If
a(0, 0) = Sqr(a(0, 0))
x(0) = a(0, n) / a(0, 0)
a(1, 0) = a(1, 0) / a(0, 0)
x(1) = a(1, n)
a(1, 1) = a(1, 1) - a(1, 0) * a(1, 0)
If Not (a(1, 1) > 0) Then
    kvroot = 1  'матрица не положительно-определенная
    Exit Function
End If
a(1, 1) = Sqr(a(1, 1))
x(1) = (x(1) - a(1, 0) * x(0)) / a(1, 1)
If n = 2 Then GoTo cont
For k = 2 To n - 1
    a(k, 0) = a(k, 0) / a(0, 0)
    x(k) = a(k, n) - a(k, 0) * x(0)
    For i = 1 To k - 1

```

```

For j = 0 To i - 1
    a(k, i) = a(k, i) - a(k, j) * a(i, j)
Next j
a(k, i) = a(k, i) / a(i, i)
x(k) = x(k) - a(k, i) * x(i)
Next i
For j = 0 To k - 1
    a(k, k) = a(k, k) - a(k, j) * a(k, j)
Next j
If Not (a(k, k) > 0) Then
    kvroot = 1  'матрица не положительно-определенная
    Exit Function
End If
a(k, k) = Sqr(a(k, k))
x(k) = x(k) / a(k, k)
Next k
cont:
x(n - 1) = x(n - 1) / a(n - 1, n - 1)
For i = n - 2 To 0 Step -1
    For j = i + 1 To n - 1
        x(i) = x(i) - a(j, i) * x(j)
    Next j
    x(i) = x(i) / a(i, i)
    Next i
End Function

```

Здесь система $Ax = b$ хранится в расширенной матрице с компонентами: $a(0,0), \dots, a(n-1,n)$.

На рис. 2.23 представлен скриншот программы, которая позволяет найти решение произвольной системы линейных уравнений как 2-ым методом ортогонализации, так и методом квадратного корня. Для преобразования произвольной системы в равносильную систему с симметричной и положительно определенной матрицей следует использовать кнопку «Умножить систему на A(t)» - умножить систему на транспонированную исходную матрицу. Имеется возможность задать систему уравнений случайным образом. Последнее позволяет провести сравнительный анализ точности (по норме невязки полученного решения) используемых методов.

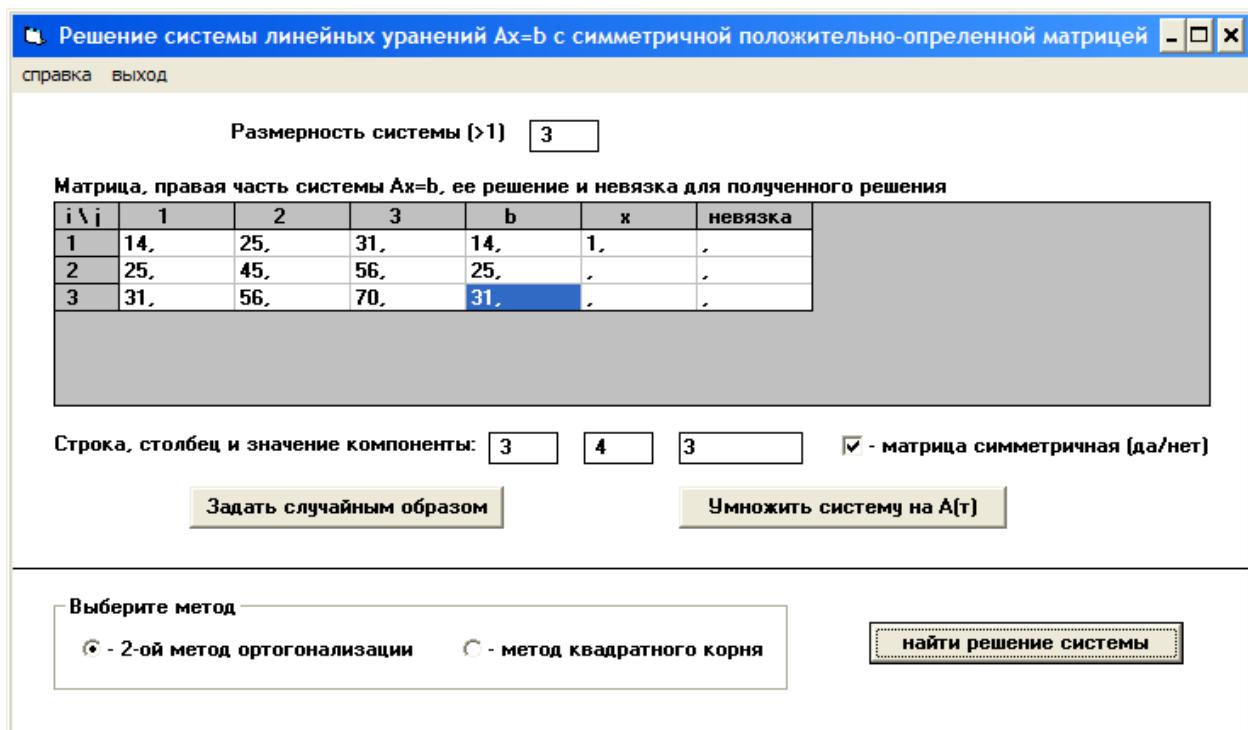


Рис. 2.23. Экранная форма программы, которая реализуют два метода, используемые для положительно определенной симметричной матрицы.

Дополнительно покажем, как этот алгоритм может быть реализован на С в среде CodeBlocks

```
#include<stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <math.h>
#include <time.h>
#define B(i,j) MM[i*(n+1)+j]
#define A(i,j) M[i*(n+1)+j] //преобразованная система – умножена на  $A^T$ 

float *MM,*M;
int n,n1;
int root_2();
void var_M();
void Ax_b();

int main()
{
int i,j,code,c;
char s[2];
SetConsoleCP(1251); //чтобы воспринимала русские буквы
```

```

SetConsoleOutputCP(1251);
printf(" *** Программа находит решение системы Ax=b методом квадратного
корня *** ");
printf("\nВведите размерность системы: ");
scanf("%d",&n); n1=n-1;
M=malloc(n*(n+1)*sizeof(float)); //добавлена правая часть
MM=malloc(n*(n+1)*sizeof(float)); //добавляем правую часть - исходная
система
printf("Введите 0/1 - задать систему случайным образом / вручную ");
scanf("%d",&c);
switch (c){
    case 0:
        //инициализация датчика п.с. чисел текущим временем
        srand(time(NULL));
        for(i=0;i<n;i++)for(j=0;j<n+1;j++)B(i,j)=0.5-rand()/(RAND_MAX+1.0);
        break;
    default:
        for (i=0;i<n;i++){
            for (j=0;j<n;j++){
                printf("Введите A(%d,%d): ",i,j);
                scanf("%f",&B(i,j));
            }
            printf("Введите b(%d): ",i);
            scanf("%f",&B(i,n));
        }
        break;
}
printf("Исходная система:\n");
for (i=0;i<n;i++){
    for (j=0;j<n;j++){
        printf("%f ",B(i,j));
    }
    printf("%f\n",B(i,n));
}
var_M(); //преобразуем исходную систему
printf("Преобразованная система Bt*B:\n");
for (i=0;i<n;i++){
    for (j=0;j<=n;j++){
        printf("%f ",A(i,j));
    }
    printf("\n");
}
code=root_2();
if (code==1) printf("\nМатрица системы ПЛОХАЯ !");

```

```

else{
    printf("\nРешение:\n");
    for(i=0;i<n;i++){
        printf("%f ",A(i,n));
    }
    Ax_b();
    printf("\nНевязка:\n");
    for(i=0;i<n;i++){
        printf("%f ",B(i,n));
    }
}
printf("\nДля завершения программы нажмите любую клавишу и ENTER ");
scanf("%s",s); printf("\n");
return 0;
}

```

//п/п реализует метод квадратного корня - находит решение

```

int root_2(){
    int i,j,k;
    //строим нижнюю треугольную матрицу
    if (A(0,0)<=0) return 1; //матрица системы плохая
    A(0,0)=sqrt(A(0,0));
    A(0,n)=A(0,n)/A(0,0); // y(0)
    A(1,0)=A(1,0)/A(0,0);
    A(1,1)=A(1,1)-A(1,0)*A(1,0);
    if (A(1,1)<=0) return 1; //матрица системы плохая
    A(1,1)=sqrt(A(1,1));
    A(1,n)=(A(1,n)-A(1,0)*A(0,n))/A(1,1); // y(1)
    if (n>2){
        for (k=2;k<n;k++){ //цикл по следующим уравнениям
            A(k,0)=A(k,0)/A(0,0);
            A(k,n)=A(k,n)-A(k,0)*A(0,n); //y(k)=b(k)-B(k,0)y(0)....
            for (i=1;i<k;i++){
                for (j=0;j<i;j++){
                    A(k,i)=A(k,i)-A(k,j)*A(i,j);
                }
                A(k,i)=A(k,i)/A(i,i);
                A(k,n)=A(k,n)-A(k,i)*A(i,n); //y(k)=(k)-B(k,i)y(i)...
            }
            for (j=0;j<k;j++){
                A(k,k)=A(k,k)-A(k,j)*A(k,j);
            }
            if (A(k,k)<=0) return 1; //матрица системы плохая
            A(k,k)=sqrt(A(k,k));
        }
    }
}
```

```

A(k,n)=A(k,n)/A(k,k); //конец вычисления y(k)
}
}
//находим x(i)
A(n1,n)=A(n1,n)/A(n1,n1); //x(n-1)
for (i=n-2;i>-1;i--){
    for (j=i+1;j<n;j++){
        A(i,n)=A(i,n)-A(j,i)*A(j,n);
    }
    A(i,n)=A(i,n)/A(i,i);
}
return 0;
}
//п/п вычисляет  $A = B^T * B$  - симметричная положительно определенная
void var_M(){
int i,j,k;
for(i=0;i<n;i++){
    for(j=0;j<=n;j++){
        A(i,j)=0.0;
        for (k=0;k<n;k++) A(i,j)=A(i,j)+B(k,i)*B(k,j);
    }
}
}
//п/п вычисляет невязку  $d = b - Ax$ 
void Ax_b(){
int i,j;
for(i=0;i<n;i++){
    for(j=0;j<n;j++) B(i,n)=B(i,n)-B(i,j)*A(j,n);
}
}

```

Скриншот с результатами работы этой программы представлен ниже

```

"C:\Users\admin\Documents\Л_Л_2016\Линейная алгебра\root_2_Metod_n...
*** Программа находит решение системы Ax=b методом квадратного корня ***
Введите размерность системы: 4
Введите 0/1 - задать систему случайным образом / вручную 0
Исходная система:
-0.097565 -0.025970 -0.293884 -0.318604 -0.295776
-0.100281 -0.308380 0.169586 -0.291656 0.072906
-0.160004 0.046722 0.131470 -0.424469 0.209686
-0.274933 0.474579 0.191101 0.090118 -0.084198
Преобразованная система Bt*B:
0.120764 -0.104495 -0.061909 0.103472 0.011144
-0.104495 0.323181 0.052170 0.121151 -0.044963
-0.061909 0.052170 0.168931 0.005589 0.110765
0.103472 0.121151 0.005589 0.374867 -0.023621

Решение:
1.797951 0.533446 1.174628 -0.749202
Невязка:
0.000000 -0.000000 -0.000000 0.000000
Для завершения программы нажмите любую клавишу и ENTER

```

2.2.7. Метод прогонки

Метод прогонки является самым экономичным способом решения систем с трех диагональной матрицей

$$b_1x_1 + c_1x_2 = d_1$$

.....

$$a_kx_{k-1} + b_kx_k + c_kx_{k+1} = d_k, \quad k = 2, n-1 \quad (2.43)$$

.....

$$a_nx_{n-1} + b_nx_n = d_n$$

Здесь для хранения системы использовано четыре вектора a, b, c, d , причем компоненты $a_1 = c_n = 0$ не используются.

К таким системам сводится разностное решение краевых задач для дифференциальных уравнений второго порядка.

Введем рекуррентные зависимости

$$x_k = \alpha_k x_{k+1} + \beta_k, \quad k = 1, n-1 \quad (2.44)$$

Найдем формулы для расчета входящих в (2.44) неизвестных параметров α_k, β_k .

Сравнивая первое уравнение в (2.43) с соотношением (2.44), сразу получаем

$$\alpha_1 = -\frac{c_1}{b_1}, \quad \beta_1 = \frac{d_1}{b_1} \quad (2.45)$$

Далее, заменим в k -ом уравнении системы (2.43) неизвестное x_{k-1} согласно формуле (2.44)

$$a_k(\alpha_{k-1}x_k + \beta_{k-1}) + b_kx_k + c_kx_{k+1} = d_k$$

Отсюда легко выводим рекуррентные формулы для последовательного определения остальных неизвестных параметров α_k, β_k

$$\alpha_k = -\frac{c_k}{t_k}, \quad \beta_k = \frac{d_k - a_k \beta_{k-1}}{t_k}, \quad k = 2, n-1 \quad (2.46)$$

$$t_k = b_k + a_k \alpha_{k-1}$$

Вычисления по формулам (2.45), (2.46) называется прямым ходом прогонки.

Теперь, подставляя в последнее уравнение (2.43) соотношение (2.44) с $k = n-1$, получим

$$a_n(\alpha_{n-1}x_n + \beta_{n-1}) + b_n x_n = d_n \Rightarrow x_n = \frac{d_n - a_n \beta_{n-1}}{b_n + a_n \alpha_{n-1}} \quad (2.47)$$

Формулы (2.47) и (2.44) (с $k = n-1, n-2, \dots, 1$) позволяют рассчитать все компоненты x_k неизвестного решения системы (2.43). Эти расчеты называются обратным ходом прогонки.

Покажем методом индукции, что для устойчивости метода прогонки ($|b_k + a_k \alpha_{k-1}| > 0$) достаточно выполнения следующих соотношений между параметрами системы

$$|b_k| \geq |a_k| + |c_k| > 0 \quad \text{при } \forall k \quad (2.48)$$

При соблюдении условия (2.48) имеем (см. (2.45)) $|b_1| > 0$ и $|\alpha_1| < 1$. Предположим, что $|\alpha_{k-1}| < 1$. Тогда $|b_k + a_k \alpha_{k-1}| > 0$. Кроме того, $|b_k + a_k \alpha_{k-1}| > |c_k|$ и, следовательно, $|\alpha_k| < 1$ и т.д.

Заметим, что метод прогонки легко обобщается на системы с пятью (и более) диагональными матрицами. В этом случае исходное рекуррентное соотношение следует брать в виде

$$x_k = \alpha_k x_{k+1} + \beta_k x_{k+2} + \gamma_k$$

Замечаний. При программной реализации метода, в целях экономии памяти, параметры α_k, β_k можно сохранять на месте компонент векторов b_k, d_k , а решение вернуть в векторе c . В этом случае алгоритм метода будет иметь следующий вид

прямой ход прогонки

$$d_1 = \frac{d_1}{b_1}, b_1 = -\frac{c_1}{b_1}$$

для $k = 2, (n-1)$

{

$$t = b_k + a_k b_{k-1}$$

$$d_k = \frac{d_k - a_k d_{k-1}}{t}$$

$$b_k = -\frac{c_k}{t}$$

}

обратный ход прогонки

$$c_n = \frac{d_n - a_n d_{n-1}}{b_n + a_n b_{n-1}}$$

для $k = (n-1), 1$

{

$$c_k = b_k c_{k+1} + d_k$$

}

В заключении рассмотрим пример, в котором метод прогонки используется для нахождения решения краевой задачи для дифференциального уравнения второго порядка.

Пример.

Распределение стационарной температуры в тепловыделяющем стержне с теплоизолированными боковыми стенками, один торец которого теплоизолирован, а через другой происходит теплообмен, в безрамерном виде задается уравнением и граничными условиями

$$\frac{d^2 y}{dx^2} = -q(x), \quad x \in [0,1]$$

$$\frac{dy}{dx} = 0 \text{ при } x = 0, \quad \frac{dy}{dx} - \alpha y \text{ при } x = 1$$

Разностный аналог этой задачи, аппроксимирующий ее с квадратичной точностью относительно шага разностной сетки $h = \frac{1}{n}$ (рис.2.24), записывается в виде

$$y_{i-1} - 2y_i + y_{i+1} = -h^2 q_i, \quad i = 0, n$$

$$\frac{y_1 - y_{-1}}{2h} = 0, \quad \frac{y_{n+1} - y_{n-1}}{2h} = -\alpha y_n$$

Отсюда для определения неизвестных значений y_i для $i = 0, n$ получаем следующую систему линейных уравнений с трех диагональной матрицей

$$\begin{aligned} -y_0 + y_1 &= -\frac{q_0 h^2}{2} \\ y_{i-1} - 2y_i + y_{i+1} &= -h^2 q_i, \quad i = 1, (n-1) \quad (2.49) \\ y_{n-1} - (1 + \alpha h)y_n &= -\frac{q_n h^2}{2} \end{aligned}$$

Листинг программы в языке пакета Maxima, которая решает эту систему методом прогонки для $q(x) = x$ и $\alpha = 1$, приведен на рис.2.25, а результаты ее выполнения – на рис.2.26.

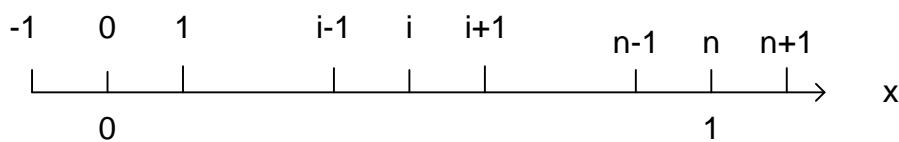


Рис. 2.24. Разностная сетка, используемая в разностной задаче.

```

kill(all);
/* подпрограмма, реализующая метод прогонки; a,b,c,d – списки с
коэффициентами */
/* системы, n – порядок системы; решение возвращается в списке c */
progonka(a,b,c,d,n):=block
( [k,t],
/* прямой ход прогонки */
d[1]:d[1]/b[1], b[1]:-c[1]/b[1],
for k:2 thru n-1 do (t:b[k]+a[k]*b[k-1], d[k]:=(d[k]-a[k]*d[k-1])/t, b[k]:-c[k]/t),
/* обратный ход прогонки */
c[n]:=(d[n]-a[n]*d[n-1])/(b[n]+a[n]*b[n-1]),
for k:(n-1) thru 1 step -1 do c[k]:=b[k]*c[k+1]+d[k]
);
/* главная программа */
numer:true;
fpprintprec:6;
define(q(x),x); /* правая часть исходного уравнения */
/* формирование коэффициентов системы */
n:11; h:1/(n-1); alf:1;
a:[]; b:[]; c:[]; d:[]; t:[]; /* инициализация списков */
t:append(t,[0]); a:append(a,[0]); b:append(b,[-1]); c:append(c,[1]); d:append(d,[-h^2*q(0)/2]);
for i:2 thru n-1 do
(
  t:append(t,[h*(i -1)]), a:append(a,[1]), b:append(b,[-2]), c:append(c,[1]),
  d:append(d,[-h^2*q(t[i])])
)

```

```

);
t:append(t,[1]); a:append(a,[1]); b:append(b,[-(1+h*alf)]); c:append(c,[0]);
d:append(d,[-h^2*q(1)/2]);
progonka(a,b,c,d,n);
c; /* выводим список значений сеточного решения, полученного методом
прогонки */
r(x):=(4-x^3)/6; /* точное решение исходной задачи */
c1:[]; for i thru n do c1:append(c1,[r(t[i])]);
c1; /* выводим список значений точного решения */
/* строим график точного решений и значения сеточного уравнения */
plot2d([[discrete, t,c], r(x)], [x,0,1],
[style, points, lines], [color, red, blue], [point_type, asterisk],
[legend, "разностное решение", "точное"],
[xlabel, "длина - x"], [ylabel, "температура"]);

```

Рис. 2.25. Листинг программы, решающей разностную задачу методом прогонки.

Список значений решения разностного уравнения:

(%o25) [0.665, 0.665, 0.664, 0.661, 0.655, 0.645, 0.63, 0.609, 0.581, 0.545, 0.5]

$$(\%o31) r(x) := \frac{4 - x^3}{6}$$

Список значений точного решения с узлах сетки:

(%o35) [0.6667, 0.6665, 0.6653, 0.6622, 0.656, 0.6458, 0.6307, 0.6095, 0.5813, 0.5452, 0.5]

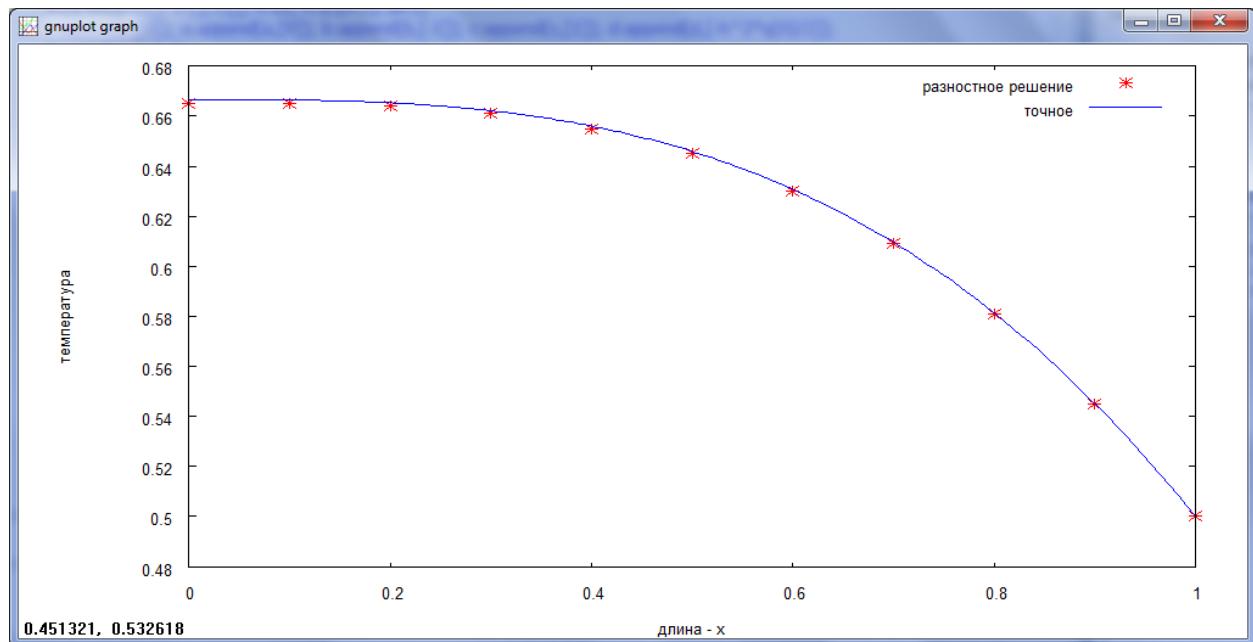


Рис. 2.26. Результаты выполнения программы. Видно, что значения разностной задачи хорошо согласуются с точным решением (графически и численно).

2.3. Метод регуляризации для решения плохо обусловленных систем.

Решение системы уравнений $Ax = b$ с плохо обусловленной матрицей существенно зависит от ошибок в исходных данных и ошибок округления (см. рис. 2.27, где для таких систем схематично показаны соответствующие области входных и выходных ошибок).



Рис. 2.27. К вопросу о соотношении ошибок в задании исходной системы уравнений и областью, в которой располагаются ее решения.

Согласно методу регуляризации для уменьшения этой зависимости, т.е. сужения области поиска, на решение системы накладывается дополнительное вполне разумное требование, чтобы его норма наименее уклонялось от нуля – «принцип минимальных затрат». В такой постановке исходную систему можно свести к задаче минимизации следующего квадратичного функционала

$$\min_x F(x) = (Ax - b, Ax - b) + \alpha(x, x) \geq 0 \quad (2.50)$$

где $\alpha > 0$ – весовой коэффициент, называемый параметром регуляризации. Его значение должен быть таким, чтобы с одной стороны обеспечить достаточную обусловленность системы, а с другой – хорошую аппроксимацию исходной задачи.

Для квадратичного функционала (2.50) решение, обеспечивающее минимум функционалу, находится аналитически из условия $\frac{dF}{dx} = 0$. Для этого распишем (2.50)

$$F(x) = (x, A^T Ax) - 2(x, A^T b) + (b, b) + \alpha(x, x) \quad (2.51)$$

Отсюда, беря производную $\frac{dF}{dx}$ с учетом очевидного преобразования

$$(x, A^T Ax) = (A^T Ax, x) = (x, (A^T A)^T x) = (x, A^T Ax)$$

и приравнивая ее нулю, получим новую систему уравнений

$$(A^T A + \alpha E)x = A^T b \quad (2.52)$$

Ее решение x_α , называемое *нормальным*, зависит от параметра α и может быть получено любым рассмотренным ранее методом.

В заключении укажем способ выбора параметра α . Обычно на практике проводится ряд вычислений с различными значениями α . После этого оптимальным считают то его значение, при котором выполняется условие приближенного равенства невязки решения сумме погрешностей исходных данных

$$\|\delta(x_\alpha)\| \equiv \|Ax_\alpha - b\| \approx \|\delta b\| + \|\delta A x_\alpha\| \quad (2.53)$$

либо то из них, для которого невязка минимальна (см. рис. 2.18).

Соответствующее этому α решение x_α принимается за нормальное решение исходной системы.

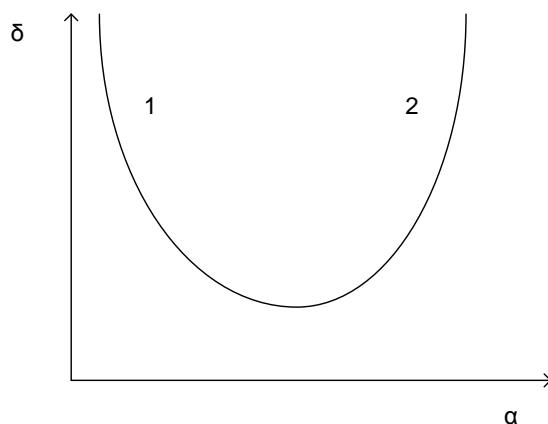


Рис. 2.28. Зависимость навязки δ от параметра α , где в зоне 1 преобладает ошибка, связанная с особенностью матрицы системы, а в зоне 2 — ошибки аппроксимации, связанные с изменением исходной системы

Наже приведены результаты работы программы, позволяющей использовать метод регуляризации для нахождения решения систем уравнений с плохо обусловленными матрицами.

Решение системы уравнений методом Гаусса-Жордана и нахождение определителя матрицы

справка выход

Размерность системы (>1) 3

Матрица, правая часть системы $Ax=b$, нормализованное решение и невязка для

i \ j	1	2	3	b	x	невязка
1	1	2	3	6		
2	1,01	2,01	3,01	6,03		
3	4	5	6	15		

Строка, столбец и значение компоненты: 3 4 15

Нормализованная система для плохо обусловленной системы уравнений - $[A(t)^*A + \alpha I]x = A(t)^*b$

i \ j	1	2	3	b	x	невязка
1	18,02010001	24,0301	30,0401	72,0903	1,00000012	-1E-16
2	24,0301	33,04010001	42,0501	99,1203	,99999975	5E-15
3	30,0401	42,0501	54,06010001	126,1503	1,00000013	5E-15

Коэффициент нормализации - α ,00000001

нормализовать систему

Определитель обрабатываемой матрицы ,00000108 Число использованных перестановок строк 2

Выбор системы

исходной системы нормализованной системы

найти решение системы

Рис. 2.29. Получено нормальное решение системы с плохо обусловленной матрицей. Заметим, что точное решение исходной системы равно (1; 1; 1). Видно, что решить непосредственно исходную систему методом Гаусса-Жордана не удалось.

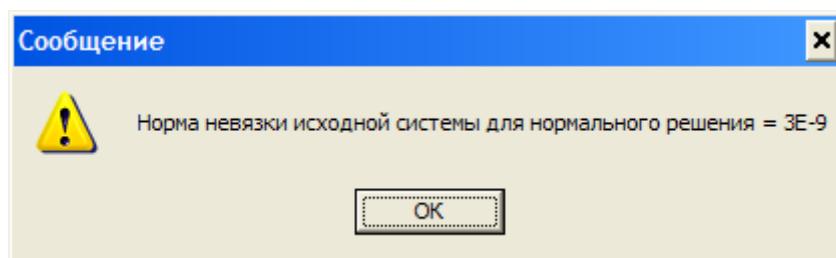


Рис. 2.30. Норма невязки исходной системы для нормального решения.

2.4. Решение систем с прямоугольными матрицами

Системы уравнений $Ax = b$, в которых матрица $A^{m \times n}$ имеют прямоугольную форму, делятся на недоопределенные ($m < n$) и переопределенные ($m > n$). Решение таких систем имеет свою специфику. При дальнейших рассуждениях предполагается, столбцы и строки матрицы A линейно независимы, поскольку только в этом случае матрицы AA^T и A^TA , которые используются при получении псевдорешений указанных систем, не будут вырожденными. В подтверждение этого факта ниже приведен результат выполнения программы на Maxima.

```
kill(all);
numer:true;
A:matrix([1,2,3],[4,5,6],[2,3,4],[2,4,6]); /* 1 и 4 строки матрицы линейно
 зависимы */
At:transpose(A); At_A:At.A;
print("Определитель матрицы At*A = ",determinant(At_A));

A:matrix([1,7,2],[4,5,8],[2,4,4],[2,6,4]); /* 1 и 3 столбцы матрицы линейно
 зависимы */
At:transpose(A); At_A:At.A;
print("Определитель матрицы At*A = ",determinant(At_A));

B:matrix([1,2,3,4],[2,3,4,6],[2,4,6,8]); /* 2 и 4 столбцы матрицы линейно
 зависимы */
Bt:transpose(B); B_Bt:B.Bt;
print("Определитель матрицы B*Bt = ",determinant(B_Bt));

B:matrix([1,2,3,7],[2,3,4,7],[2,4,6,14]); /* 1 и 3 строки матрицы линейно
 зависимы */
Bt:transpose(B); B_Bt:B.Bt;
print("Определитель матрицы B*Bt = ",determinant(B_Bt));
```

$$(\%o2) \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 2 & 3 & 4 \\ 2 & 4 & 6 \end{bmatrix}$$

$$(\%o3) \begin{bmatrix} 1 & 4 & 2 & 2 \\ 2 & 5 & 3 & 4 \\ 3 & 6 & 4 & 6 \end{bmatrix}$$

$$(\%o4) \begin{bmatrix} 25 & 36 & 47 \\ 36 & 54 & 72 \\ 47 & 72 & 97 \end{bmatrix}$$

Определитель матрицы $A t^* A = 0$

$$(\%o6) \begin{bmatrix} 1 & 7 & 2 \\ 4 & 5 & 8 \\ 2 & 4 & 4 \\ 2 & 6 & 4 \end{bmatrix}$$

$$(\%o7) \begin{bmatrix} 1 & 4 & 2 & 2 \\ 7 & 5 & 4 & 6 \\ 2 & 8 & 4 & 4 \end{bmatrix}$$

$$(\%o8) \begin{bmatrix} 25 & 47 & 50 \\ 47 & 126 & 94 \\ 50 & 94 & 100 \end{bmatrix}$$

Определитель матрицы $A t^* A = 0$

$$(\%o11) \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 6 \\ 2 & 4 & 6 & 8 \end{bmatrix}$$

$$(\%o12) \begin{bmatrix} 1 & 2 & 2 \\ 2 & 3 & 4 \\ 3 & 4 & 6 \\ 4 & 6 & 8 \end{bmatrix}$$

$$(\%o13) \begin{bmatrix} 30 & 44 & 60 \\ 44 & 65 & 88 \\ 60 & 88 & 120 \end{bmatrix}$$

Определитель матрицы $B t^* Bt = 0$

$$(\%o16) \begin{bmatrix} 1 & 2 & 3 & 7 \\ 2 & 3 & 4 & 7 \\ 2 & 4 & 6 & 14 \end{bmatrix}$$

$$(\%o17) \begin{bmatrix} 1 & 2 & 2 \\ 2 & 3 & 4 \\ 3 & 4 & 6 \\ 7 & 7 & 14 \end{bmatrix}$$

$$(\%o18) \begin{bmatrix} 63 & 69 & 126 \\ 69 & 78 & 138 \\ 126 & 138 & 252 \end{bmatrix}$$

Определитель матрицы $B^*Bt = 0$

2.4.1. Решение недоопределенных систем

Недоопределенные системы имеют множество решений. Например, их можно найти, фиксируя любые $n - m$ компонент неизвестного вектора x и определяя остальные его компоненты, как решение приведенной системы с квадратной матрицей размерности ($m * m$).

Итак, решений много, но какое из них следует считать оптиальным.

Эта задача решается в соответствии с принципом «минимальной достаточности», который всегда действует в природе. Следуя этому принципу, предпочтение необходимо отдать решению с минимальной нормой. Такому требованию удовлеворяет **нормальное псевдорешение**

$$x = A^T (AA^T)^{-1} b \text{ или } (AA^T)y = b, \quad x = A^T y$$

Легко проверит, что оно является решением исходного уравнения

$$Ax \equiv AA^T (AA^T)^{-1} b = A(A^T (A^T)^{-1}) A^{-1} b = b$$

Для доказательства оптимальности этого решения (в указанном выше смысле) приведем результаты вычислительного эксперимента, в котором расчитывались нормы нормального псевдорешения и совокупности частных решений. Последние строились путем обнуления всевозможных комбинаций «лишних» компонент вектора x . Напомним, что число таких комбинаций равно числу сочетаний для n элементов по $(n - m)$

$$C_n^{n-m} = \frac{n!}{m!(n-m)!}$$

В эксперименте использовалась недоопределенная система со случайной матрицей рамерности: $m = 3, n = 5$, для которой можно построить 10 частных решений. Листинг тестирующей программы на Maxima и результаты ее работы приведены на рис. 2.31 и 2.32.

```
kill(all);
```

```

numer:true;
fpprintprec:6;
ratprint:false;
/* недопределенная система */
b:matrix([1],[2],[3]);
g[i,j]:=random(1.0); A:genmatrix(g,3,5); /* генерация случайной матрицы A(3,5)
*/
rank(A);
At:transpose(A);
xo:At.invert(A.At).b; /* находим нормальное псевдорешение и его норму */
L_norm:[sqrt(sum(xo[i,1]*xo[i,1],i,1,5))];
for i thru 4 do /* определение всех возможных частных решений */
  for j:i+1 thru 5 do
  (
    B:submatrix(A,i,j), /* удаляем i и j столбцы из матрицы A */
    x1:zeromatrix(5,1),x1[i,1]:1, x1[j,1]:1, /* помечаем какие столбцы удалили */
    x:invert(B).b,
    norm:sqrt(sum(x[i,1]*x[i,1],i,1,3)),
    L_norm:append(L_norm,[norm]),
    k1:1,
    for k thru 5 do if x1[k,1]=0 then (x1[k,1]:x[k1,1],k1:k1+1) else x1[k,1]:0,
    xo:addcol(xo,x1)
  );
  print("Список норм нормального псевдорешения и 10 частных решений:");
  L_norm;
  print("Минимальная норма в списке = ",min_norm:lmin(L_norm));
  print("Матрица, содержащая нормальное псевдорешение и 10 частные
решения:");xo;

```

Рис.2.31. Программа для выполнения вычислительного эксперимента по проверке минимальности нормы нормального псевдорешения.

```

(%o20) 
$$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

(%o22) 
$$\begin{bmatrix} 0.9236 & 0.08232 & 0.1215 & 0.9031 & 0.6471 \\ 0.8513 & 0.5443 & 0.04307 & 0.6641 & 0.9104 \\ 0.9646 & 0.09911 & 0.1984 & 0.2868 & 0.8019 \end{bmatrix}$$

(%o28) Список норм нормального псевдорешения и 10 частных решений:
(%o29) [3.86757, 5.72693, 5.39021, 40.2689, 15.5301, 3.95214, 17.7361, 9.39905, 20.9348,
5.01752, 27.0913]
Минимальная норма в списке = 3.86757

```

(%o31) Матрица, содержащая нормальное псевдорешение и 10 частных решений:

(%o32)

1.93388	0	0	0	0	2.17761	-5.70363	5.56882	-9.64374	3.86135	-2.91516
0.1078	0	-1.60315	12.5981	4.0428	0	0	0	-8.70283	1.23957	6.15911
0.7602	4.24237	0	37.5803	14.9407	0	15.3541	-6.6067	0	0	26.2204
-2.49013	-1.87157	-2.10974	0	-1.27097	-2.58613	0	-3.69891	0	-2.95447	0
2.10427	3.36113	4.69397	-7.11275	0	2.04681	6.80363	0	16.4172	0	0

Рис.2.32. Основные результаты численного эксперимента. Видно, что по сравнению со всеми частными решениями нормальное псевдорешение действительно имеет минимальную норму.

2.4.2. Решение переопределенных систем

В этом случае в качестве решения переопределенной системы используется **обобщенное псевдорешение**, которое строится таким образом, чтобы обеспечить минимальное значение нормы невязки для данной системы уравнений, т.е. минимизирует следующий функционал

$$F(x) = (Ax - b, Ax - b)$$

Согласно формуле (2.5) обобщенным псевдорешением является решение уранения

$$(A^T A)x = A^T b \text{ или } x = (A^T A)^{-1} A^T b \quad (2.54)$$

Чтобы показать оптимальность обобщенного псевдорешения (в указанном выше смысле) был поставлен вычислительный эксперимент. В нем норма невязки обобщенного псевдорешения сравнивается с соответствующими нормами частных решений всевозможных систем, которые получаются из исходной системы путем отбрасывания «лишних» уравнений.

В эксперименте использовалась переопределенная система со случайной матрицей размерности: $m = 5, n = 3$. Она допускает построение 10 частных решений. Листинг тестирующей программы на Maxima и результаты ее работы представлены на рис. 2.33 и 2.34.

```
kill(all);
/* подпрограмма формирует случайную матрицу размерности m*n */
rndA(m,n):=block
(
A:zeromatrix(m,n),
for i thru m do
  for j thru n do A[i,j]:random(1.0)
);

numer:true;
fpprintprec:6;
ratprint:false;
```

```

/* переопределенная система */
b:matrix([1],[2],[3],[4],[5]);
rndA(5,3); A;
rank(A); At:transpose(A);
xo:invert(At.A).At.b;
nvo:b-A.xo;
print("Норма невязки для обобщенного псевдорешения =
",sqrt(sum(nvo[i,1]*nvo[i,1],i,1,5)));
L_norm:[]; /* задаем нулевой список для сбора норм невязки частных
решений */
for i thru 4 do /* определение норм невязки для всех возможных частных
решений */
    for j:i+1 thru 5 do
    (
        B:submatrix(i,j,A), bt:submatrix(i,j,b),
        x:invert(B).bt, nv:b-A.x,
        norm:sqrt(sum(nv[i,1]*nv[i,1],i,1,5)),
        L_norm:append(L_norm,[norm]),
        xo:addcol(xo,x)
    );
print("Список норм невязки для частных решений:"); L_norm;
print("Минимальная норма невязки для частных решений =
",min_norm:lmin(L_norm));
print("Матрица, содержащая обобщенное псевдорешение и частные решения:");
xo;

```

Рис.2.33. Листинг программы, сравнивающей нормы невязки для обобщенного псевдорешения и совокупности частных решений.

```

(%o5) 
$$\begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{bmatrix}$$

(%o7) 
$$\begin{bmatrix} 0.9138 & 0.1952 & 0.4824 \\ 0.3554 & 0.2522 & 0.4156 \\ 0.6126 & 0.8418 & 0.5056 \\ 0.6785 & 0.03639 & 0.6253 \\ 0.9937 & 0.04159 & 0.6326 \end{bmatrix}$$


```

(%08) 3

$$(\%09) \begin{bmatrix} 0.9138 & 0.3554 & 0.6126 & 0.6785 & 0.9937 \\ 0.1952 & 0.2522 & 0.8418 & 0.03639 & 0.04159 \\ 0.4824 & 0.4156 & 0.5056 & 0.6253 & 0.6326 \end{bmatrix}$$

Норма невязки для обобщенного псевдорешения = 2.10856

(%015) Список норм невязки для частных решений:

(%016) [3.2197, 3.1584, 3.41101, 4.06083, 16.7474, 6.68568, 2.87704, 8.48847, 4.06901, 4.10387]

Минимальная норма невязки для частных решений = 2.87704

(%018) Матрица, содержащая обобщенное псевдорешение и частные решения:

(%019)

$$\begin{bmatrix} -1.96301 & 3.1097 & 3.12592 & 4.08161 & 5.22466 & 3.4191 & -13.5449 & -5.40146 & -5.04903 & -3.48305 & -2.28093 \\ -0.9393 & -0.5333 & -1.57837 & -0.315 & -0.6999 & -20.4697 & -4.27399 & 0.1372 & -12.3851 & -4.34466 & 1.82752 \\ 8.92249 & 3.0539 & 3.09713 & 1.51286 & 0.7688 & 3.87852 & 29.461 & 12.2497 & 16.6488 & 10.429 & 5.65442 \end{bmatrix}$$

Рис.2.34. Некоторые результаты работы программы тестирования. Видно, что по сравнению со всей совокупностью частных решений обобщенное псевдорешение действительно дает минимальную норму невязки для данной системы уравнений.

Рассмотрим практический пример, который приводит к переопределенной системе уранений.

Требуется построить полином n -ой степени ($p_n(x) = \sum_{i=0}^n c_i x^i$, $c_i = ?$) ,

оптимально аппроксимирующий $m (> n)$ точек, заданных на плоскости $(x, y = f(x))$. Эту задачу математически можно сформулировать следующим образом: найти такие c_i , при которых сферическая норма невязки уранений

$$p_n(x_j) = y_j, j = 1, m$$

будет минимальна.

На рис. 2.35 приведена программа на языке пакета Maxima, которая, используя формулу (2.54), решает поставленную задачу при следующих исходных данных: $n = 4$ и $9, m = 11$, $x_j = (-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5)$, $f(x) = 1/(1 + \exp(-x))$ - сигмоидная функция. Результаты ее работы представлены на рис. 2.36 и 2.37. Они подтверждают успешное решение поставленной задачи.

```
kill(all);
numer:true;
fpprintprec:5;
/* полином p(x):=c[1,1]+c[2,1]*x+c[3,1]*x^2+c[4,1]*x^3+c[5,1]*x^4+.... */
fp(x):=block /* возвращает значение полинома в точке x */
(
    [i,pp,xx],
```

```

pp:c[1,1],xx:x,
for i:2 thru n do (pp:pp+c[i,1]*xx,xx:xx*x),
return (pp)
);
n:5; /* число слагаемых в аппроксимирующем полиноме */
fs(x):=1/(1+exp(-x)); /* исходная сигмоидная функция */
b:zeromatrix(11,1); lx:[]; x:-5;
/* формируем список 11 точек оси x и столбец со значениями функции в них*/
for i thru 11 do (lx:append(lx,[x]), b[i,1]:fs(x), x:x+1);
lx;
print("Значения сигмоидной функции в заданных точках =",b);
c:zeromatrix(n,1); A:zeromatrix(11,n);
/* формируем прямоугольную матрицу (11,n) системы для 11 точек */
for i thru 11 do
(
  x0:lx[i], x:x0, A[i,1]:1,
  for j:2 thru n do
  (
    A[i,j]:x, x:x*x0
  )
);
print("Матрица A =",A); At:transpose(A); determinant(At.A);
/* вычисляем коэффициенты аппроксимирующего полинома */
print("Коэффициенты аппроксимирующего полинома =",c:invert(At.A).(At.b));
lf:[]; lp:[]; x:-5;
for i thru 11 do (lf:append(lf,[fs(lx[i])]), z:fp(x),x:x+1,lp:append(lp,[z]));
print("Список значения f(xj) =",lf);
print("Список значений p(xj) =",lp);
lp;
normd:0; otnd:0;
for i thru 11 do
(
  d:abs(lp[i]-lf[i]), ot:d/abs(lf[i]),
  if d>normd then (normd:d),
  if ot>otnd then (otnd:ot)
);
print("Максимальные абсолютная и относительная погрешности =",normd," ,
",otnd*100,"%");
plot2d([[discrete,lx,lp], fs(s)], [s,-5,5], [style,points,lines], [color,red,blue],
[gnuplot_preamble,"set grid;"],
[legend,"аппроксимация","сигмоидная функция"], [ylabel,"f(x), p(x)"],
[xlabel,"x"]);

```

Рис. 2.35. Программа на языке пакета Maxime, которая решает задачу аппроксимации сигмоидной функции по ее значениям в точках $x_j=(-5,-4,-3,-2,-1,0,1,2,3,4,5)$ с использованием аппроксимирующего полинома n -го порядка ($n \leq 11$).

$$\text{Значения сигмоидной функции в заданных точках} = \begin{bmatrix} 0.00669 \\ 0.018 \\ 0.0474 \\ 0.119 \\ 0.269 \\ 0.5 \\ 0.731 \\ 0.881 \\ 0.953 \\ 0.982 \\ 0.993 \end{bmatrix}$$

$$Матрица A = \begin{bmatrix} 1 & -5 & 25 & -125 & 625 \\ 1 & -4 & 16 & -64 & 256 \\ 1 & -3 & 9 & -27 & 81 \\ 1 & -2 & 4 & -8 & 16 \\ 1 & -1 & 1 & -1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 4 & 8 & 16 \\ 1 & 3 & 9 & 27 & 81 \\ 1 & 4 & 16 & 64 & 256 \\ 1 & 5 & 25 & 125 & 625 \end{bmatrix}$$

$$\text{Коэффициенты аппроксимирующего полинома} = \begin{bmatrix} 0.5 \\ 0.192 \\ -1.11022 \cdot 10^{-16} \\ -0.0039 \\ -3.46945 \cdot 10^{-18} \end{bmatrix}$$

Для $n=5$

Список значений $f(x_j) = [0.00669, 0.018, 0.0474, 0.119, 0.269, 0.5, 0.731, 0.881, 0.953, 0.982, 0.993]$

Список значений $p(x_j) = [0.0273, -0.0186, 0.0291, 0.147, 0.312, 0.5, 0.688, 0.853, 0.971, 1.0186, 0.973]$

Максимальные абсолютная и относительная погрешности = 0.0429 , 307.57 %

Для $n=9$

Максимальные абсолютная и относительная погрешности = 0.0027929 , 3.3126 %

Рис. 2.36. Некоторые результаты, рассчитанные программой.

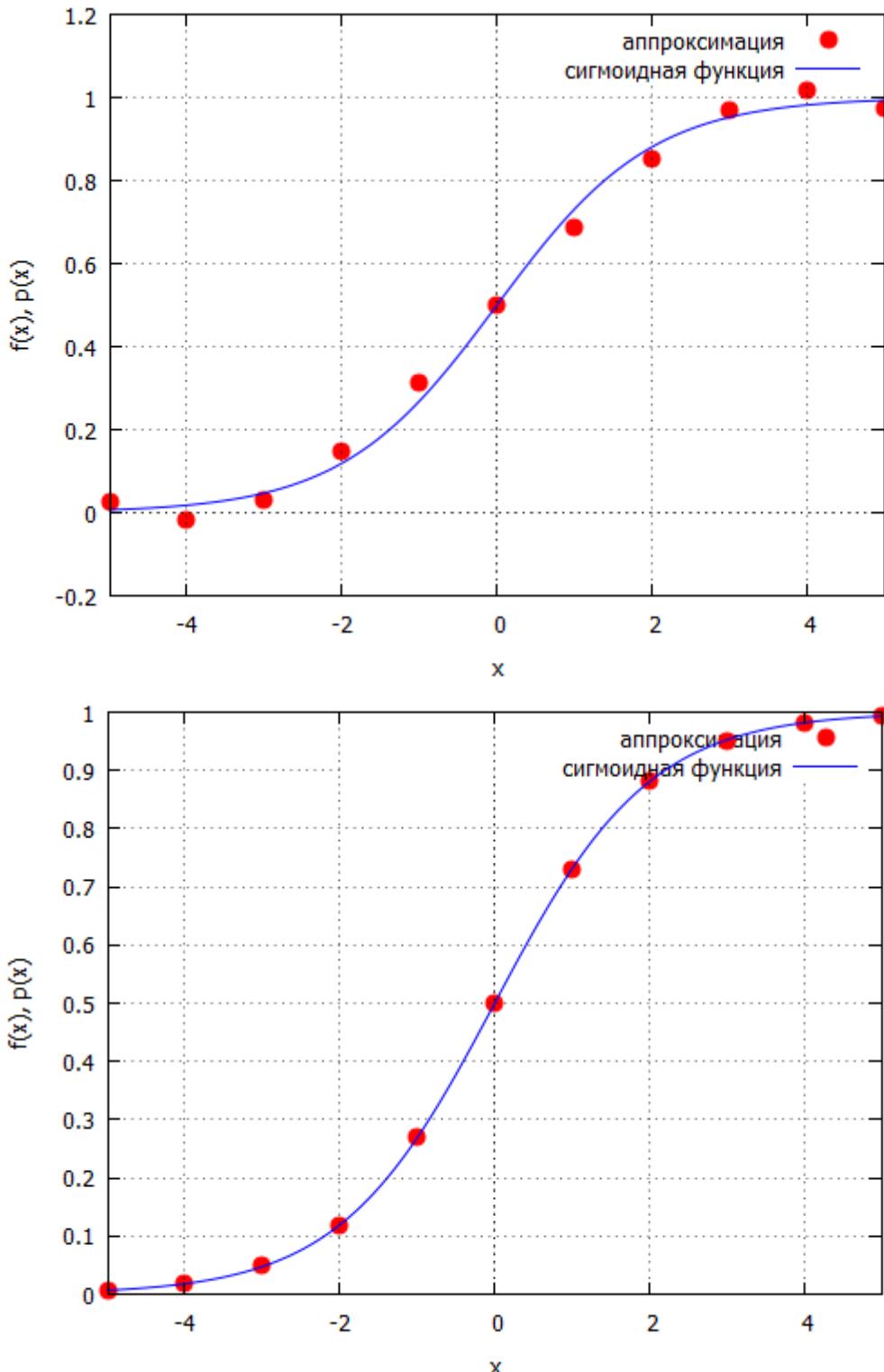


Рис. 2.37. Рисунок демонстрирует качество аппроксимации сигмоидной функции полиномом 4-ой и 8-ой степени по ее значениям в точках $x_j = (-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5)$.

Вопросы и задания для самоконтроля

1. При каком условии вектор $y = (x_1, \dots, x_n, 1)$, используемый в первом методе ортогонализации, будет линейно-независимым от векторов-строк расширенной матрицы A ?
2. Какие манипуляции с векторами выполняет процедура Грамма-Шмидта?
3. Как с помощью процедуры Грамма-Шмидта можно проверить: является ли система векторов линейно независимой?
4. Для каких систем линейных уравнений можно использовать второй метод ортогонализации и почему?
5. В чем сущность метода квадратного корня и почему он применяется только для систем с симметричной положительно определенной матрицей?
6. Запишите алгоритм модифицированного метода ортогонализации Грамма-Шмидта.
7. Как подбирается параметр α в методе регуляризации, который используется при решении плохо обусловленных систем линейных уравнений?
8. Запишите условие сходимости метода прогонки.
9. Составьте алгоритм метода прогонки для системы с пятидиагональной матрицей.
10. Распишите через компоненты скалярное произведение двух столбцов единичной матрицы $sk(j, k) \equiv (E^j, AE^k)$ и составьте алгоритм для вычисления его значения.
11. Сравните норму нормального псевдорешения и нормы всех частных решений, полученных из исходной системы после приравнивания единице «лишних» компонент вектора x . Выполните задание в пакете Maxima.

Глава 3. Методы решения задачи на собственные значения

3.1. Собственные значения и собственные векторы матрицы

Собственным числом (значением) матрицы называется действительное или комплексное число λ , которое удовлетворяет системе

$$Ax = \lambda x \Rightarrow (Ax - \lambda E)x = 0 \quad (3.1)$$

При этом нетривиальный вектор x является собственным вектором соответствующим данному λ .

Для того, чтобы система (3.1) имела нетривиальное решение, должно выполняться равенство

$$\det(Ax - \lambda E) = 0 \quad (3.2)$$

Уравнение (3.2), называемое **характеристическим полиномом** матрицы A , представляет собой полином степени n относительно λ и имеет n корней.

Приведем некоторые свойства собственных значений и векторов.

1. Матрица A порядка $n * n$ имеет n собственных значений.
2. Для каждого уникального λ существует, по крайней мере, один собственный вектор (т.к. не всегда k -кратное собственное число имеет k собственных векторов). Каждый собственный вектор определяет для данной матрицы только собственное направление в пространстве R^n .
3. Собственные векторы, соответствующие различным λ , линейно-независимы.

Доказательство этого утверждения проведем «по принципу индукции».

Имеем $a_1 x_1 = 0 \Rightarrow a_1 = 0$.

Пусть $\sum_{i=1}^k a_i x_i = 0$ при $a_i = 0, i = 1, k$, т.е. вектора x_1, \dots, x_k - линейно-независимы.

Докажем, что вектора x_1, \dots, x_k, x_{k+1} - также линейно-независимые. Доказательство проведем «от противного». Предположим, что это не так, т.е., что

$$\sum_{i=1}^{k+1} a_i x_i = 0 \quad (3.3)$$

и не все $a_i = 0$. Умножим равенство (3.3) на матричный оператор $(A - \lambda_{k+1} E)$

$$\begin{aligned} (A - \lambda_{k+1} E) \sum_{i=1}^{k+1} a_i x_i &= \sum_{i=1}^{k+1} A a_i x_i - \sum_{i=1}^{k+1} \lambda_{k+1} a_i x_i = \\ &= \sum_{i=1}^{k+1} (\lambda_i - \lambda_{k+1}) a_i x_i = \sum_{i=1}^k (\lambda_i - \lambda_{k+1}) a_i x_i = 0 \end{aligned} \quad (3.4)$$

Но по предположению вектора x_1, \dots, x_k - линейно-независимые и поэтому

$$(\lambda_i - \lambda_{k+1}) a_i = 0 \Rightarrow a_i = 0, \text{ т.к. } (\lambda_i - \lambda_{k+1}) \neq 0 \text{ для } i = 1, k$$

Отсюда согласно равенству (3.3) $a_{k+1} = 0$ и, следовательно, в (3.3) все $a_i = 0, i = 1, k + 1$. А это означает, что все вектора x_1, \dots, x_k, x_{k+1} - линейно-независимые.

4. Если все собственные числа матрицы простые, то соответствующие им собственные векторы образуют полную линейно-независимую систему векторов в пространстве R^n . Поэтому любой вектор из этого пространства может быть представлен в виде линейной комбинации этих векторов.

5. Собственные числа действительной, симметричной матрицы являются действительными числами. Кроме того, если матрица положительно определенная, то ее собственные числа положительные.

Пусть x – собственный вектор матрицы A . Тогда

$$(Ax, x) = \lambda(x, x) \quad (3.5)$$

Отсюда, т.к. (Ax, x) - для симметричной матрицы и (x, x) действительные числа, следует, что λ – также действительное. Если матрица положительно определенная, то в (3.5) $(Ax, x) > 0$, $(x, x) > 0$ и поэтому $\lambda > 0$.

6. Симметричная, действительная матрица всегда имеет полную ортонормированную систему собственных векторов, которую можно использовать как базис для пространства R^n .

7. Матрицы A , A^{-1} и $B = \sum_{i=1}^k c_i A^i$ имеют одни и те же собственные вектора, а

собственные числа двух последних матриц $\lambda(A^{-1})$, $\lambda(B)$ связаны с собственными числами $\lambda(A)$ матрицы A следующими соотношениями:

$$Ax = \lambda(A)x \Rightarrow x = \lambda(A)A^{-1}x \Rightarrow A^{-1}x = \frac{1}{\lambda(A)}x \Rightarrow \lambda(A^{-1}) = \frac{1}{\lambda(A)}$$

$$Bx = \lambda(B)x \Rightarrow \sum_{i=1}^k c_i A^i x = \lambda(B)x \Rightarrow \quad (3.6)$$

$$\Rightarrow \sum_{i=1}^k c_i \lambda(A)^i x = \lambda(B)x \Rightarrow \lambda(B) = \sum_{i=1}^k c_i \lambda(A)^i$$

8. Преобразование матрицы A вида $C = B^{-1}AB$, где матрица B – невырожденная ($\det(B) \neq 0$), называется подобным, а сами матрицы A и C подобными. Подобные матрицы имеют одинаковые характеристические полиномы (т.к.

$$\begin{aligned} \det(C - \lambda E) &= \det(B^{-1}AB - \lambda E) = \det(B^{-1}AB - \lambda B^{-1}EB) = \\ &\det(B^{-1}) \det(A - \lambda E) \det(B) = \det(A - \lambda E) \end{aligned}$$

где учтено, что $\det(B^{-1}) = \frac{1}{\det(B)}$ и, следовательно, одинаковые собственные

числа, а их собственные вектора (соответственно x, y) связаны следующим соотношением

$$Cy = \lambda y \Rightarrow B^{-1}ABy = \lambda y \Rightarrow A(By) = \lambda(By) \Rightarrow x = By$$

Если $B^{-1} = B^T$, то преобразование, выполняемое над A с помощью такой матрицы, называется ортогональным подобным преобразованием.

9. Для любой нормы матрицы справедливо неравенство $|\lambda_i| \leq \|A\|$. Оно следует из цепочки соотношений: $Ax = \lambda x \Rightarrow \|A\| \|x\| \geq \|Ax\| = \|\lambda x\| = |\lambda| \|x\|$.

В заключение приведем два полезных соотношения, которые на практике могут быть использованы для нахождения двух последних неизвестных собственных значений или для проверки надежности рассчитанной совокупности собственных значений

$$\det(A) = \lambda_1 \dots \lambda_n$$

$$\text{trace}(A) \equiv \sum_{i=1}^n A_{ii} = \lambda_1 + \dots + \lambda_n$$

Подумайте. Будут ли эти соотношения выполняться для действительной матрицы при наличии у нее комплексных собственных значений?

3.2. Решение частичной проблемы собственных чисел для симметричной матрицы.

Рассмотрим итерационный метод нахождения максимального по модулю собственного числа симметричной матрицы. Все собственные числа такой матрицы действительные, а собственные вектора $e_i, i = 1, n$ образуют в пространстве R^n ортонормированный базис.

Замечание. Этот метод применим и для обычной матрицы, если у нее существует полный набор собственных векторов и собственные числа действительные. В этом случае может пригодиться теорема Пиррона: если все компоненты матрицы положительные ($A_{ij} > 0$), то максимальное собственное число матрицы простое и положительное $\lambda_1 > 0$.

Случай 1.

Пусть $\lambda_1 = \lambda_2 = \dots = \lambda_r$ и $|\lambda_1| > |\lambda_{r+1}| > \dots > |\lambda_n|$. Возьмем произвольный вектор $x^{(0)} = \sum_{i=1}^n c_i e_i$ и будем последовательно умножать его на матрицу A

$$x^{(k)} = Ax^{(k-1)} = \sum_{i=1}^n c_i \lambda_i^k e_i \quad (3.7)$$

Для полученных таким образом векторов можно построить следующие скалярные произведения (учтено, что вектора $e_i, i = 1, n$ ортонормированы)

$$\begin{aligned} (x^{(k-1)}, x^{(k-1)}) &= \sum_{i=1}^n \sum_{j=1}^n c_i c_j \lambda_i^{k-1} \lambda_j^{k-1} (e_i, e_j) = \sum_{i=1}^n c_i^2 \lambda_i^{2k-2} = \\ &= \lambda_1^{2k-2} (c_1^2 + c_2^2 + \dots + c_r^2 + O(\gamma^{2k-2})) \quad (3.8) \\ (x^{(k)}, x^{(k-1)}) &= \sum_{i=1}^n \sum_{j=1}^n c_i c_j \lambda_i^k \lambda_j^{k-1} (e_i, e_j) = \sum_{i=1}^n c_i^2 \lambda_i^{2k-1} = \\ &= \lambda_1^{2k-1} (c_1^2 + c_2^2 + \dots + c_r^2 + O(\gamma^{2k-1})) \end{aligned}$$

где $\gamma = \frac{\lambda_{r+1}}{\lambda_1} < 1$.

Отношение этих скалярных произведений при $k \rightarrow \infty$ стремится к максимальному по модулю собственному числу матрицы A

$$\frac{(x^{(k)}, x^{(k-1)})}{(x^{(k-1)}, x^{(k-1)})} = \lambda_1 + O(\gamma^{2k-2}) \approx \lambda_1 \quad (3.9)$$

При этом один из собственных векторов, соответствующих данному кратному собственному числу (поскольку любая линейная комбинация этих собственных векторов также является собственным вектором $A(\sum_{i=1}^r c_i e_i) = \lambda_1 (\sum_{i=1}^r c_i e_i)$, т.е. эти вектора определяют r -мерную собственную гиперплоскость для λ_1), определяется следующим выражением

$$\frac{x^{(k)}}{\|x^{(k)}\|} = \frac{\sum_{i=1}^n c_i \lambda_i^k e_i}{\sqrt{\sum_{i=1}^n c_i^2 \lambda_i^{2k}}} = \frac{\lambda_1^k (c_1 e_1 + c_2 e_2 + \dots + c_r e_r + O(\gamma^k))}{\sqrt{\lambda_1^{2k} (c_1^2 + c_2^2 + \dots + c_r^2 + O(\gamma^{2k}))}} = e_1 + O(\gamma^k) \approx e_1 \quad (3.10)$$

Чтобы для λ_1 найти остальные собственные векторы, необходимо повторить описанную итерационную процедуру, но с другими начальными векторами.

В заключение приведем модифицированный вариант итерационной процедуры, который позволяет избежать возможных аварийных ситуаций, связанных с переполнением (когда $|\lambda_1| > 1$) или «исчезновением порядка» (когда $|\lambda_1| < 1$)

$$e_1^{(0)} = \frac{x^{(0)}}{\|x^{(0)}\|}, \quad \text{где } \|x^{(0)}\| = \sqrt{(x^{(0)}, x^{(0)})}$$

$$\lambda_1^{(0)} = 10^{30}$$

для $k = 1, K_{\max}$ - максимальное число итераций

$$x^{(k)} = A e_1^{(k-1)}, \quad \lambda_1^{(k)} = (x^{(k)}, e_1^{(k-1)}), \quad e_1^{(k-1)} = \frac{x^{(k)}}{\|x^{(k)}\|} \quad (3.11)$$

если $|\lambda_1^{(k)} - \lambda_1^{(k-1)}| < \text{eps}$, то выход из цикла

иначе $\lambda_1^{(k-1)} = \lambda_1^{(k)}$

конец цикла по k

Такая запись справедлива, т.к.

$$\frac{(x^{(k)}, x^{(k-1)})}{(x^{(k-1)}, x^{(k-1)})} = \frac{(Ax^{(k-1)}, x^{(k-1)})}{\|x^{(k-1)}\| \|x^{(k-1)}\|} = (Ae^{(k-1)}, e^{(k-1)})$$

Случай 2 – он не возможен для положительно определенной матрицы, т.к. у нее все $\lambda_i > 0$.

Пусть $\lambda_1 = -\lambda_2$ и $|\lambda_1| = |\lambda_2| > |\lambda_3| \dots > |\lambda_n|$. Тогда

$$\begin{aligned}
 s1 &\equiv \frac{(x^{(k)}, x^{(k-1)})}{(x^{(k-1)}, x^{(k-1)})} = \frac{\lambda_1^{2k-1}(c_1^2 - c_2^2 + O(\gamma^{2k-1}))}{\lambda_1^{2k-2}(c_1^2 + c_2^2 + O(\gamma^{2k-2}))} \rightarrow \lambda_1 \left(\frac{c_1^2 - c_2^2}{c_1^2 + c_2^2} \right) + O(\gamma^{2k-2}) \\
 s2 &\equiv \frac{(x^{(k)}, x^{(k)})}{(x^{(k-1)}, x^{(k-1)})} = \frac{\lambda_1^{2k}(c_1^2 + c_2^2 + O(\gamma^{2k-1}))}{\lambda_1^{2k-2}(c_1^2 + c_2^2 + O(\gamma^{2k-2}))} \rightarrow \lambda_1^2 + O(\gamma^{2k-2}) \\
 e^{(k)} &\equiv \frac{x^{(k)}}{\|x^{(k)}\|} = \frac{\lambda_1^k(c_1 e_1 + (-1)^k c_2 e_2 + O(\gamma^k))}{\sqrt{\lambda_1^{2k}(c_1^2 + c_2^2 + O(\gamma^{2k}))}} \rightarrow \\
 &\rightarrow a(c_1 e_1 + (-1)^k c_2 e_2) + O(\gamma^k),
 \end{aligned} \tag{3.12}$$

где $a = \text{const}$.

Видно, что в этом случае при $k \rightarrow \infty$ $(s1)^2 \neq s2$ и вектор $e^{(k)}$ поочередно меняет свое «значение». Оба собственных вектора e_1 и e_2 можно определить из соотношений (здесь k - четное)

$$\begin{aligned}
 e^{(k)} &= ac_1 e_1 + ac_2 e_2 \\
 e^{(k-1)} &= ac_1 e_1 - ac_2 e_2
 \end{aligned}$$

следующим образом (т.к. собственные вектора определяются с точностью до константы)

$$\begin{aligned}
 \tilde{e}_1 &\equiv 2ac_1 e_1 = e^{(k)} + e^{(k-1)} \\
 \tilde{e}_2 &\equiv 2ac_2 e_2 = e^{(k)} - e^{(k-1)}
 \end{aligned} \tag{3.13}$$

Ниже представлен обобщенный модифицированный алгоритм нахождения максимального по модулю собственного числа, учитывающий возможность появления любого из случаев:

$$e_1^{(0)} = \frac{x^{(0)}}{\|x^{(0)}\|}, \quad \varrho e \|x^{(0)}\| = \sqrt{(x^{(0)}, x^{(0)})}, \quad \tilde{\lambda}_1^{(0)} = 10^{30}$$

для $k = 1, K_{\max}$ - максимальное число итераций

$$x^{(k)} = A e_1^{(k-1)}, \quad \lambda_1^{(k)} = (x^{(k)}, e_1^{(k-1)})$$

$$\tilde{\lambda}_1^{(k)} \equiv \|x^{(k)}\|^2 = (x^{(k)}, x^{(k)}), \quad e_1^{(k)} = \frac{x^{(k)}}{\|x^{(k)}\|} \quad (3.14)$$

если $|\tilde{\lambda}_1^{(k)} - \tilde{\lambda}_1^{(k-1)}| < \text{eps}$, то выход из цикла, иначе $\tilde{\lambda}_1^{(k-1)} = \tilde{\lambda}_1^{(k)}$)

конец цикла по k

если $|(\lambda_1^{(k)})^2 - \tilde{\lambda}_1^{(k)}| = 0,0001$, то $\lambda_1 = \lambda_1^{(k)}$ - имеет место случай 1

иначе $\lambda_1 = \sqrt{\tilde{\lambda}_1^{(k)}}$ - имеет место случай 2

Посмотрим, какую информацию относительно собственных значений матрицы A можно получить, если применить процедуру (3.14) к

симметричным матрицам вида: $B = E - \frac{A^2}{\lambda_1^2}$ ($B_{ij} = \delta_{ij} - \frac{\sum_{m=1}^n A_{im} A_{mj}}{\lambda_1^2}; i, j = 1, n$) и

$C = \lambda_1 E - A$. Принадлежность собственных значений различным матрицам будем обозначать следующим образом: $\lambda_i(A), \lambda_i(B), \lambda_i(C)$.

1. Определение минимального по модулю собственного числа матрицы A .

Его можно рассчитать, как

$$\lambda_1(B) = 1 - \frac{\lambda_n^2(A)}{\lambda_1^2(A)} \Rightarrow |\lambda_n(A)| = |\lambda_1(A)| \sqrt{1 - \lambda_1(B)}$$

или, если матрицы A положительно определенная ($A > 0$), то матрица $C \geq 0$ и поэтому

$$\lambda_1(C) = \lambda_1(A) - \lambda_n(A) > 0 \Rightarrow \lambda_n(A) = \lambda_1(A) - \lambda_1(C)$$

2. Определение границ отрезка, содержащего все собственные значения матрицы A .

- Если при реализации процедуры (3.14) имеет место случай 2 ($\lambda_1 = -\lambda_2$), то

$$\lambda_{\min}(A) = -\lambda_1(A), \quad \lambda_{\max}(A) = \lambda_1(A)$$

- Если матрица A положительно определенная, то

$$\lambda_{\min}(A) = \lambda_n(A) = \lambda_1(A) - \lambda_1(C) > 0, \quad \lambda_{\max}(A) = \lambda_1(A) > 0$$

- Если $\lambda_1(C) > 0, \lambda_1(A) > 0$, то

$$\lambda_{\min}(A) = \lambda_1(A) - \lambda_1(C) < 0, \quad \lambda_{\max}(A) = \lambda_1(A) > 0$$

- Если $\lambda_1(A) < 0, \lambda_1(C) < 0$, то

$$\lambda_{\min}(A) = \lambda_1(A) < 0, \quad \lambda_{\max}(A) = \lambda_1(A) - \lambda_1(C) < 0 \text{ или } > 0$$

На рис. 3.1 и 3.2 приведены скриншоты программы, в которой использован алгоритм (3.14). Программа позволяет задать «вручную» или случайным образом произвольную исходную матрицу A , найти для нее максимальное по модулю собственное число, затем с помощью этих данных сформировать два варианта вспомогательной матрицы B и определить для нее максимальное по модулю собственное число. Используя эту информацию, пользователь может рассчитать минимальное по модулю собственное число матрицы A и границы отрезка, содержащего все ее собственные значения. Например, согласно результатам, представленным на рис. 3.1 и 3.2, следует, что все собственные числа матрицы лежат на отрезке

$$\lambda_{\min}(A) = \lambda_1(A) = -0,83092;$$

$$\lambda_{\max}(A) = \lambda_1(A) - \lambda_1(B) = -0,83092 - (-1,41574) = 0,58482$$

Нахождение итерационным методом максимального по модулю собственного числа - L1 произвольной матрицы - A

справка выход

Размерность матрицы (>1)		4
------------------------------	--	---

Исходная матрица - A и начальный вектор X0					
i \ j	1	2	3	4	X0
1	-0,047466	0,367333	0,093211	-0,432751	1
2	0,367333	-0,497383	-0,313436	-0,022929	1
3	0,093211	-0,313436	0,14101	-0,14469	1
4	-0,432751	-0,022929	-0,14469	0,156563	1

Сформированная матрица B					
i \ j	1	2	3	4	
1					
2					
3					
4					

Строка, столбец и задаваемое значение компоненты:

<input type="text"/>	<input type="text"/>	<input type="text"/>
----------------------	----------------------	----------------------

Вид матрицы

- $B = L1^*E - A$ - $B = E - A^*A / L1^*L1$

матрица A - симметричная
 A - задать случайным образом [равномерно распределенные числа [-0,5; 05]]

Сформировать матрицу B

Число итераций Требуемая точность **Найти L1**

Выбор матрицы

- для матрицы A - для матрицы B L1 - максимальное по модулю собственное число выбранной матрицы

Рис. 3.1. Задана случайным образом симметричная исходная матрица A и найдено ее максимальное по модулю собственное число

Нахождение итерационным методом максимального по модулю собственного числа - L1 произвольной матрицы - A

справка выход

Размерность матрицы (>1)

Исходная матрица - A и начальный вектор X0					
i \ i	1	2	3	4	X0
1	-0.047466	0.367333	0.093211	-0.432751	1
2	0.367333	-0.497383	-0.313436	-0.022929	1
3	0.093211	-0.313436	0.14101	-0.14469	1
4	-0.432751	-0.022929	-0.14469	0.156563	1

Сформированная матрица B					
i \ i	1	2	3	4	
1	-0.783454	-0.367333	-0.093211	0.432751	
2	-0.367333	-0.333537	0.313436	0.022929	
3	-0.093211	0.313436	-0.97193	0.14469	
4	0.432751	0.022929	0.14469	-0.987483	

Строка, столбец и задаваемое значение компоненты:

Вид матрицы - B = L1 * E - A - B = E - A * A / L1 * L1

матрица A - симметричная

A - задать случайным образом [равномерно распределенные числа [-0,5; 0,5]]

Число итераций Требуемая точность

Выбор матрицы - для матрицы A - для матрицы B L1 - максимальное по модулю собственное число выбранной матрицы

Рис. 3.2. Для исходной матрицы сформирована вспомогательная матрица B и найдено ее максимальное по модулю собственное число.

В таблице 3.1 приведены результаты небольшого исследования, выполненного с помощью указанной программы. Строгая регулярность в представленных результатах отсутствует.

Таблицы 3.1. Зависимость числа итераций от размерности случайной матрицы для $\epsilon_{\text{eps}} = 0,000001$. Величина выборки -5.

Размерность A	5	10	15	20
Мин. итераций	17	26	49	27
Max. итераций	44	96	196	123

Вопросы и задания для самоконтроля

- Для чего производят нормировку текущего вектора $e_1^{(k)} = \frac{x^{(k)}}{\|x^{(k)}\|}$ при реализации итерационного метода нахождения максимального собственного значения?

2. Что такое характеристический полином матрицы?
3. Почему рассмотренный итерационный метод нахождения максимального по модулю собственного значения не всегда можно применить для произвольной матрицы?
4. Как с помощью рассмотренного итерационного метода найти минимальное по модулю собственное значение матрицы?
5. Как связаны между собой собственные значения и собственные вектора следующих матриц A и $B = \sum_{i=0}^k A^i$?
6. Составьте алгоритмы для вычисления компонент матриц $B = E - \frac{A^2}{\lambda_1^2}$ и $C = \lambda_1 E - A$.
7. Какие матрицы называются подобными?
8. Чем можно объяснить отсутствие строгой регулярности в результатах, представленных в таблице 3.1?

3.3. Решение задачи на собственные значения методом Данилевского

Сущность метода Данилевского состоит в приведении матрицы A с помощью $(n-1)$ подобных преобразований к подобной ей матрице Фробениуса P

$$P = \begin{bmatrix} p_1 & p_2 & p_3 & \dots & p_{n-1} & p_n \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ \dots & & & & & \\ 0 & 0 & 0 & \dots & 1 & 0 \end{bmatrix} \quad (3.15)$$

Для последней характеристический полином записывается просто путем разложения по элементам первой строки

$$\begin{aligned} \det(P - \lambda E) &= \begin{bmatrix} p_1 - \lambda & p_2 & p_3 & \dots & p_{n-1} & p_n \\ 1 & -\lambda & 0 & \dots & 0 & 0 \\ 0 & 1 - \lambda & \dots & 0 & 0 & \\ \dots & & & & & \\ 0 & 0 & 0 & \dots & 1 & -\lambda \end{bmatrix} = \\ &= (p_1 - \lambda)(-\lambda)^{n-1} - p_2(-\lambda)^{n-2} + \dots + (-1)^{n+1} p_n \Rightarrow \quad (3.16) \\ &\lambda^n - p_1 \lambda^{n-1} - p_2 \lambda^{n-2} - \dots - p_n = 0 \end{aligned}$$

Подобное преобразование матрицы A представляет собой последовательный перевод ее строк, начиная с n -ой, в строки матрицы Фробениуса.

Детально рассмотрим k -ое ($k = 1, 2, \dots, (n-1)$) преобразование, формирующее $(n-(k-1))$ -ую строку матрицы P .

После $(k-1)$ -го преобразования матрица имеет вид ($A^{(0)} = A$)

$$A^{(k-1)} = \begin{bmatrix} A_{11}^{(k-1)} & \dots & A_{1n-k}^{(k-1)} & A_{1n-(k-1)}^{(k-1)} & \dots & A_{1n-1}^{(k-1)} & A_{1n}^{(k-1)} \\ \dots & & \dots & & & & \\ A_{n-(k-1)1}^{(k-1)} & \dots & A_{n-(k-1)n-k}^{(k-1)} & A_{n-(k-1)n-(k-1)}^{(k-1)} & \dots & A_{n-(k-1)n-1}^{(k-1)} & A_{n-(k-1)n}^{(k-1)} \\ 0 & \dots & 0 & 1 & \dots & 0 & 0 \\ \dots & & \dots & & & & \\ 0 & \dots & 0 & 0 & \dots & 1 & 0 \end{bmatrix} \quad (3.17)$$

Над ней выполняются следующие операции:

1. «Сажаем» единицу на место компонента $A_{n-(k-1)n-k}^{(k-1)}$, путем деления на него всего $(n-k)$ -го столбца

$$\tilde{A}_{in-k}^{(k)} = \frac{A_{in-k}^{(k-1)}}{A_{n-(k-1)n-k}^{(k-1)}}, \quad i = 1, n-(k-1) \quad (3.18)$$

2. Обнуляем все остальные компоненты приводимой строки. Для этого вычитаем из каждого j -го столбца $(n-k)$ -ый столбец, умноженный на соответствующий компонент $A_{n-(k-1)j}^{(k-1)}$ $(n-(k-1))$ -ой строки

$$\tilde{A}_{ij}^{(k)} = A_{ij}^{(k-1)} - \tilde{A}_{in-k}^{(k)} A_{n-(k-1)j}^{(k-1)}, \quad i = 1, n-(k-1), j = 1, n \quad (j \neq n-k) \quad (3.19)$$

Эти операции равносильны матричной операции

$$\tilde{A}^{(k)} = A^{(k-1)} M^{(n-k)},$$

где матрица $M^{(n-k)}$ отличается от единичной только $(n-k)$ -ой строкой, компоненты которой равны

$$M_{n-k j}^{(n-k)} = -\frac{A_{n-(k-1)j}^{(k-1)}}{A_{n-(k-1)n-k}^{(k-1)}}, \quad j = 1, n \quad (j \neq n-k) \quad (3.20)$$

$$M_{n-k n-k}^{(n-k)} = \frac{1}{A_{n-(k-1)n-k}^{(k-1)}}$$

Чтобы завершить k -ое подобное преобразование необходимо умножить промежуточную матрицу $\tilde{A}^{(k)}$ на матрицу $(M^{(n-k)})^{-1}$

$$A^{(k)} = (M^{(n-k)})^{-1} \tilde{A}^{(k)} M^{(n-k)} = (M^{(n-k)})^{-1} \tilde{A}^{(k)} \quad (3.21)$$

Последняя получается из единичной матрицы подстановкой в ее $(n-k)$ -ую строку компонент $(n-(k-1))$ -ой строки матрицы $A^{(k-1)}$. Легко проверить, что в этом случае будет выполняться необходимое соотношение

$$(M^{(n-k)})^{-1} M^{(n-k)} = E$$

Матричная операция (3.20) приводит к изменению компонент только одной $(n-k)$ -ой строки $\tilde{A}^{(k)}$

$$A_{n-k}^{(k)} = \sum_{m=1}^n A_{n-(k-1)m}^{(k-1)} \tilde{A}_{mj}^{(k)}, \quad j = 1, n \quad (3.22)$$

или, с учетом имеющихся нулевых компонент в нижних подстолбцах матрицы $\tilde{A}^{(k)}$,

$$A_{n-k}^{(k)} = \begin{cases} \sum_{m=1}^{n-k} A_{n-(k-1)m}^{(k-1)} \tilde{A}_{mj}^{(k)}, & j = 1, n - (k+1); j = n \\ \sum_{m=1}^{n-k} A_{n-(k-1)m}^{(k-1)} \tilde{A}_{mj}^{(k)} + A_{n-(k-1)j}^{(k-1)}, & j = (n-k), (n-1) \end{cases} \quad (3.23)$$

После построения характеристического полинома и нахождения его корней – собственных чисел матрицы A , легко определяются собственные векторы матрицы $P(y)$, а затем и матрицы A , т.к.

$$Py = \lambda y \Rightarrow (M^{-1}AM)y = \lambda y \Rightarrow A(My) = \lambda(My) \Rightarrow x = My \quad (3.24)$$

где $M = M^{(n-1)}M^{(n-2)}...M^{(2)}M^{(1)}$.

Для определения векторов y распишем систему $(P - \lambda E)y = 0$

$$\begin{cases} (p_1 - \lambda)y_1 + p_2y_2 + ... + p_ny_n = 0 \\ y_1 - \lambda y_2 = 0 \\ \\ y_{n-1} - \lambda y_n = 0 \end{cases} \quad (3.25)$$

Положив $y_n = 1$ (это можно сделать, т.к. собственные векторы определяются с точностью до постоянной), из последнего и последующих (до первого) уравнений легко находим

$$y = \{y_n = 1, y_{n-1} = \lambda, y_{n-2} = \lambda^2, \dots, y_1 = \lambda^{n-1}\} \quad (3.26)$$

При таких значениях y_i первое уравнение системы (3.25) удовлетворяется автоматически, поскольку оно совпадает с характеристическим уравнением (3.2).

Теперь с учетом вида матриц $M^{(n-k)}$

$$M^{(n-k)} = \begin{bmatrix} 1 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ M_{n-k1}^{(n-k)} & M_{n-k2}^{(n-k)} & \dots & M_{n-kn-1}^{(n-k)} & M_{n-kn}^{(n-k)} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 & 1 \end{bmatrix} \quad (2.27)$$

легко рассчитываются компоненты собственных векторов x (первоначально устанавливается $x = y$)

$$x_{n-k} = \sum_{j=1}^n M_{n-kj}^{(n-k)} x_j, \quad k = (n-1), 1 \quad (3.28)$$

Остановимся на особенностях программной реализации данного метода.

1. Обрабатываемая матрица хранится на месте исходной.

2. Для повышения устойчивости алгоритма перед выполнением очередного k -го подобного преобразования необходимо выполнять процедуру выбора «ведущего» элемента: найти среди компонент $A_{n-(k-1),j}^{(k-1)}$, $j=1,(n-k)$ максимальный по модулю элемент и, если $j_{\max} \neq (n-k)$, то переставить местами верхние части столбцов и строки с номерами $(n-k)$ и j_{\max} . Такая перестановка $(TA^{(k-1)}T)$ будет подобной, поскольку для матрицы перестановок T справедливо соотношение $T^{-1} = T$. Если окажется, что «ведущий» элемент равен нулю, то имеет место сингулярный случай, когда характеристический полином распадается на произведение двух характеристических полиномов

$$\det(A - \lambda E) = \begin{bmatrix} A^{(n-k)} - \lambda E & B \\ 0 & P^{(k)} - \lambda E \end{bmatrix} = \det(A^{(n-k)} - \lambda E) \det(P^{(k)} - \lambda E)$$

причем для одного из них матрица Фробениуса $P^{(k)}$ уже получена, а для подматрицы $A^{(n-k)}$ ее предстоит еще найти с помощью того же метода. В этом особом случае собственные векторы матрицы следует искать вне данного метода (см. параграф 3.5).

3. Строку $M_{n-k,j}^{(n-k)}$, $j=1,n$ (см. формулы 3.20) матрицы $M^{(n-k)}$ можно первоначально формировать в дополнительном векторе m , а в конце k -го преобразования сохранять на месте $((n-(k-1))-ой$ обработанной строки исходной матрицы. При этом формулы (3.18), (3.19) перепишутся следующим образом

$$\begin{aligned} \tilde{A}_{i,n-k}^{(k)} &= A_{i,n-k}^{(k-1)} m_{n-k}, \quad i=1,(n-k) \\ \tilde{A}_{i,j}^{(k)} &= A_{i,j}^{(k-1)} + A_{i,n-k}^{(k-1)} m_j, \quad i=1,(n-k), \quad j=1,n \quad (j \neq n-k) \end{aligned} \quad (3.29)$$

Здесь исключена явная обработка $((n-(k-1))-ой$ приводимой строки, поскольку ее компоненты используются при завершении k -го подобного преобразования (см. формулы (3.23)).

В соответствии с изложенным выше, была разработана программа, позволяющая построить матрицу Фробениуса для произвольной матрицы, а затем определить методом парабол корни характеристического полинома этой матрицы. Программа успешно работает и случаи вырождения матрицы Фробениуса в две и более подматриц. Скриншоты этой программы представлены на рис. 3.3 – 3.7. Обратите внимание, что, как и положено, сумма следов двух матриц Фробениуса равна следу исходной матрицы ($21+7=28$), а суммы корней каждой из подматриц совпадает с ее следом.

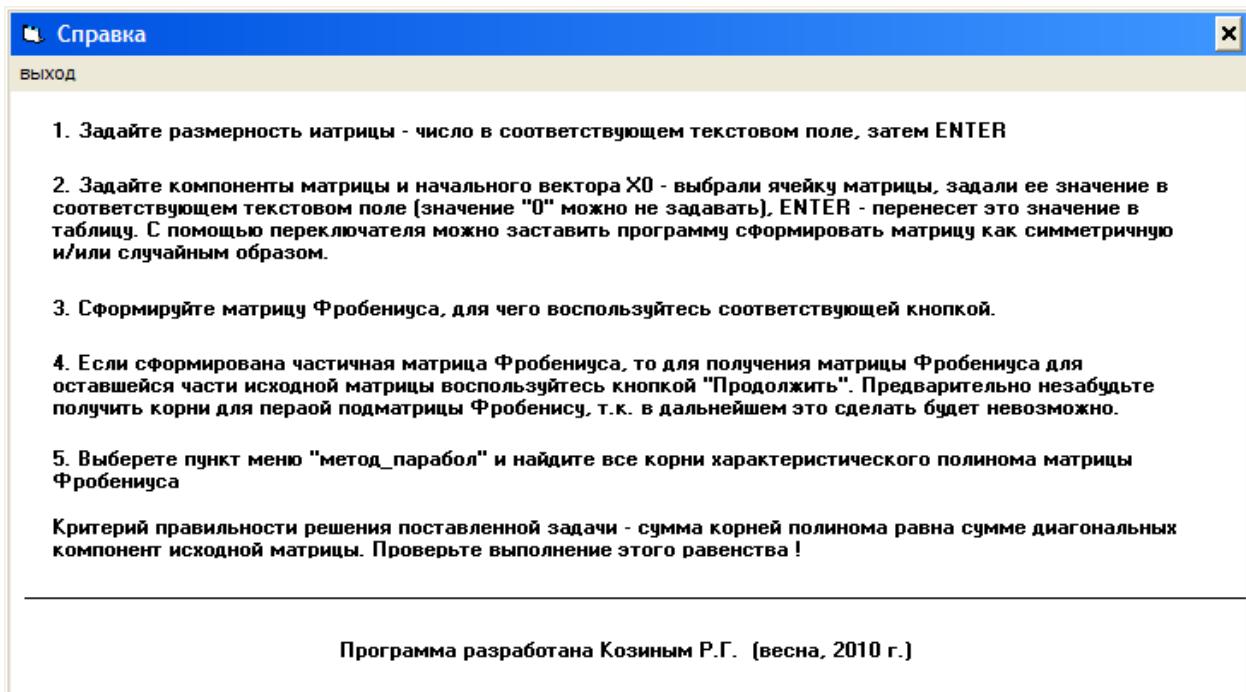


Рис. 3.3. Форма, с описанием порядка работы с программой.

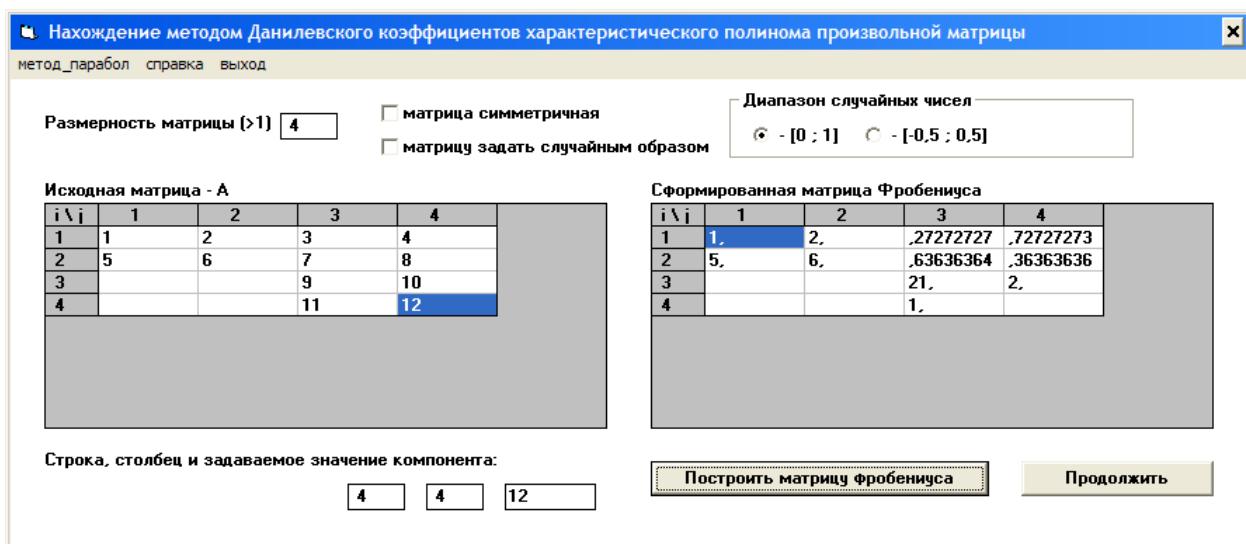


Рис. 3.4. Главная форма программы, на которой приведен пример матрицы, для которой матрица Фробениуса распадается на две подматрицы. Здесь приведен этап получения первой подматрицы.

Нахождение корней характеристического полинома методом парабол

Выход

Число итераций при нахождении корня	1000	Точность	,0000000001	найти корни полинома
-------------------------------------	------	----------	-------------	-----------------------------

№ корня	1	2
действ.часть	21,0948101	-,0948101
мним.часть	,	,
P-действ.часть	,	,
P-мним.часть	,	,

Здесь P - действительная и мнимая части значения полинома для соответствующего корня

След обрабатываемой матрицы Фробениуса 21, Сумма корней полинома (R,Im) 21, ,

В идеале → след матрицы = сумме корней характеристического полинома матрицы

Рис. 3.5. Корни характеристического полинома, полученные методом парабол для первой подматрицы Фробениуса.

Нахождение методом Данилевского коэффициентов характеристического полинома произвольной матрицы

метод_парабол справка выход

Размерность матрицы (>1)	4	<input type="checkbox"/> матрица симметричная	<input type="checkbox"/> Диапазон случайных чисел
		<input type="checkbox"/> матрицу задать случайным образом	<input checked="" type="radio"/> - [0 ; 1] <input type="radio"/> - [-0,5 ; 0,5]

Исходная матрица - A				
i \ i	1	2	3	4
1	1	2	3	4
2	5	6	7	8
3			9	10
4			11	12

Сформированная матрица Фробениуса				
i \ i	1	2	3	4
1	7,	4,	,27272727	,72727273
2	1,		,63636364	,36363636
3			21,	2,
4			1,	

Строка, столбец и задаваемое значение компонента:

4 4 12 Построить матрицу Фробениуса Продолжить

Рис. 3.6. Главная форма программы после получения второй подматрицы Фробениуса.

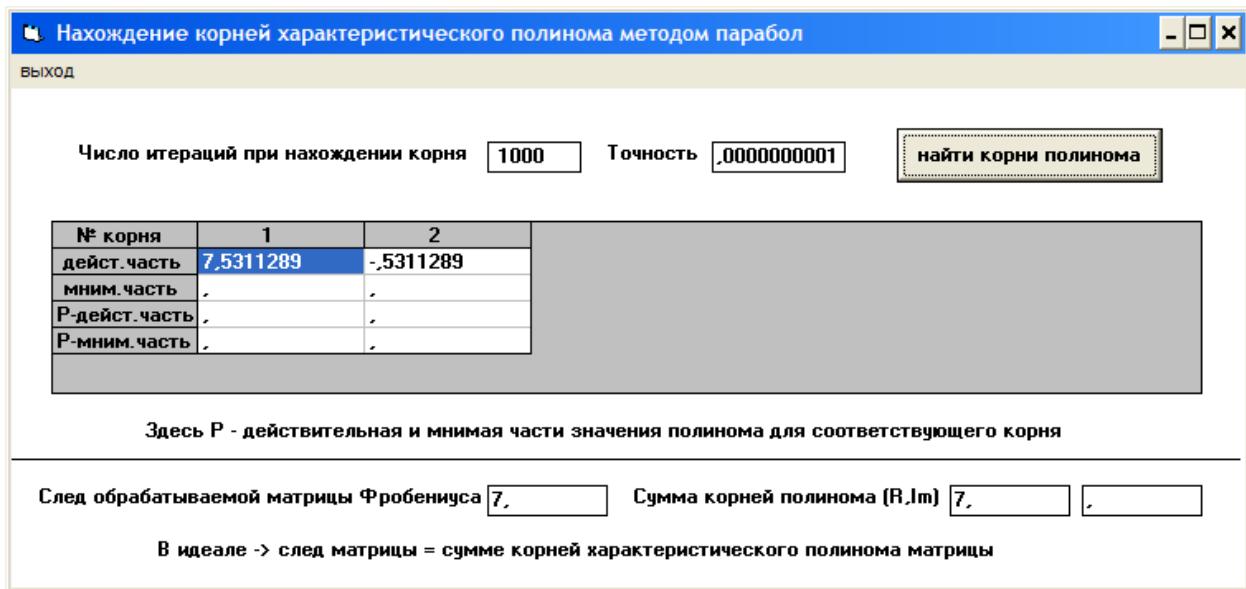


Рис. 3.7. Корни характеристического полинома, полученные методом парабол для второй подматрицы Фробениуса.

Ниже приведены листинги на языке Visual Basic алгоритма, вычисляющего матрицу Фробениуса, и процедуры выбора ведущего элемента, которая используется на каждом шаге алгоритма.

```

For k = 1 To n - 1
    kk = kk + 1
    If v_element(k) = 1 Then   'выделена подматрица Фробениуса
        code = 1
        Exit For
    End If
    For j = 0 To n - 1
        ma(j) = a(n - k, j) 'сохраняем обрабатываемую строку
    Next j
    'ставим 1 в обрабатываемой строке
    For i = 0 To n - k
        a(i, n - 1 - k) = a(i, n - 1 - k) / a(n - k, n - 1 - k)
    Next i
    'ставим 0 вместо остальных элементов обрабатываемой строки
    For i = 0 To n - k
        For j = 0 To n - 1
            If Not (j = n - 1 - k) Then
                a(i, j) = a(i, j) - a(i, n - 1 - k) * a(n - k, j)
            End If
        Next j
    Next i
    'умножаем на обратную матрицу (M)
    For j = 0 To n - 1

```

```

buf = 0
For m = 0 To n - 1
    buf = buf + ma(m) * a(m, j)
Next m
a(n - 1 - k, j) = buf
Next j
Next k

```

```

Function v_element (k As Integer) As Integer
Dim amax#, jmax%, i%, j%, buf#, cd%
cd = 0
amax = Abs(a(n - k, n - 1 - k))
jmax = n - 1 - k
For j = 0 To n - 2 - k
    buf = Abs(a(n - k, j))
    If buf > amax Then
        amax = buf
        jmax = j
    End If
Next j
If Not (k = jmax) Then
    For i = 0 To n - 1
        buf = a(i, n - 1 - k)
        a(i, n - 1 - k) = a(i, jmax)
        a(i, jmax) = buf
    Next i
    For j = 0 To n - 1
        buf = a(n - 1 - k, j)
        a(n - 1 - k, j) = a(jmax, j)
        a(jmax, j) = buf
    Next j
End If
If amax < 1E-30 Then cd = 1
code = cd
v_element = cd
End Function

```

Здесь $ma(n)$ – вспомогательный массив для временного хранения обрабатываемой строки; n – размерность обрабатываемо матрицы; $v_element$ - функция, реализующая процедуру выбора ведущего элемента.

3.4. Модифицированный метод Данилевского.

В модифицированном методе Данилевского исходная матрица A с помощью $(n-1)$ подобных преобразований вида $C^{-1}AC$ приводится к следующей матрице Фробениуса

$$\Phi = \begin{bmatrix} 0 & 0 & \dots & 0 & p_n \\ 1 & 0 & \dots & 0 & p_{n-1} \\ 0 & 1 & \dots & 0 & p_{n-2} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & p_1 \end{bmatrix}$$

для которой характеристическое уравнение записывается в виде

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

Здесь на k -ом ($k=1 \div (n-1)$) шаге матрицы $C^{(k)}$ и $(C^{-1})^{(k)}$ формируются следующим образом

$$C^{(k)} = \begin{bmatrix} 0 & 0 & \dots & 0 & A_{1n}^{(k-1)} \\ 1 & 0 & \dots & 0 & A_{2n}^{(k-1)} \\ 0 & 1 & \dots & 0 & A_{3n}^{(k-1)} \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & A_{nn}^{(k-1)} \end{bmatrix}, \quad (C^{-1})^{(k)} = \begin{bmatrix} -\frac{A_{2n}^{(k-1)}}{A_{1n}^{(k-1)}} & 1 & 0 & \dots & 0 \\ -\frac{A_{3n}^{(k-1)}}{A_{1n}^{(k-1)}} & 0 & 1 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ -\frac{A_{nn}^{(k-1)}}{A_{1n}^{(k-1)}} & 0 & 0 & \dots & 1 \\ \frac{1}{A_{1n}^{(k-1)}} & 0 & 0 & \dots & 0 \end{bmatrix}$$

Для повышения устойчивости метода к ошибкам округления и для диагностики сингулярного случая (случай, когда характеристический полином распадается на произведение нескольких полиномов) перед выполнением очередного k -ого преобразования необходимо производить поиск максимального по модулю элемента среди компонент $A_{in}^{(k-1)}, i=1 \div (n-k)$. При этом, если максимальный элемент не совпадает с компонентом $A_{1n}^{(k-1)}$, то выполняется подобное преобразование путем перестановки соответствующих столбцов и соответствующих строк. Кроме того, если этот элемент равен нулю, то имеет место сингулярный случай. Тогда для завершения построения матрицы Фробениуса, необходимо продолжить преобразования, но уже для оставшейся необработанной левой верхней подматрицы размерности $(n-k) * ((n-k))$.

$$A^{(k-1)} = \begin{bmatrix} A_{11}^{(k-1)} & \dots & \dots & A_{1n-k}^{(k-1)} & 0 & \dots & 0 & 0 \\ \vdots & \ddots & \dots & \vdots & 0 & \dots & 0 & 0 \\ A_{n-k1}^{(k-1)} & \dots & \dots & A_{n-kn-k}^{(k-1)} & 0 & \dots & 0 & 0 \\ A_{n-k+11}^{(k-1)} & \dots & \dots & 1 & 0 & \dots & 0 & A_{n-k+1n}^{(k-1)} \\ \vdots & \ddots & \dots & \vdots & \vdots & \dots & \vdots & \vdots \\ A_{nl}^{(k-1)} & \dots & \dots & 0 & 0 & \dots & 1 & A_{nn}^{(k-1)} \end{bmatrix}$$

Детали алгоритма, реализующего все процедуры модифицированного метода Данилевского, представлены ниже в виде программы на языке математического пакета Maxima.

Листинг программы для пакета Maxima

```

kill(all);
numer:true;
fpprintprec:5;
/* генерация шаблонов матриц C и C-1 */
genCC(n):=block
(
  c:zeromatrix(n,n), c1:zeromatrix(n,n),
  for i:2 thru n do c[i,i-1]:1,
  for j thru n-1 do c1[j,j+1]:1
);
/* текущий расчет матриц C и C-1 */
doCC(n):=block
(
  for i:1 thru n do c[i,n]:a[i,n],
  c1[n,1]:1/a[1,n],
  for j thru n-1 do c1[j,1]:-a[j+1,n]/a[1,n]
);
/* выбор ведущего элемента в столбце n*/
select(k,n):=block
(
  cod:0,
  amod:abs(a[1,n]), imax:1,
  for i:2 thru n-k do
  (
    buf:abs(a[i,n]), if buf>amod then (amod:buf, imax:i)
  ),
  if amod<1.0e-10 then return(cod:1), /* сингулярный случай */
  if imax#1 then
  (
    for i thru n do /* переставляем столбцы */

```

```

(
    buf:a[i,1], a[i,1]:a[i,imax], a[i,imax]:buf
),
for j thru n do /* переставляем строки */
(
    buf:a[1,j], a[1,j]:a[imax,j], a[imax,j]:buf
)
)
);
/* модифицированный метод Данилевского */
mod_danil(n):=block
(
    genCC(n),
    for k thru n-1 do
    (
        select(k,n),
        if cod=1 then (print("Сингулрный случай на шаге = ",k), return(k)),
        doCC(n), a:a.c, a:c1.a
    )
);
/* формируем подматрицу для сингулярного случая */
/* удаляем лишние нижние строки и правые столбцы */
reform(k):=block
(
    for i:n-k thru n-1 do a:submatrix(i+1,a,i+1),
    n:n-k
);
/* тестирование */
n:3; a:matrix([1,2,3],[2,3,4],[3,4,5]); genCC(n); c;c1; doCC(n); c; c1;
n:3; a:matrix([1,2,0],[3,4,5],[6,7,8]); k:1; select(k,n); a;
n:3; a:matrix([1,2,0],[3,4,5],[6,7,8]); mod_danil(n); a;
n:2; a:matrix([1,2],[3,4]); mod_danil(n); a;
n:4; a:matrix([1,2,3,0],[4,5,6,7],[2,3,4,0],[5,6,7,8]); mod_danil(n); a;
if cod=1 then
(
    b:copy(a), reform(s), mod_danil(n)
);
a;
for i thru n do for j thru n do b[i,j]:=a[i,j];
print("Сингулярная матрица Фробениуса - ",b);

```

Некоторые результаты, выводимые программой

Исходная матрица

$$(\%o9) \begin{bmatrix} 1 & 2 & 3 \\ 2 & 3 & 4 \\ 3 & 4 & 5 \end{bmatrix}$$

Шаблоны для матриц C, C^{-1}

$$(\%o11) \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad (\%o12) \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

Матрицы C, C^{-1} на первом шаге

$$(\%o14) \begin{bmatrix} 0 & 0 & 3 \\ 1 & 0 & 4 \\ 0 & 1 & 5 \end{bmatrix} \quad (\%o15) \begin{bmatrix} -1.3333 & 1 & 0 \\ -1.6667 & 0 & 1 \\ 0.333 & 0 & 0 \end{bmatrix}$$

Исходная матрица

$$(\%o17) \begin{bmatrix} 1 & 2 & 0 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

После выбора ведущего элемента переставлены столбцы и строки

$$(\%o20) \begin{bmatrix} 4 & 3 & 5 \\ 2 & 1 & 0 \\ 7 & 6 & 8 \end{bmatrix}$$

Матрица Фробениуса для этой матрицы

$$(\%o24) \begin{bmatrix} 0.0 & 0.0 & 9.0 \\ 1.0 & 0.0 & -3.0 \\ 0.0 & 1.0 & 13.0 \end{bmatrix}$$

Исходная матрица

$$(\%o30) \begin{bmatrix} 1 & 2 & 0 & 0 \\ 4 & 5 & 0 & 0 \\ 2 & 3 & 4 & 2 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

Сингулрный случай на шаге = 2

$$(\%o32) \begin{bmatrix} 5 & 4 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ -6 & -3 & 0 & -18 \\ 1.5 & 1.0 & 1.0 & 12.0 \end{bmatrix}$$

Вырезанная матрица и ее матрица Фробениуса

$$(3.36) \quad \begin{bmatrix} 5 & 4 \\ 2 & 1 \end{bmatrix} \quad (3.38) \quad \begin{bmatrix} 0.0 & 3.0 \\ 1.0 & 6.0 \end{bmatrix}$$

Сингулярная матрица Фробениуса –

$$\begin{bmatrix} 0.0 & 3.0 & 0 & 0 \\ 1.0 & 6.0 & 0 & 0 \\ -6 & -3 & 0 & -18 \\ 1.5 & 1.0 & 1.0 & 12.0 \end{bmatrix}$$

В заключении заметим, что преимущество модифицированного метода перед стандартным методом Данилевского – простая логика, а недостаток – дополнительный расход памяти (2^*n^*n) и значительное число «ненужных» операций (умножение на нулевые компоненты вспомогательных матриц C, C^{-1}).

3.5. Метод Крылова

В данном методе нахождение коэффициентов характеристического полинома базируется на использовании матричного тождества Гамильтона – Кели

$$A^n + p_1 A^{n-1} + p_2 A^{n-2} + \dots + p_n E = 0 \quad (3.30)$$

где p_i – коэффициенты характеристического полинома

$$\det(A - \lambda E) = \lambda^n + p_1 \lambda^{n-1} + p_2 \lambda^{n-2} + \dots + p_n = 0$$

Возьмем произвольный вектор $y^{(0)} \neq 0$ и умножим на него (3.30). В результате получим векторное уравнение относительно неизвестных p_i

$$A^n y^{(0)} + p_1 A^{n-1} y^{(0)} + p_2 A^{n-2} y^{(0)} + \dots + p_n y^{(0)} = 0$$

или в новых обозначениях $y^{(k)} = A y^{(k-1)} = A^k y^{(0)}$

$$p_1 y^{(n-1)} + p_2 y^{(n-2)} + \dots + p_n y^{(0)} = -y^{(n)} \quad (3.31)$$

Эту систему линейных уравнений можно переписать в привычной матричной форме

$$Yp = -y^{(n)} \quad (3.32)$$

где

$$Y = \begin{bmatrix} y^{(n-1)} & y^{(n-2)} & \dots & y^{(0)} \end{bmatrix} = \begin{bmatrix} Y_{00} & \dots & Y_{0n-1} \\ \dots & \dots & \dots \\ Y_{n-10} & \dots & Y_{n-1n-1} \end{bmatrix}$$

$$p = (p_1, p_2, \dots, p_n)$$

Приведем алгоритм для расчета матрицы Y и вектора $y^{(n)}$ (не забудте, что правый столбец матрицы $Y_{in-1} \equiv y_i^{(0)}$ задается пользователем)

$$\text{для } k = 1, (n-1) \quad \{\text{для } i = 0, (n-1) \quad \{y_i^{(k)} \equiv Y_{i,n-1-k} = \sum_{m=0}^{n-1} A_{im} Y_{mn-1-(k-1)}\}\}$$

$$\text{для } i = 0, (n-1) \quad \{y_i^{(n)} = \sum_{m=0}^{n-1} A_{im} Y_{m0}\}$$

Система (3.32) решается любым известным методом, например, методом исключения Гаусса. Если матрицы Y окажется особенной (это имеет место, если векторы $y^{(0)}, y^{(1)}, \dots, y^{(n-1)}$ - линейно-зависимы), то следует сменить начальный вектор $y^{(0)}$.

Чтобы в процессе вычисления векторов $y^{(k)}$ не произошло переполнения, исходную матрицу целесообразно предварительно отнормировать - $A_{ij} = \frac{A_{ij}}{\max_{i,j}|A_{ij}|}$. После этого собственные значения отнормированной и

исходной матриц будут связаны соотношением $\lambda_{\text{новое}} = \frac{\lambda_{\text{старое}}}{\max_{i,j}|A_{ij}|}$ (т.к.

$$(cA)x = (c\lambda)x).$$

После определения собственных значений матрицы A ее собственные вектора можно искать в виде разложения по полной линейно-независимой системе векторов $y^{(0)}, y^{(1)}, \dots, y^{(n-1)}$

$$x = \beta_1 y^{(n-1)} + \beta_2 y^{(n-2)} + \dots + \beta_n y^{(0)} \quad (3.33)$$

Для нахождения неизвестных коэффициентов β_i , подставим (3.33) в уравнение $Ax = \lambda x$. Тогда с учетом процедуры построения векторов $y^{(k)}$, получим

$$\beta_1 y^{(n)} + \beta_2 y^{(n-1)} + \dots + \beta_n y^{(1)} = \lambda(\beta_1 y^{(n-1)} + \beta_2 y^{(n-2)} + \dots + \beta_n y^{(0)}) \quad (3.34)$$

Заменим в (3.34) $y^{(n)}$ выражением (3.31) и соберем подобные слагаемые

$$\begin{aligned} & (\beta_2 - p_1\beta_1 - \lambda\beta_1)y^{(n-1)} + (\beta_3 - p_2\beta_1 - \lambda\beta_2)y^{(n-2)} + \dots \\ & + (\beta_n - p_{n-1}\beta_1 - \lambda\beta_{n-1})y^{(1)} - (p_n\beta_1 + \lambda\beta_n)y^{(0)} = 0 \end{aligned}$$

Поскольку вектора $y^{(0)}, y^{(1)}, \dots, y^{(n-1)}$ - линейно-независимые, то коэффициенты в этом равенстве при всех $y^{(k)}$ должны быть равны нулю. Отсюда получаем

$$\begin{aligned} \beta_2 &= (\lambda + p_1)\beta_1 \\ \beta_3 &= \lambda\beta_2 + p_2\beta_1 \equiv (\lambda^2 + p_1\lambda + p_2)\beta_1 \\ &\dots \\ \beta_n &= \lambda\beta_{n-1} + p_{n-1}\beta_1 \equiv (\lambda^{n-1} + p_1\lambda^{n-2} + \dots + p_{n-1})\beta_1 \\ (\lambda^n &+ p_1\lambda^{n-1} + \dots + p_n)\beta_1 = 0 \end{aligned} \quad (3.35)$$

Здесь в последнем соотношении в скобках стоит характеристический полином, который равен нулю. Поэтому, не нарушая это равенство, можно положить

$\beta_1 = 1$. Тогда остальные неизвестные коэффициенты можно рассчитать с помощью рекуррентной формулы

$$\beta_k = \lambda \beta_{k-1} + p_{k-1}, \quad k = 2, n \quad (3.36)$$

Для кратных λ недостающие собственные вектора (если они существуют) следует искать, изменяя вектор $y^{(0)}$. В этом случае необходимо проверить, что полученные собственные вектора $x^{(i)}$ являются линейно-независимыми. Для этой цели можно использовать процедуру ортогонализации, которая проходит до конца только для линейно-независимых векторов.

На рис. 3.8 и 3.9 приведены скриншоты программы, написанной на Visual Basic, которая использует метод Крылова для построения характеристического полинома для производной матрицы, введенной пользователем или случайным образом сформированной самой программой. После ввода матрицы можно построить «систему Крылова» (см. формулу 3.32), а затем найти коэффициенты характеристического полинома исходной матрицы. Система Крылова решается методом оптимального исключения Гаусса-Жордана (см. параграф 2.2.2). Далее методом парабол (см. параграф 4.3) можно определить все корни построенного полинома (собственные значения исходной матрицы). Вместе с корнями программа выдает значения следа матрицы ($\sum_{i=1}^n A_{ii}$) и сумму этих

корней. Сравнительный анализ этих величин позволяет оценить корректность выполнения всех промежуточных этапов решения задачи по нахождению собственных значений исходной матрицы.

Дополнительно на рис. 3.10 представлены листинг и результаты аналогичного поиска собственных значений матрицы методом Крылова в среде пакета Maxima. Видно, что на всех этапах решения задачи данные представленные на рис. 3.8, 3.9 и 3.10 совпадают. При этом трудоемкость получения этих результатов для пакета существенно меньше.

Нахождение методом Крылова коэффициентов характеристического полинома произвольной матрицы

метод_парабол справка выход

Размерность матрицы (>1) матрицу задать случайным образом (равномерно распределенные числа [-0,5; 0,5])

Исходная матрица - A и произвольный вектор X0

i \ j	1	2	3	X0
1	1	2	3	1
2	4	5	6	0
3	7	8	9	1

Матрица и правая часть системы Крылова - [Yn-1, ... Y0]^T P = Yn

i \ j	Y2	Y1	Y0	Y3
1	72,	4,	1,	-1152,
2	162,	10,		-2610,
3	252,	16,	1,	-4068,

Строка, столбец и задаваемое значение компоненты:

Коэффициенты характеристического полинома матрицы A -> $x^n + p(1)x^{n-1} + \dots + p(n)=0$

коэффициент	p(0)	p(1)	p(2)	p(3)
значение	1	-15,	-18,	

Рис. 3.8. Главная экранная форма программы для нахождения коэффициентов характеристического полинома произвольной матрицы методом Крылова.

Нахождение корней характеристического полинома методом парабол

Выход

Число итераций при нахождении корня Точность

№ корня	1	2	3
Действ. часть	16,116844	-1,116844	
Мним. часть			
Р-действ. часть			
Р-мним. часть			

Здесь Р - действительная и мнимая части значения полинома для соответствующего корня

След исходной матрицы Сумма корней полинома (R,Im)

В идеале -> след матрицы = сумме корней характеристического полинома матрицы

Рис. 3.9. Вспомогательная экранная форма программы, для определения корней характеристического полинома исходной матрицы. Форма вызывается пунктом меню «метод парабол» на главной форме рис.3.1.

```
/* формируем систему Крылова для матрицы A и начального вектора y0 */
formKrilov(A,y0):=block
(
  [k,i,m,n], /* локальные переменные */
```

```

n:length(A),
Y:zeromatrix(n,n-1), /* генерация нулевой матрицы размерности n*(n-1) */
Y:addcol(Y,y0),      /* добавляем к матрице Y справа столбец y0 */
for k:2 thru n do
  for i thru n do
    for m thru n do Y[i,n+1-k]:Y[i,n+1-k]+A[i,m]*Y[m,n+2-k],
yn:zeromatrix(n,1),
for i thru n do
  for m thru n do yn[i]:yn[i]-A[i,m]*Y[m,1]
);

/* главная программа */
numer:true;
fpprintprec:6;
A:matrix([1,2,3],[4,5,6],[7,8,9]);
y0:matrix([1],[0],[1]);
formKrilov(A,y0);
print("Матрица и правая часть системы Крылова");
Y;yn;
Y_обр:invert(Y); /* находим обратную матрицу системы Крылова */
n:length(A);
p:zeromatrix(n,1);
/* решаем систему Крылова через обратную матрицу p=Y_обр*yn */
for i thru n do for m thru n do p[i]:p[i]+Y_обр[i,m]*yn[m];

c:[1]; for i thru n do c:append(c,p[i]);
print("Список коэффициентов характеристического полинома = ",c);;
solve(x^3+p[1]*x^2+p[2]*x+p[3],[x]); /* определение корней полинома */;

```

Результаты выполнения программы:

```

(%o8) 
$$\begin{bmatrix} 72 & 4 & 1 \\ 162 & 10 & 0 \\ 252 & 16 & 1 \end{bmatrix}$$

(%o9) 
$$\begin{bmatrix} -1152 \\ -2610 \\ -4068 \end{bmatrix}$$

(%o10) 
$$\begin{bmatrix} 0.06944 & 0.08333 & -0.06944 \\ -1.125 & -1.25 & 1.125 \\ 0.5 & -1.0 & 0.5 \end{bmatrix}$$


```

Список коэффициентов характеристического полинома: = [1,-15.0,-18.0,0.0]
 (%o17) [x=-1.11684,x=16.1168,x=0] - корни полинома

Рис.3.10. Листинг программы для пакета Maxima и результаты ее выполнения.

В заключении приведем результаты (см. таблицу 3.2) вычислительного эксперимента, целью которого была сравнительная косвенная оценка точности определения собственных значений случайной матрицы для методов Крылова и Данилевского в зависимости от размерности матрицы. Они оказались вполне ожидаемыми, поскольку метод Крылова требует почти в 5 раз больше операций, чем метод Данилевского. Кроме того, в случае метода Крылова свои ошибки добавляет метод исключения, который используется при рассчете коэффициентов характеристического полинома матрицы.

Таблица 3.2. Косвенная оценка погрешности методов $\left(\sum_{i=1}^n A_{ii} - \sum_{i=1}^n \lambda(A)_i \right)$ для

случайных матриц, компонентами которых являлись псевдослучайные равномерно распределенные числа из интервала [-0,5; 0,5].

Размерность матрицы	3	5	7	10	15
Метод Данилевского	0	0	0	0	0
Метод Крылова	0	≤ 0.0000 01	$<0,00000$ 14	$<0,00000$ 6	$<0,00000$ 7

3.6. Определение собственных векторов в общем случае

Способ 1.

Собственный вектор для известного собственного значения λ должен удовлетворять системе уравнений

Определитель этой системы равен нулю, т.е. уравнения линейно зависимы. Поэтому собственный вектор x можно найти следующим образом:

1. Поскольку собственный вектор x задается с точностью до множителя, то положим одну из компонент вектора x равной единице, например, $x_1 = 1$.
 2. Отбросим одно из уравнений, так чтобы оставшиеся уравнения были линейно независимыми, например, последнее.
 3. Далее решаем получившуюся неоднородную систему уравнений любым известным методом

и находим оставшиеся компоненты собственного вектора.

Если решение системы невозможно, то следует сменить номер фиксируемого компонента или отбросить другое уравнение в исходной системе. Также следует поступить при нахождении другого собственного вектора, если собственное значение кратно. Поскольку не всегда произвольная матрица имеет полную систему собственных векторов, то после нахождения нескольких вариантов собственных векторов необходимо проверить их линейную независимость, например, с помощью процедуры ортогонализации.

Способ 2.

Поскольку собственные значения $\tilde{\lambda}_i$, как правило, (например, из-за ошибок округления), определяются с погрешностью, то определитель матрицы $(A - \tilde{\lambda}_i E)$ всегда будет отличен от нуля. С учетом этого обстоятельства для нахождения соответствующего собственного вектора предложен метод обратной итерации: берется произвольный вектор $x^{(0)} = \frac{x^{(0)}}{\|x^{(0)}\|}$ и выполняется итерационный процесс

для $k = 1, K$, $\{ (A - \tilde{\lambda}_i E)x^{(k)} = x^{(k-1)}, x^{(k)} = \frac{x^{(k)}}{\|x^{(k)}\|} \}.$

Для его сходимости обычно бывает достаточно 2-3 итераций.

Сходимость метода обосновывается следующим образом. Пусть матрица A имеет полную линейно-независимую систему собственных векторов $x^{(s)}$. Тогда вектора $x^{(k)}, x^{(k-1)}$ можно представить в виде разложений

$$x^{(k)} = \sum_s a_s^{(k)} x^{(s)}, \quad x^{(k-1)} = \sum_s a_s^{(k-1)} x^{(s)}$$

Подставив эти разложения в систему, получим

$$(A - \tilde{\lambda}_i E)x^{(k)} = x^{(k-1)} \Rightarrow \sum_s [a_s^{(k)}(\lambda_s - \tilde{\lambda}_i) - a_s^{(k-1)}]x^{(s)} = 0 \quad (3.39)$$

Отсюда, поскольку вектора $x^{(s)}$ линейно-независимые, следуют соотношения

$$a_s^{(k)} = \frac{a_s^{(k-1)}}{(\lambda_s - \tilde{\lambda}_i)}, \quad s = 1, n$$

Ho, T.K. $\tilde{\lambda}_i \approx \lambda_i$, to $a_i^{(k)} \gg a_s^{(k)}$, $s \neq i$.

3.7. Метод вращения

Любая симметричная действительная матрица может быть приведена к диагональному виду с помощью подобного ортогонального преобразования

$$D = T^{-1}AT \quad (3.40)$$

где T - ортогональная матрица, удовлетворяющая условию $T^{-1} = T^T$.

Если такая матрица T найдена, то диагональные компоненты матрицы D совпадают с собственными значениями матрицы A , а столбцы матрицы T - с ее соответствующими собственными векторами. Действительно, пусть $\lambda_k = D_{kk}$. Тогда, очевидно,

$$De^{(k)} = \lambda_k e^{(k)}, \quad e^{(k)} = \left\{ e_i^{(k)} = \delta_{ik}, i=1, n \right\}$$

Отсюда следует, что

$$T^{-1}ATE^{(k)} = \lambda_k e^{(k)} \Rightarrow ATE^{(k)} = \lambda_k Te^{(k)} \quad (3.41)$$

т.е. λ_k - собственное значение матрицы A , а $Te^{(k)} = \{T_{ik}, i=1,n\}$ - соответствующий ему собственный вектор.

В методе вращения матрица T находится с помощью серии подобных ортогональных преобразований, которые последовательно уменьшают сумму квадратов недиагональных элементов исходной матрицы A . Указанные преобразования выполняются с использованием матриц вращения (отличается от единичной матрицы только четырьмя компонентами $T_{11}, T_{kk}, T_{lk}, T_{kl}$)

$$T(l,k,\varphi) = \begin{bmatrix} l & & & & k & & & & \\ & 1 & 0 & 0 & - & 0 & 0 & 0 & 0 \\ & 0 & \cos\varphi & 0 & - & 0 & -\sin\varphi & 0 & 0 \\ & 0 & 0 & 1 & - & 0 & 0 & 0 & 0 \\ & - & - & - & - & - & - & - & - \\ & 0 & 0 & 0 & - & 1 & 0 & 0 & 0 \\ & 0 & \sin\varphi & 0 & - & 0 & \cos\varphi & 0 & 0 \\ & 0 & 0 & 0 & - & 0 & 0 & 1 & 0 \\ & 0 & 0 & 0 & - & 0 & 0 & 0 & 1 \end{bmatrix} \dots l \dots k$$

(3.42)

Матрица $T(l, k, \varphi)$ называется матрицей вращения, поскольку с помощью такой матрицы осуществляется пересчет проекций произвольного вектора при повороте системы координат на угол φ (см. рис. 3.11)

$$x_1 = x \cos \varphi - y \sin \varphi$$

$$y_1 = x \sin \varphi + y \cos \varphi$$

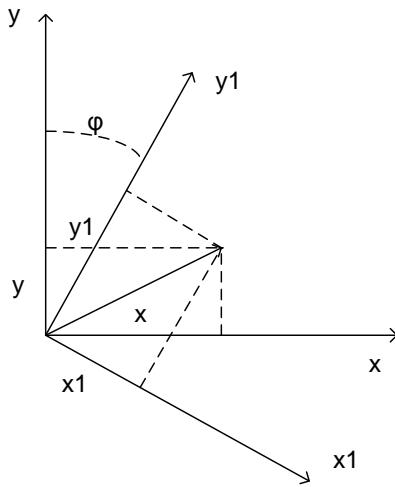


Рис. 3.11. Изменение проекций произвольного вектора x при повороте системы координат на угол φ

Легко проверить, что для матрицы вращения выполняется условие ортогональности, например,

$$T^T T = \begin{bmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{bmatrix} \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Распишем формулы, по которым текущие матрицы A и T (T первоначально задается в виде единичной матрицы) пересчитываются в процессе одного преобразования.

1. Первая фаза преобразования

$$\tilde{A} = AT(l, k, \varphi) = \left[\begin{array}{ccccc} A_{11} & - & - & - & A_{1n} \\ - & A_{il} & - & A_{ik} & - \\ - & - & - & - & - \\ - & - & - & - & - \\ A_{n1} & - & - & - & A_{nn} \end{array} \right] \left[\begin{array}{ccccc} 1 & - & - & - & 0 \\ - & \cos\varphi & - & -\sin\varphi & - \\ - & - & 1 & - & 0 \\ - & \sin\varphi & - & \cos\varphi & - \\ 0 & - & - & - & 1 \end{array} \right] \begin{array}{c} l \\ k \end{array}$$

и аналогично $\tilde{T} = TT(l, k, \varphi)$ меняет компоненты текущих матриц A и T только в l -ом и k -ом столбцах ($i = 1, n$)

$$\begin{aligned} \tilde{A}_{il} &= A_{il} \cos\varphi + A_{ik} \sin\varphi \\ \tilde{A}_{ik} &= -A_{il} \sin\varphi + A_{ik} \cos\varphi \end{aligned} \quad (3.43)$$

$$\begin{aligned} \tilde{T}_{il} &= T_{il} \cos\varphi + T_{ik} \sin\varphi \\ \tilde{T}_{ik} &= -T_{il} \sin\varphi + T_{ik} \cos\varphi \end{aligned} \quad (3.44)$$

2.. Вторая фаза преобразования

$$\tilde{\tilde{A}} = T^T(l, k, \varphi) \tilde{A} =$$

$$\begin{bmatrix} 1 & - & - & - & 0 \\ - & \cos\varphi & - & \sin\varphi & - \\ - & - & 1 & - & - \\ - & -\sin\varphi & - & \cos\varphi & - \\ 0 & - & - & - & 1 \end{bmatrix} \begin{bmatrix} \tilde{A}_{11} & - & - & - & \tilde{A}_{1n} \\ - & \tilde{A}_{lj} & - & - & - \\ - & - & - & - & - \\ - & \tilde{A}_{kj} & - & - & - \\ \tilde{A}_{n1} & - & - & - & \tilde{A}_{nn} \end{bmatrix} \dots$$

l *k*

меняет компоненты текущей матрицы \tilde{A} только в l -ой и k -ой строках ($j = 1, n$)

$$\begin{aligned} \tilde{\tilde{A}}_{lj} &= \tilde{A}_{lj} \cos\varphi + \tilde{A}_{kj} \sin\varphi \\ \tilde{\tilde{A}}_{kj} &= -\tilde{A}_{lj} \sin\varphi + \tilde{A}_{kj} \cos\varphi \end{aligned} \quad (3.45)$$

С помощью формул (3.43) и (3.45) легко получить следующие соотношения

1. Для $i \neq l, k$

$$\begin{aligned} \tilde{\tilde{A}}_{il}^2 + \tilde{\tilde{A}}_{ik}^2 &= A_{il}^2 \cos^2\varphi + A_{ik}^2 \sin^2\varphi + 2A_{il}A_{ik} \sin\varphi \cos\varphi + \\ &+ A_{il}^2 \sin^2\varphi + A_{ik}^2 \cos^2\varphi = A_{il}^2 + A_{ik}^2 \end{aligned} \quad (3.46)$$

и аналогично для $j \neq l, k$

$$\tilde{\tilde{A}}_{lj}^2 + \tilde{\tilde{A}}_{kj}^2 = A_{lj}^2 + A_{kj}^2 \quad (3.47)$$

2.

$$\begin{aligned} \tilde{\tilde{A}}_{lk} &= \tilde{A}_{lk} \cos\varphi + \tilde{A}_{kk} \sin\varphi = -A_{ll} \sin\varphi \cos\varphi + A_{lk} \cos^2\varphi - A_{kl} \sin^2\varphi + A_{kk} \sin\varphi \cos\varphi = \\ &= A_{lk} (\cos^2\varphi - \sin^2\varphi) + (A_{kk} - A_{ll}) \sin\varphi \cos\varphi = A_{lk} \cos 2\varphi - \frac{1}{2} (A_{ll} - A_{kk}) \sin 2\varphi \end{aligned} \quad (3.48)$$

С учетом формул (3.47) и (3.48) сумму квадратов недиагональных элементов матрицы после текущего подобного преобразования можно представить следующим образом

$$\sum_{i \neq j} \tilde{\tilde{A}}_{ij}^2 = \sum_{i \neq j} A_{ij}^2 - 2A_{lk}^2 + \frac{1}{2} [2A_{lk} \cos 2\varphi - (A_{ll} - A_{kk}) \sin 2\varphi]^2 \quad (3.49)$$

Отсюда легко выводятся условия, обеспечивающие на каждой итерации уменьшение суммы квадратов недиагональных элементов:

1. выбор параметров l и k

$$A_{lk} = \max_{i \neq j} |A_{ij}| \quad (3.50)$$

2. выбор параметра φ

$$\varphi = \frac{1}{2} \operatorname{arctg} \left(\frac{2A_{lk}}{A_{ll} - A_{kk}} \right) \quad (3.51)$$

Если $A_{ll} = A_{kk}$, то $|\varphi| = \frac{\pi}{4}$ и $\sin\varphi = \frac{\sqrt{2}}{2} \operatorname{sign}(A_{lk})$, $\cos\varphi = \frac{\sqrt{2}}{2}$.

Замечание. Для расчета $\sin \varphi$, $\cos \varphi$ можно (но не обязательно) использовать формулы, не требующие вычисления угла φ

$$\sin \varphi = \text{sign}(p) \sqrt{\frac{1}{2} \left(1 - \frac{1}{\sqrt{(1 + p^2)}}\right)}, \quad \cos \varphi = \sqrt{(1 - \sin^2 \varphi)}$$

где $p = \frac{2A_{lk}}{(A_{ll} - A_{kk})}$.

Алгоритм метода вращения схематично можно представить в виде (его программная реализация для пакета Maxima приведена в листинге рис.1.2)

```

for it = 1, Iter
{
    находим  $A_{lk}$ 
    if  $|A_{lk}| < \text{eps}$  then "выход из цикла"
    вычисление  $\sin \varphi, \cos \varphi$ 
    for i = 1, n
    {
        вычисляем по формулам (3.43), (3.44)
    }
    for j = 1, n
    {
        вычисляем по формулам (3.45)
    }
}

```

Здесь $Iter$ - максимальное число итераций – преобразований, eps - допустимое значение максимального по модулю недиагонального элемента. Результаты вычислений по первым формулам в (3.43)-(3.45) на каждом шаге вначале следует сохранять в буферной переменной и лишь затем переносить в соответствующее место исходной матрицы и формируемой матрицы вращения.

Поиск максимального по модулю недиагонального элемента можно выполнить следующим образом

```

max = -1, l = 0, k = 0
for i = 1, (n - 1)
{
    for j = i + 1, n
    {
        buf = |Aij|
        if buf > max then { max = buf, l = i, k = j }
    }
}

```

Метод устойчив к ошибкам округления и обладает асимптотической скоростью сходимости. Действительно, из соотношения

$$\sum_{i \neq j} \tilde{A}_{ij}^2 = \sum_{i \neq j} A_{ij}^2 - 2A_{lk}^2$$

$$\sum_{i \neq j} A_{ij}^2$$

имеем (т.к. $\sum_{i \neq j} A_{ij}^2 \leq n(n-1)A_{lk}^2 \Rightarrow A_{lk}^2 \geq \frac{\sum_{i \neq j} A_{ij}^2}{n(n-1)}$)

$$\sum_{i \neq j} \tilde{A}_{ij}^2 \leq \sum_{i \neq j} A_{ij}^2 [1 - \frac{2}{n(n-1)}] \equiv q \sum_{i \neq j} A_{ij}^2$$

и, следовательно, после m -ой итерации

$$(\sum_{i \neq j} \tilde{A}_{ij}^2)_{текущая} \leq q^m (\sum_{i \neq j} A_{ij}^2)_{исходная} \quad (3.52)$$

Ниже в таблице 3.3 приведены значения параметра $q = [1 - \frac{2}{n(n-1)}]$ для

различных значений размерности n исходной матрицы.

Таблица 3.3. Зависимость параметра q от размерности матрицы n

n	2	3	4	5	10	20
q	0	0,67	0,83	0,9	0,98	0,995

Из таблицы видно, что с ростом n скорость сходимости метода быстро падает.

На рис. 3.12 и 3.13 приведены экраны программы, реализующей метод вращения (на начальной и конечной стадиях). Использована случайная матрица, компонентами которой являются псевдослучайные числа, равномерно распределенные на отрезке $[0,1]$. Видно, что после 90 итераций выполняется инвариант - сумма найденных собственных значений (они расположены на диагонале преобразованной матрицы) равна сумме диагональных элементов исходной матрицы, т.е. следу матрицы.

Кроме того, проведено дополнительное тестирование надежности полученного решения (пункт меню «дополнительное_тестирование»). Его результаты представлены на рис. 3.14 и 3.15.

С помощью этой программы исследована зависимость среднего числа итераций, требуемых для нахождения с заданной точностью собственных значений случайной матрицы, от ее размерности. Результаты исследования представлены в таблице 3.4 и в графическом виде на рис. 3.16.

Нахождение собственных чисел симметричной матрицы методом вращения

дополнительное_тестирование справка выход

Размерность матрицы (>1) <input type="text" value="8"/>								
Симметричная матрица A								
i \ j	1	2	3	4	5	6	7	8
1	.46349734	.65019953	.05731708	.37179208	.50838417	.79605663	.02819306	.48382235
2	.65019953	.34831876	.99817669	.50606138	.04753184	.9152469	.77322233	.00343424
3	.05731708	.99817669	.33788395	.94677848	.22086394	.28380042	.87691236	.4393124
4	.37179208	.50606138	.94677848	.66440475	.43975025	.19934607	.35116166	.5414058
5	.50838417	.04753184	.22086394	.43975025	.22110063	.8682344	.55567831	.54431093
6	.79605663	.9152469	.28380042	.19934607	.8682344	.61302	.74057198	.04437846
7	.02819306	.77322233	.87691236	.35116166	.55567831	.74057198	.0110718	.6641534
8	.48382235	.00343424	.4393124	.5414058	.54431093	.04437846	.6641534	.48292375

Строка, столбец и задаваемое значение компоненты:

Матрицу задать случайным образом Сформировать случайную матрицу

Диапозон случайных чисел
 - от 0 до 1
 - от -0,5 до 0,5

Число итераций для метода вращения Выполнить След матрицы A
 Требуемая точность (max |Aij| < eps) Сумма собст. значений

Суммарная матрица преобразований S - ее столбцы являются собственными векторами соответствующих собственных чисел

i \ j	1	2	3	4	5	6	7	8
1								
2								
3								
4								
5								
6								
7								
8								

Рис. 3.12. Задание исходной случайной матрицы

Нахождение собственных чисел симметричной матрицы методом вращения

дополнительное_тестирование справка выход

Размерность матрицы (>1) 8

Симметричная матрица A

i \ j	1	2	3	4	5	6	7	8
1	.2899461
2	.	3,9154884
3	.	.	-1,0299518
40575187
57828876	.	.	.
6	1,1130271	.	.
7	-1,1303035	.
88244212

Строка, столбец и задаваемое значение компоненты:

Матрицу задать случайным образом

Диапазон случайных чисел
 - от 0 до 1
 - от -0,5 до 0,5

Число итераций для метода вращения 100 След матрицы A 3,1422211

Требуемая точность (max |Aij| < eps) .00000001 Выполнить Сумма собст. значений 3,1422211

Суммарная матрицы преобразований S - ее столбцы являются собственными векторами соответствующих собственных чисел

i \ j	1	2	3	4	5	6	7	8
1	.593833	.3007249	-.0104585	-.3330046	.2612952	.4238448	.3891994	.2157075
2	.1380266	.4025365	-.4233987	-.2749375	.0160219	.0446833	-.5389832	-.5208644
3	-.0599366	.3874648	.57556	-.0406596	.4839755	-.444963	.083347	-.2723693
4	.4834742	.3611881	-.1468623	.5188552	-.3583394	-.4415507	.0742964	.1269744
5	-.2673407	.3036744	-.228176	.4715384	.5234272	.233881	-.2607979	.4064556
6	-.1845932	.4052068	.4661507	.2135833	-.4609896	.5482549	-.0372835	-.1559923
7	-.5132314	.3680748	-.4217684	-.1423265	-.118766	-.1110812	.6087992	-.0767338
8	-.1483885	.2739588	.1433073	-.5044108	-.2612199	-.2403686	-.3248263	.629631

Рис. 3.13. Собственные значения исходной, случайной матрицы, найденные методом вращения. Видно, что сумма собственных значений равна следу матрицы.

Дополнительное тестирование результатов метода вращения

Выход

Построенная матрица

i \ j	1	2	3	4	5	6	7	8
1	1.
2	.	1.
3	.	.	1.
4	.	.	.	1.
5	1.	.	.	.
6	1.	.	.
7	1.	.
8	1.

Построить $S(t)^*S$ Построить $S(t)^*A^*S$

Рис.3.14. Видно, что суммарная матрица преобразования S , полученная итерационным методом вращения, действительно является ортогональной.

Дополнительное тестирование результатов метода вращения

ВЫХОД

Построенная матрица

i \ j	1	2	3	4	5	6	7	8	
1	.2899461								
2		3.9154884							
3			-1.0299518						
4				.0575187					
5					.7828876				
6						1.1130271			
7							-1.1303035		
8								.8244212	

Построить $S(t)^*S$

Построить $S(t)^*A^*S$

Рис.3.15. Видно, подобное преобразование исходной матрицы, выполненное с помощью суммарной матрицы S , дает те же результаты, что и итерационная процедура метода вращения (см. рис. 3.13).

Таблица 3.4. Экспериментальная зависимость среднего числа требуемых итераций N в методе вращения от размерности n случайной матрицы.

n	2	4	6	8	10	12
N	1	15	46	84	143	211

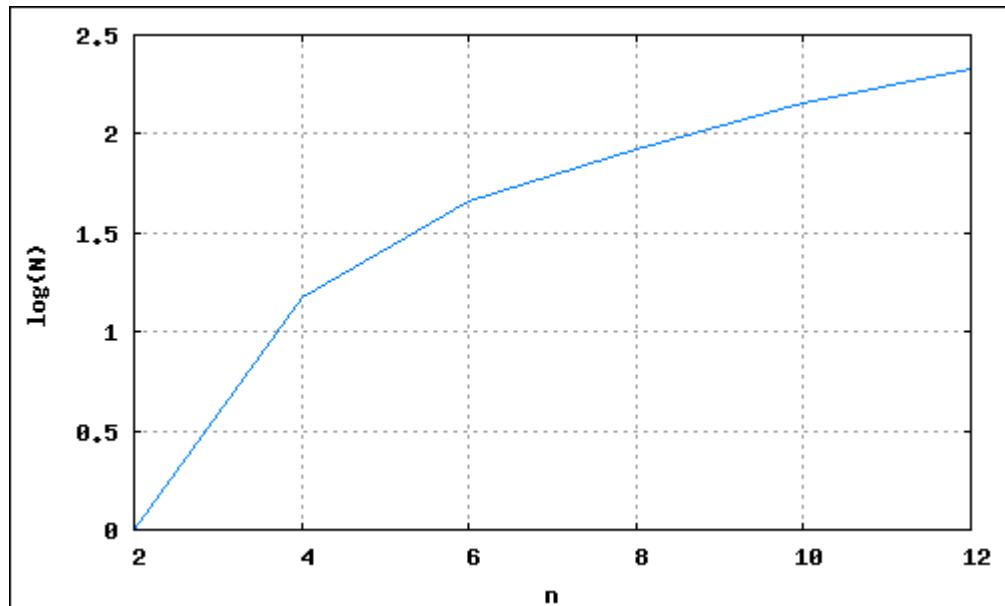


Рис. 3.16. Графическое представление результатов таблицы 3.4. ($\log(N)$ от n).

Вопросы и задания для самоконтроля

1. В чем смысл метода Данилевского?
2. Составьте алгоритм процедуры выбора ведущего элемента для метода Данилевского.
3. О чём сигнализирует нулевой ведущий элемент в методе Данилевского?
4. Каким образом в общем случае можно найти собственный вектор для заданного собственного значения?
5. В чем смысл метода Крылова?
6. В чем смысл метода вращения?
7. Каким образом выбирается текущая матрица вращения в методе вращения?

Глава 4. Алгоритмы для работы с полиномами

4.1. Подсчет значения полинома и деление на множители

1. Подсчет значения полинома $P_n(x)$ в точке x (схема Горнера)

Если полином представить в виде

$$P_n(x) = \sum_{i=0}^n a_i x^{n-i} = a_0 x^n + a_1 x^{n-1} + \dots + a_n = \\ (\dots (a_0 x + a_1) x + a_2) x + \dots + a_{n-1}) x + a_n \quad (4.1)$$

то для подсчета значения полинома $P_n(x)$ в точке x легко записать наиболее экономичный (по количеству операций) алгоритм

$$\begin{aligned} P &= a_0 \\ \text{for } i &= 1, n \{ P = Px + a_i \} \end{aligned} \quad (4.2)$$

Этот алгоритм легко обобщается на комплексный случай $x = \alpha + i\beta$

$$\begin{aligned} P1 &= a_0 \\ P2 &= 0 \\ \text{for } i &= 1, n \\ \{ & \\ R1 &= P1 \\ P1 &= \alpha P1 - \beta P2 + a_i \\ P2 &= \alpha P2 + \beta R1 \\ \} & \end{aligned} \quad (4.3)$$

Здесь использовано представление $P(\alpha + i\beta) = P1 + iP2$.

Ниже показано, как этот алгоритм реализуется на Maxima

```
/* п/п вычисления значений полинома p=(p1+i*p2) в точке x=(xd+i*xm) */  
/* c – список коэффициентов полинома: c[1]*x^n+c[2]*x^(n-1)+...+c[n+1] */  
p(c,xd,xm):=block  
(  
    p1:c[1],  
    p2:0,  
    for i:2 thru length(c) do /* длина списка */  
        (r1:p1, p1:xd*p1-xm*p2+c[i],p2:xd*p2+xm*r1)  
);  
  
/* список коэффициентов полинома:  
c[1]*x^5+c[2]*x^4+...+c[3]*x^3+c[4]*x^2+c[5]*x+c[6] */  
c:[1,2,3,4,5,6];
```

```

p(c,1,1);
print("Действительная и мнимая части значения полинома = ",p1," , ",p2);

/* другой способ задания полинома – как собственной функции */
pol(x):=x^5+2*x^4+3*x^3+4*x^2+5*x+6;
/* определение 2-х функций – действительной и мнимой частей полинома*/
fd(x):=realpart(pol(x));fm(x):=imagpart(pol(x));
x:1+%i*1;
print("Действительная и мнимая части значения полинома = ",fd(x)," , ",fm(x));

(%o1) p(c, xd, xm):=block (p1: c_1, p2: 0, for i from 2 thru length(c) do
(r1:p1, p1 : xd p1 - xm p2 + c_i, p2 : xd p2 + xm r1)) block
(p1: c_1, p2: 0, for i from 2
thru length(c) do (r1:p1, p1 : xd p1 - xm p2 + c_i, p2 : xd p2 + xm r1)) (%o2) [1, 2, 3,
(%o3) done
Действительная и мнимая части значения полинома = - 7 , 15
(%o4) 15
(%o5) pol(x):=x^5+2 x^4+3 x^3+4 x^2+5 x+6
(%o6) fd(x):=realpart(pol(x))
(%o7) fm(x):=imagpart(pol(x))
(%o8) %i+1
Действительная и мнимая части значения полинома = - 7 , 15

```

2. Деление полинома $P_n(x)$ на линейный множитель $(x - \alpha)$

Запишем очевидное соотношение между исходным полиномом и результатом его деления на линейный множитель

$$\begin{aligned}
P_n(x) &= \sum_{i=0}^n a_i x^{n-i} = a_0 x^n + a_1 x^{n-1} + \dots + a_n = \\
&= (x - \alpha)(b_0 x^{n-1} + b_1 x^{n-2} + \dots + b_{n-1}) + b_n = \\
&= b_0 x^n + b_1 x^{n-1} + \dots + b_{n-1} x + b_n - b_0 \alpha x^{n-1} - b_1 \alpha x^{n-2} - \dots - b_{n-1} \alpha
\end{aligned} \tag{4.4}$$

Отсюда, сравнивая сомножители при одинаковых степенях x в крайних частях равенства, получим рекуррентные формулы для определения коэффициентов b_i

$$\begin{aligned}
b_0 &= a_0 \\
b_1 &= a_1 + b_0 \alpha \\
b_2 &= a_2 + b_1 \alpha \\
&\dots \\
b_n &= a_n + b_{n-1} \alpha = P_n(\alpha) - \text{остаток}
\end{aligned} \tag{4.5}$$

Остаток – коэффициент b_n равен нулю, если α – корень полинома (см. (4.2)).

Приведем алгоритм расчета коэффициентов b_i , который в общих чертах совпадает с алгоритмом (4.2)

$$b_0 = a_0 \quad (4.6)$$

for $i = 1, n \{ b_i = b_{i-1} \alpha + a_i \}$

3. Деление полинома $P_n(x)$ на квадратичный множитель $(x^2 + px + q)$

Аналогично предыдущему случаю имеет

$$\begin{aligned} P_n(x) &= \sum_{i=0}^n a_i x^{n-i} = a_0 x^n + a_1 x^{n-1} + \dots + a_{n-1} x + a_n = \\ &= (x^2 + px + q)(b_0 x^{n-2} + b_1 x^{n-3} + b_2 x^{n-4} + \dots + b_{n-3} x + b_{n-2}) + b_{n-1} x + b_n = \\ &= b_0 x^n + b_1 x^{n-1} + b_2 x^{n-2} + \dots + b_{n-3} x^3 + b_{n-2} x^2 + \\ &\quad b_0 p x^{n-1} + b_1 p x^{n-2} + b_2 p x^{n-3} + \dots + b_{n-3} p x^2 + b_{n-2} p x + \\ &\quad b_0 q x^{n-2} + b_1 q x^{n-3} + b_2 q x^{n-4} + \dots + b_{n-3} q x + b_{n-2} q + b_{n-1} x + b_n \end{aligned} \quad (4.7)$$

Отсюда получаем рекуррентные формулы для определения коэффициентов b_i

$$\begin{aligned} b_0 &= a_0 \\ b_1 &= a_1 - b_0 p \\ b_2 &= a_2 - b_1 p - b_0 q \\ &\dots \\ b_{n-1} &= a_{n-1} - b_{n-2} p - b_{n-3} q \\ b_n &= a_n - b_{n-2} q \end{aligned} \quad (4.8)$$

и алгоритм для их последовательного вычисления

$$\begin{aligned} b_0 &= a_0 \\ b_1 &= a_1 - b_0 p \\ &\dots \\ &\text{for } i = 2, (n-1) \{ b_i = a_i - b_{i-1} p - b_{i-2} q \} \\ b_n &= a_n - b_{n-2} q \end{aligned} \quad (4.9)$$

Если $b_{n-1} = b_n = 0$, то полином $P_n(x)$ делится на квадратичный множитель $(x^2 + px + q)$ нацело.

Замечание. При выделении из полинома пары комплексно сопряженных корней $\alpha \pm i\beta$ коэффициенты соответствующего квадратичного множителя равны $p = -2\alpha$, $q = (\alpha^2 + \beta^2)$.

4.2. Локализация корней полинома

Приведем некоторые сведения, касающиеся корней полинома $P_n(x)$ с действительными коэффициентами:

- все комплексные корни полинома попарно комплексно сопряженные;
- полином нечетной степени имеет, по крайней мере, один действительный корень.

Укажем несколько теорем, позволяющих грубо оценивать местоположение корней на комплексной плоскости:

- все корни полинома лежат внутри круга радиуса $R = 1 + \frac{A}{|a_0|}$, где $A = \max_{i=1,n} |a_i|$;
- все корни полинома лежат за пределами круга радиуса $r = \frac{1}{1 + \frac{B}{|a_n|}}$, где $B = \max_{i=0,n-1} |a_i|$ и $a_n \neq 0$ - иначе полином имеет нулевой корень и его порядок можно понизить;
- число корней полинома внутри замкнутой фигуры на комплексной плоскости, в частном случае, круга, по которой пробегает против часовой стрелки аргумент x , равно количеству полных оборотов вокруг начала координат соответствующей замкнутой кривой $P_n(x)$, являющейся отображением этой фигуры.

Опираясь на эти результаты, можно предложить следующую последовательность операций по локализации корней полинома (для выполнения этих операций разработана программы, экранные формы которой с конкретным примером представлены на рис. 4.1-4.8).

1. Находим радиусы r, R окружностей, между которыми на комплексной плоскости расположены все корни заданного полинома (управляющая кнопка «локализация корней» на рис. 4.1).

2. Нажатием кнопки «построить отображение окружности», строим отображение полиномом $P_3(x)$ окружности (радиус - 9, расположение центра окружности - 0,0). Отображение показано на рис. 4.2. Видно, что кривая три раза оборачивается вокруг центра координат. Это подтверждает правильность грубой оценки (см. рис. 4.1.).

Подобным образом, меняя центр и радиус окружности, можно протестировать любую область комплексной плоскости на наличие в ней корней полинома.

3. Локализуем положение действительного корня полинома, построив кривую $P_3(x)$ для аргумента, изменяющегося в интервале $-9 \leq x \leq 9$ (кнопка «построить график $P_n(x)$ ». Построенная кривая приведена на рис. 4.3. Здесь же приведено значение действительного корня, уточненное методом дихотомии, алгоритм которого записывается следующим образом (здесь xb , xe - левая и правая границы интервала, содержащего один корень, eps - точность, $root$ - найденное значение корня)

```

 $pb = P(xb), pe = P(xe), pm = 1000$ 
do while  $|pm| > eps$ 
 $xm = \frac{xb + xe}{2}, pm = P(xm)$ 
if  $sign(pb) = sign(pm)$  then
     $pb = pm, xb = xm$ 
else
     $pe = pm, xe = xm$ 
end if
loop
root = xm

```

В данном алгоритме отрезок последовательно делится пополам и за новый отрезок выбирается та его половина, на концах которой значение полинома имеет разные знаки. Выход из процесса деления происходит, когда значение полинома в средней точке текущего отрезка оказывается меньше заданной точности.

4. После нахождения уточненного значения корня, выделяем его из текущего полинома и тем самым понижаем его порядок. Коэффициенты нового полинома, полученного после выделения действительного корня из предыдущего полинома, приведены в нижней таблице рис. 4.4. Используя кнопку «обновить исходный полином», коэффициенты нового полинома ставим в верхнюю таблицу на место их предыдущих значений (см. рис. 4.5).
5. Новый полином имеет второй порядок, поэтому его корни определяются по известным формулам – кнопка «найти корни квадратичного полинома». Экранная форма со значениями этих корней приведена на рис. 4.6.
6. С помощью кнопки «найти значение полинома» - можно рассчитать значение полинома для найденного корня и убедится в правильности полученных его действительных и мнимых значений.
7. Пункт меню «графическая локализация», расположенный на главной форме, загружает форму, приведенную на рис. 4.7. В рамках этой формы можно методом перебора найти местоположение точек на комплексной плоскости, в которых значение модуля полинома меньше заданного предельного значения: $|P(x + iy)| = |P1 + iP2| = \sqrt{(P1 * P1 + P2 * P2)} < eps$. Это позволяет произвести графическую локализацию местоположения корней полинома на комплексной плоскости. Затем значение каждого локализованного корня можно уточнить, минимизируя выпуклый квадратичный функционал

$$F(x, y) = |P(x + iy)| = |P1 + iP2| = \sqrt{(P1 * P1 + P2 * P2)}$$

методами наискорейшего или покоординатного спуска. Согласно первому методу спуск к нулевому значению функционала осуществляется против направления, заданного ортами вектора градиента функционала в текущей

начальной точке комплексной плоскости. В программе используются два алгоритма наискорейшего спуска. В первом алгоритме одномерный спуск вдоль выбранного направления аналогичен затухающим колебаниям шарика при скатывании на дно «лунки», во втором – при переходе через минимум производится отступ к предыдущей точке, а затем выполняется новый уменьшенный в три раза шаг. При покоординатном спуске в пределах одной итерации производится одномерный спуск вначале по координате x , а затем – y . В этом случае стратегия одномерного спуска – аналогична затухающим колебаниям шарика при скатывании на дно «лунки».

Ниже приведены соответствующие алгоритмы:

1. Алгоритмы метода наискорейшего спуска для двух стратегий одномерного спуска

Стратегия 1

```

 $it = 0$  – число использованных итераций
do while  $it < it \max$ 
     $F \min = F(rx, ry)$ 
     $h = 0,001, x = rx + h, y = ry$ 
     $Fx = F(x, y)$ 
     $x = rx, y = ry + h$ 
     $Fy = F(x, y)$ 
     $\frac{dF}{dx} = \frac{Fx - F \min}{h}, \frac{dF}{dy} = \frac{Fy - F \min}{h}$ 
     $g \equiv |grad(F)| = \sqrt{\left(\frac{dF}{dx}\right)^2 + \left(\frac{dF}{dy}\right)^2}$ 
     $dx = \frac{dF}{dx} / g, dy = \frac{dF}{dy} / g$ 
     $h = 0,01$ 
    do while  $h > 0,00000001$ 
         $rx = rx - h * dx, ry = ry - h * dy$ 
         $Ft = F(rx, ry)$ 
        if  $Ft < F \min$  then  $F \min = Ft$  else  $h = -h / 3$ 
    loop
    if  $F \ min < eps$  then exit do
     $it = it + 1$ 
loop

```

Стратегия 2

$it = 0$ – число использованных итераций

do while $it < it_{\max}$

$$F_{\min} = F(rx, ry)$$

$$h = 0,001$$

$$x = rx + h$$

$$y = ry$$

$$Fx = F(x, y)$$

$$x = rx$$

$$y = ry + h$$

$$Fy = F(x, y)$$

$$\frac{dF}{dx} = \frac{Fx - F_{\min}}{h}$$

$$\frac{dF}{dy} = \frac{Fy - F_{\min}}{h}$$

$$g \equiv |grad(F)| = \sqrt{\left(\frac{dF}{dx}\right)^2 + \left(\frac{dF}{dy}\right)^2}$$

$$dx = \frac{dF}{dx} / g, \quad dy = \frac{dF}{dy} / g$$

$$h = 0,01$$

do while $h > 0,00000001$

$$x = rx - h^* dx$$

$$y = ry - h^* dy$$

$$Ft = F(x, y)$$

if $Ft < F_{\min}$ then $rx = x, ry = y, F_{\min} = Ft$ else $h = h/3$

loop

if $F_{\min} < eps$ then exit do

$$it = it + 1$$

loop

2. Алгоритм метода покоординатного спуска

```
it = 0    —  число использованных итераций
do while it < it max
    for i = 0 to 1
        'определение направления спуска для данной координаты
        h = 0,001
        s = 1
        F min = F(rx, ry)

        a :
            x = rx + h * s * (1 - i)
            y = ry + h * s * i
            Ft = F(x, y)
            if Ft < F min then goto b
            if s = -1 then goto c else s = -1, goto a

        b :
            h = 0,01 * s
            'одномерный спуск вдоль данной координаты
            do while h > 0,00000001
                rx = rx + h * (1 - i)
                ry = ry + h * i
                Ft = F(rx, ry)
                if Ft < F min then F min = Ft else h = -h / 3
            loop

        c :
            if F min < eps then exit do
            it = it + 1
        loop
```

Здесь $it\ max$ - максимальное число допустимых итераций (спусков), h - параметр спуска, eps - требуемая точность.

Выполнение различных операций с полиномами

графическая_локализация метод_парабол справка выход

Размерность полинома (>1)

Коэффициенты полинома: $P(x)=a(0)x^n+a(1)x^{n-1}+\dots+a(n)$

коэффициент	$a(0)$	$a(1)$	$a(2)$	$a(3)$
значение	1	-5	8	-6

Номер коэффициента и его значение:

Значение аргумента $x=x_r+i*x_c$

Значение полинома $P(x)=Pr+i*Pc$

Корни $R(i)$ лежат на комплексной плоскости в пределах окружностей радиусов = $\langle R(i) <$

Радиус окружности, отображаемой полиномом

Центр окружности (X_0, Y_0)

Шаг при движении по окружности (в радианах)

Границы интервала на действительной оси $x_{min}=\langle x=\langle x_{max}$

Величина шага при построении графика

Действительный корень

Рис. 4.1. Главная экранная форма программы, предназначенная для выполнения различных операций с заданным полиномом $P_3(x)$ с целью локализации его корней.

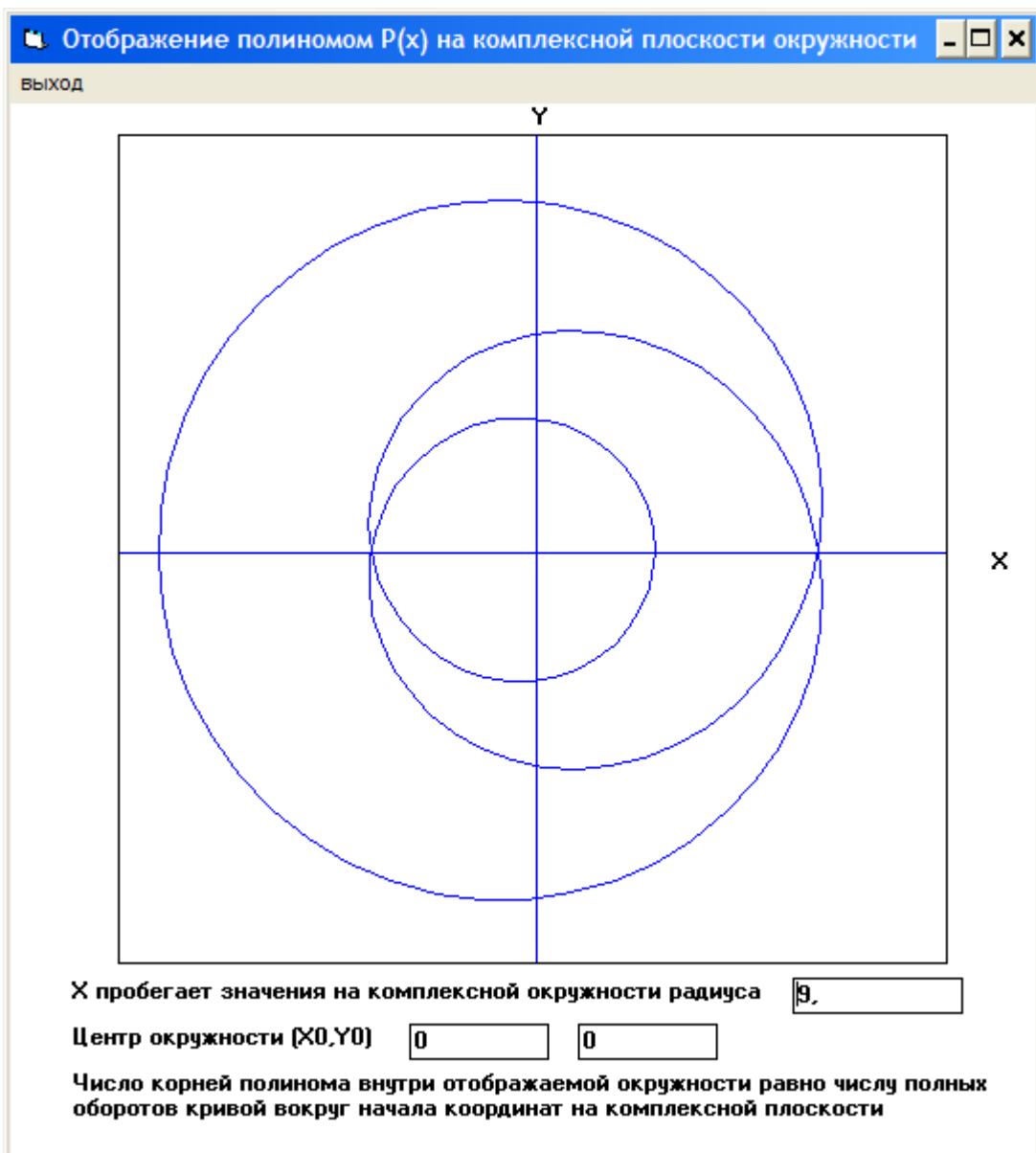


Рис. 4.2. Экранная форма с отображением окружности, построенным согласно заданному полиному $P_3(x)$. Три полных оборота вокруг начала координат, которое делает построенное отображение, указывают на наличие трех корней внутри области комплексной плоскости, ограниченной центральной окружностью радиуса = 9.

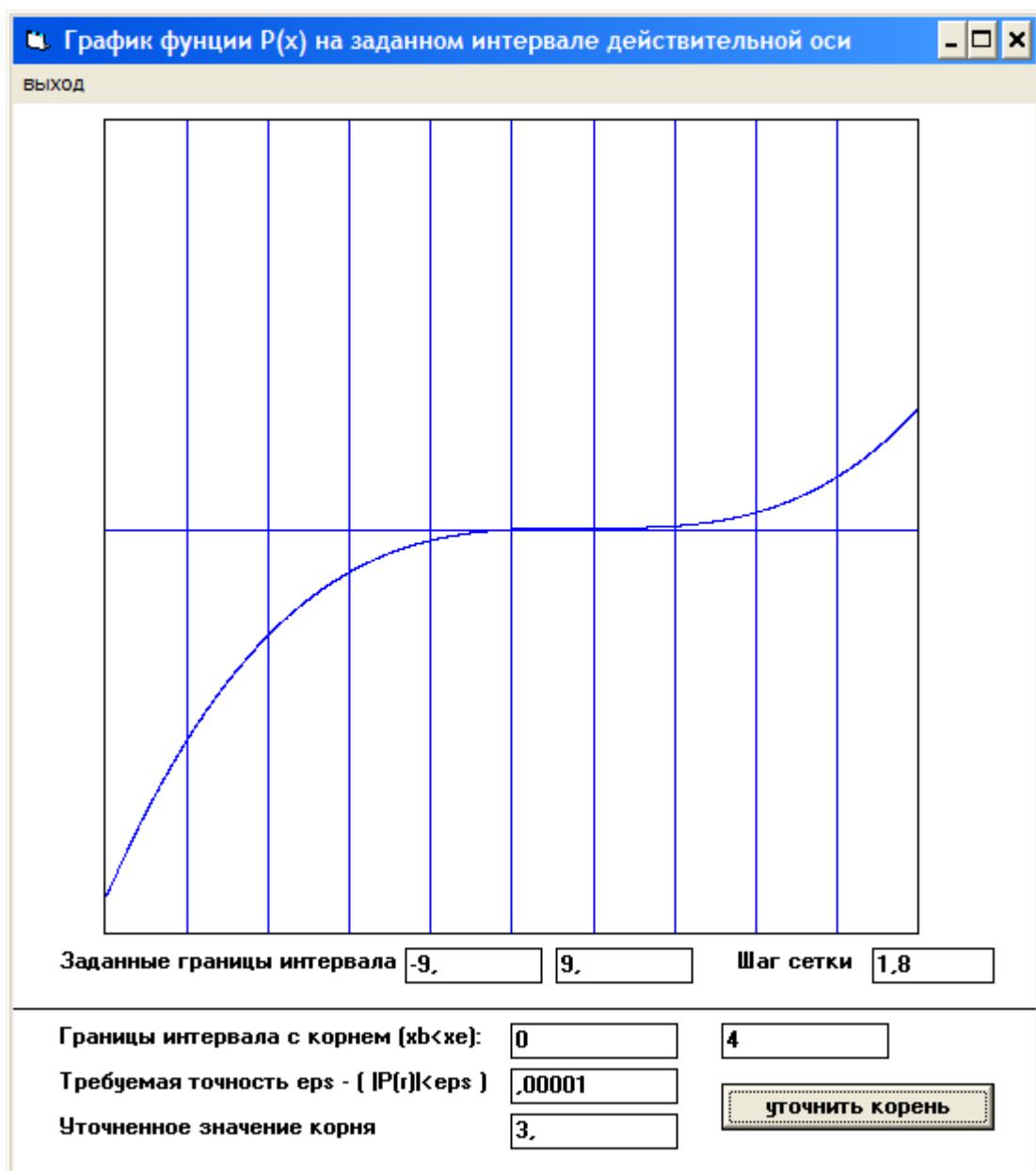


Рис. 4.3. График изменения $P_3(x)$ для заданного интервала. Здесь же приведено значение действительного корня, уточненное методом дихотомии.

Выполнение различных операций с полиномами

графическая_локализация метод_парабол справка выход

Размерность полинома (>1) <input type="text" value="3"/>				
Коэффициенты полинома: $P(x)=a(0)x^n+a(1)x^{n-1}+\dots+a(n)$				
коэффициент	$a(0)$	$a(1)$	$a(2)$	$a(3)$
значение	1	-5	8	-6

Номер коэффициента и его значение:

Значение аргумента $x=xr+i*xc$

Значение полинома $P(x)=Pr+i*Pc$

Корни $R(i)$ лежат на комплексной плоскости в пределах
окружностей радиусов $\leq R(i) \leq$

Радиус окружности, отображаемой полиномом

Центр окружности (X0,Y0)

Шаг при движении по окружности (в радианах)

Границы интервала на действительной оси $x_{min} \leq x \leq x_{max}$

Величина шага при построении графика

Действительный корень

коэффициент	$b(0)$	$b(1)$	$b(2)$	остаток
значение	1,	-2,	2,	0

Рис. 4.4. В нижней таблице приведены коэффициенты нового полинома, полученного после выделения действительного корня из предыдущего полинома.

Выполнение различных операций с полиномами

графическая_локализация метод_парабол справка выход

Размерность полинома (>1)

Коэффициенты полинома: $P(x)=a(0)x^n+a(1)x^{n-1}+\dots+a(n)$

коэффициент	$a(0)$	$a(1)$	$a(2)$
значение	1,	-2,	2,

Номер коэффициента и его значение:

Значение аргумента $x=x_r+i*x_c$

Значение полинома $P(x)=Pr+i*Pc$

Корни $R(i)$ лежат на комплексной плоскости в пределах окружностей радиусов = $< R(i) <$

Радиус окружности, отображаемой полиномом 1,

Центр окружности (X0,Y0) 0

Шаг при движении по окружности (в радианах) ,05

Границы интервала на действительной оси $x_{min} < x < x_{max}$ -1,

Величина шага при построении графика ,05

Действительный корень

коэффициент	
значение	

Рис. 4.5. Произведен перенос коэффициентов нового полинома на место исходного. Порядок анализируемого полинома понижен на единицу. Получена оценка местоположения его корней.

Корни квадратичного полинома

БЫХОД

Действительные и мнимые части корней:

1	<input type="text" value="1,"/> <input type="text" value="1,"/>
2	<input type="text" value="1,"/> <input type="text" value="-1,"/>

Рис. 4.6. Экранная форма с комплексными корнями квадратичного полинома.

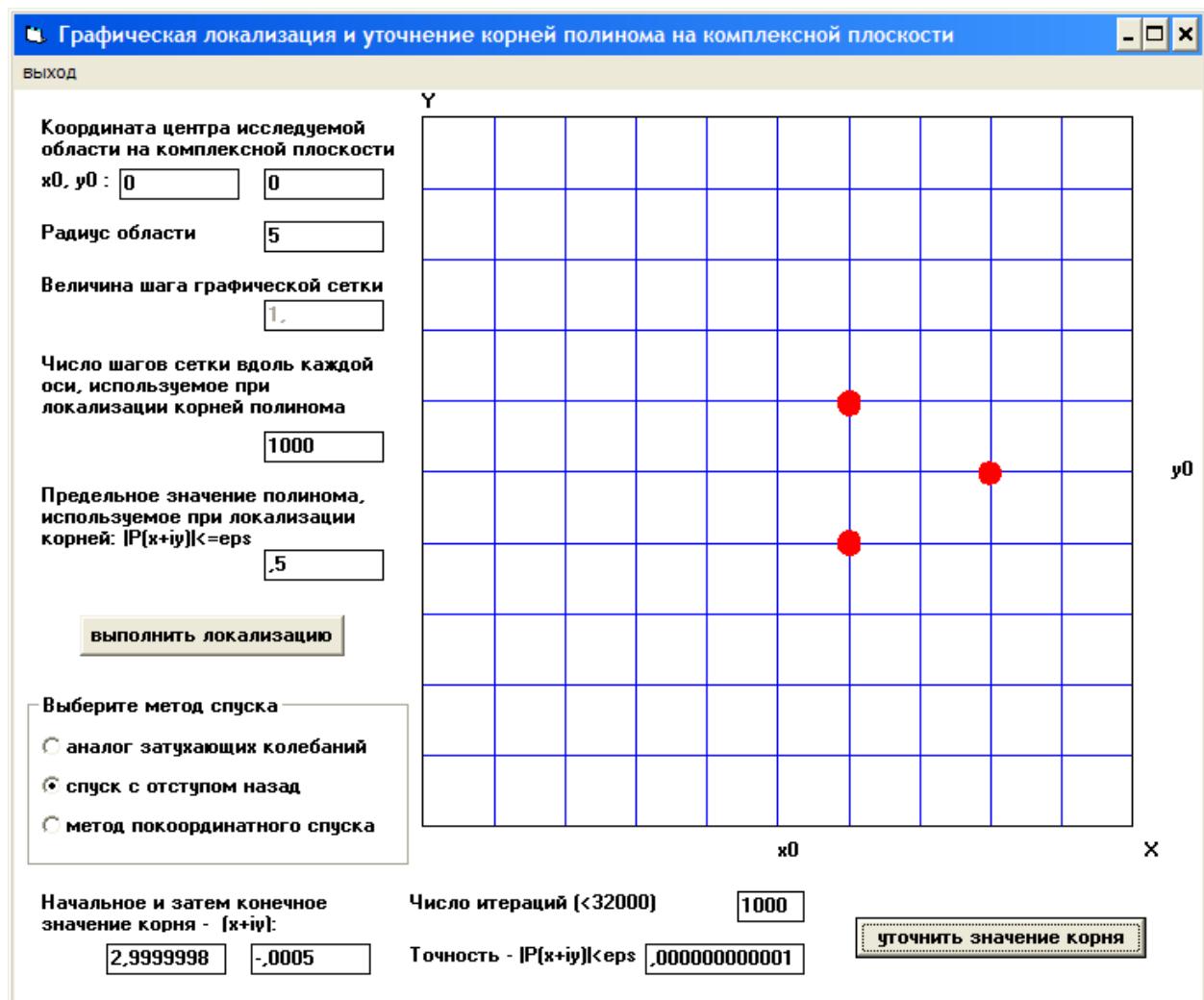


Рис. 4.7. Форма программы для локализации корней на комплексной плоскости и последующего уточнения локализованных корней. Здесь показан результат графической локализации методом перебора всех корней исходного полинома. Затем методом наискорейшего спуска произведено уточнение действительного корня (его точное значение равно 3). При этом использован алгоритм одномерного спуска с отступом назад.

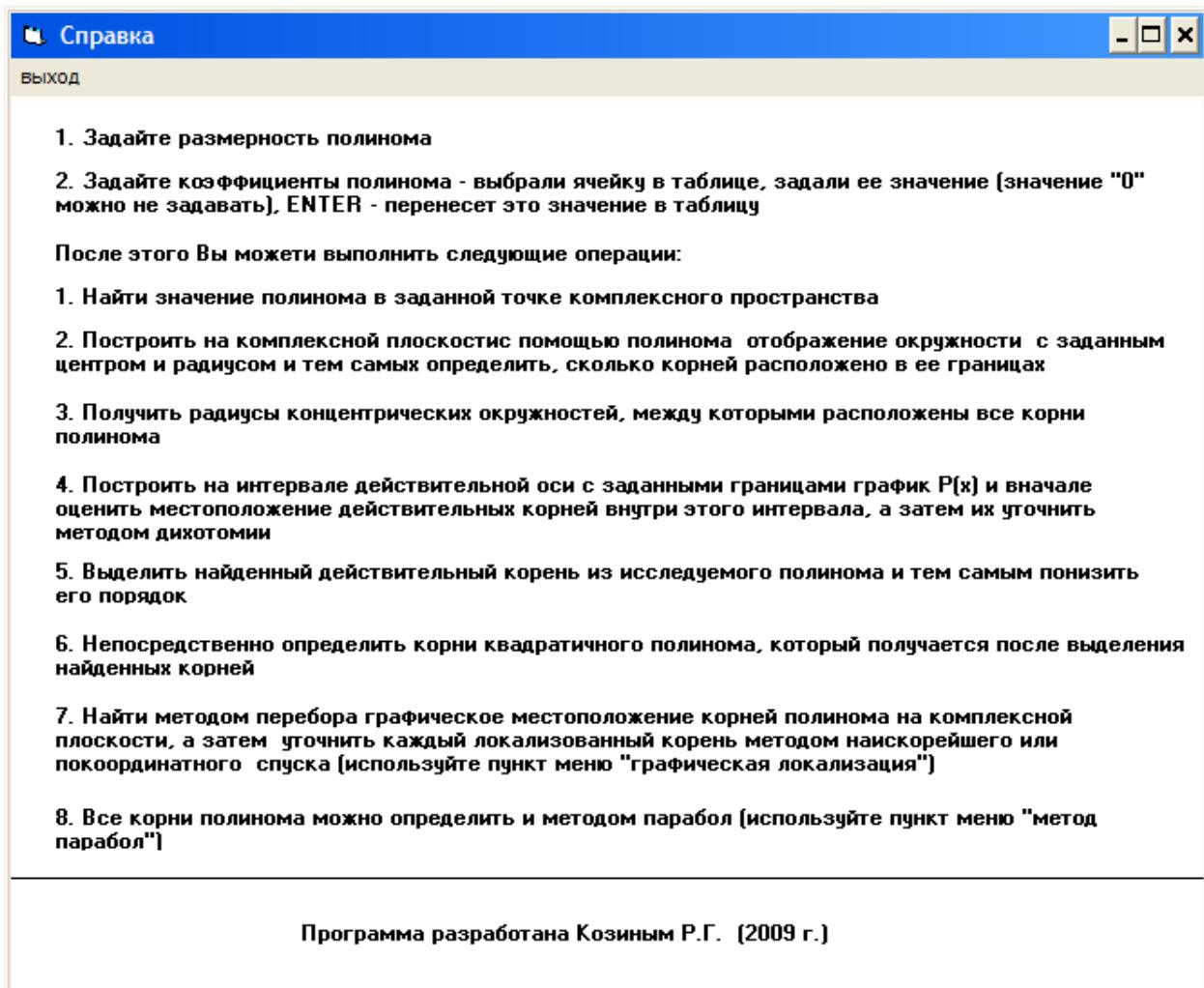


Рис. 4.8. Экранная форма с описанием порядка работы с программой.

Ниже показано как в рамках пакета Maxima решаются некоторые из ранее рассмотренных задач.

```
/* строит отображение, которое дает полином для заданной окружности */
/* здесь: c-список коэффициентов полинома,x0,y0,r-центр окружности и ее
радиус */
/* n-число исследуемых точек на окружности */
otobragnieP(c,x0,y0,r,n):=block
(
dfi:2%pi/n, /* шаг по углу */
lx:[],ly:[], /* инициализируем два списка */
for i:0 thru n do
(
  xt:x0+r*cos(dfi*i),yt:y0+r*sin(dfi*i),
  /* функция введена ранее - вычисляет значение полинома в заданной
точке */
  p(c,xt,yt),
```

```

lx:append(lx,[p1]),ly:append(ly,[p2]) /* добавляем элементы в списки */
),
/* использованием 2-х списков строим в отдельном окне график
отображения*/
plot2d([discrete,lx,ly])
);

numer:true;
fpprintprec:5; /* в выводимом числе 5 цифр, не считая точки */
c:[1,-5,8,6];
/* локализация корней полинома на плоскости */
n:length(c);
c1:makelist(abs(c[i]),i,2,n); /* формирование 2-х вспомогательных
списков */
c2:makelist(abs(c[i]),i,1,n-1);
/* оценка местоположения корней полинома на комплексной плоскости
*/
print(" Максимальный R = ",1+lmax(c1)/abs(c[1]));
print(" Минимальный r = ",1/(1+lmax(c2)/abs(c[n])));
r:9; x0:0; y0:0;
n:960; otobrarenieP(c,x0,y0,r,n); /* строим отображение */;

pol(x):=x^3-5*x^2+8*x-6;
/* проверяем - есть ли у полинома действительные корни на отрезке [-9,9]
*/
/* строим график функции в основном окне */
wxplot2d(pol(t),[t,-9,9],[color,red]);
wxplot2d(pol(t),[t,2.5,3.5],[color,red]); /* уточняем положение
действ.корня */

/* внутренняя функция находит корни полинома */
to_poly_solve([x^3-5*x^2+8*x-5=0], [x]);

```

Результаты выполнения команд и подпрограмм листинга:

*Максимальный R = 9
 Минимальный r = 0.429
 Корни полинома (%o27) %union([t=3],[t=1 - %i],[t=%i +1])*

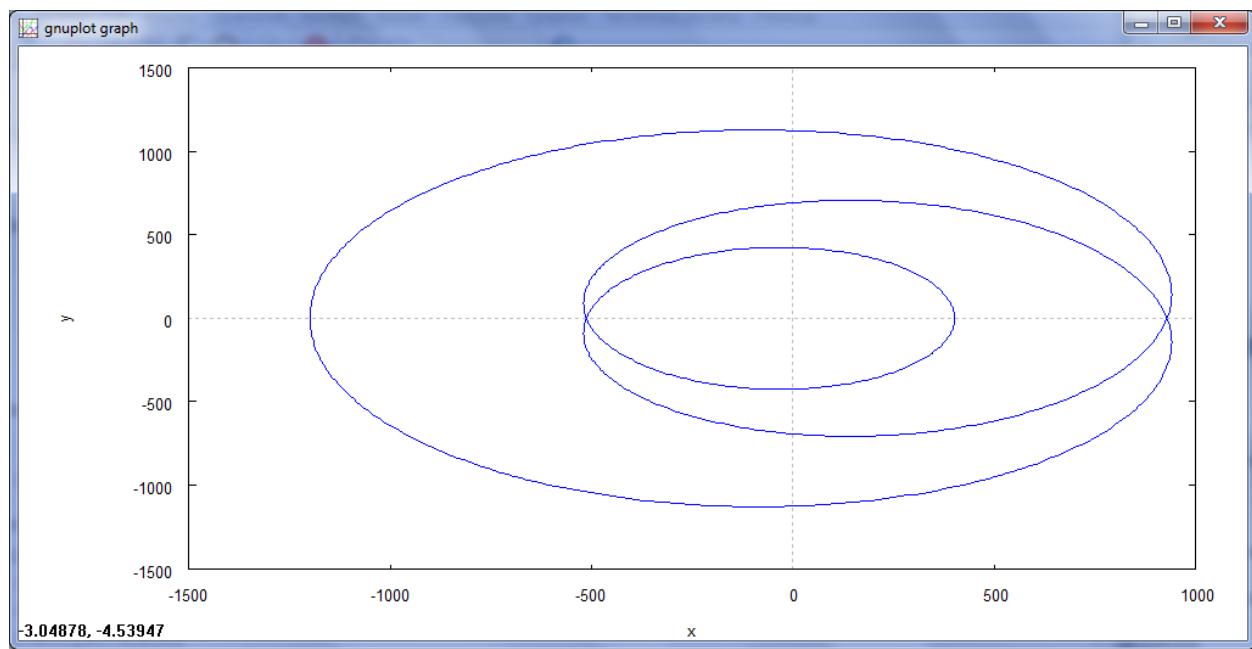


Рис. 4.9. Отображение полиномом $(x^3 - 5x^2 + 8x - 6)$ на комплексной плоскости окружности радиуса 9.

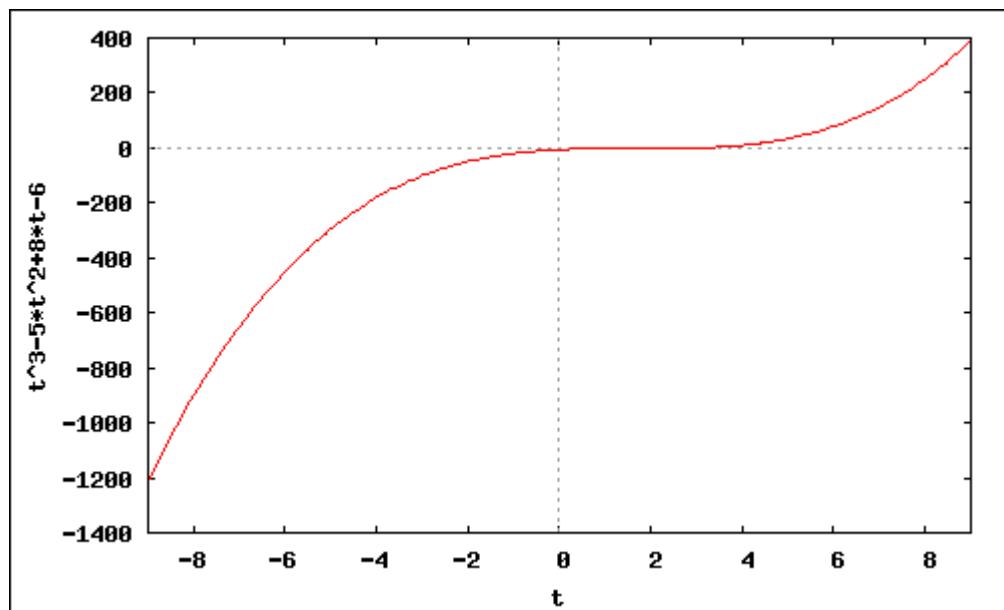


Рис. 4.10. Проверка наличия у полинома действительного корня на отрезке $[-9, 9]$.

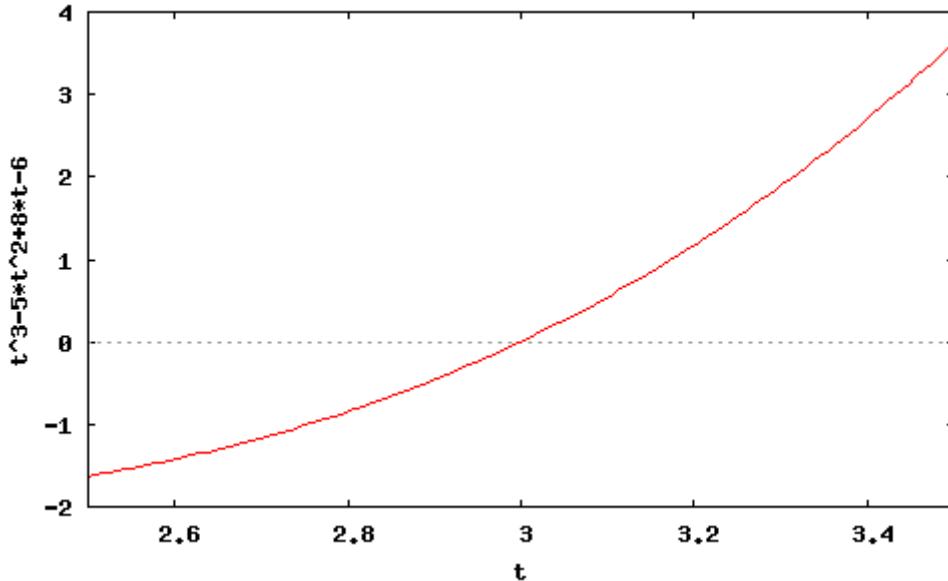


Рис. 4.10. Уточнение положения действительного корня на отрезке [2,5,3,5].

4.3. Метод парабол для нахождения всех корней полинома

Метод парабол является наиболее надежным алгоритмом поиска всех корней полинома. Метод не требует задания приближенных значений корней и определяет их последовательно один за другим. Сходимость метода теоретически не доказана, но не известно ни одного случая, когда бы метод не сходился.

Для запуска метода произвольно выбираются три начальные точки на комплексной плоскости $z = x + iy \equiv z(x, y)$, обычно $z_0 = (-1, 0)$, $z_1 = (1, 0)$, $z_2 = (0, 0)$. По значениям полинома в этих точках $P_n(z_0)$, $P_n(z_1)$, $P_n(z_2)$ (для их расчета используется схема (4.3)) строится интерполяционный многочлен Лагранжа второго порядка.

$$\begin{aligned} L(z) &= P_n(z_0) \frac{(z - z_1)(z - z_2)}{(z_0 - z_1)(z_0 - z_2)} + P_n(z_1) \frac{(z - z_0)(z - z_2)}{(z_1 - z_0)(z_1 - z_2)} + P_n(z_2) \frac{(z - z_0)(z - z_1)}{(z_2 - z_0)(z_2 - z_1)} = \\ &= L_0(z - z_1)(z - z_2) + L_1(z - z_0)(z - z_2) + L_2(z - z_0)(z - z_1) = Az^2 - Bz + C, \end{aligned} \quad (4.10)$$

где

$$\begin{aligned} A &= L_0 + L_1 + L_2, \quad B = L_0(z_1 + z_2) + L_1(z_0 + z_2) + L_2(z_0 + z_1), \\ C &= L_0z_1z_2 + L_1z_0z_2 + L_2z_0z_1, \\ L_0 &= \frac{P_n(z_0)}{(z_0 - z_1)(z_0 - z_2)}, \quad L_1 = \frac{P_n(z_1)}{(z_1 - z_0)(z_1 - z_2)}, \quad L_2 = \frac{P_n(z_2)}{(z_2 - z_0)(z_2 - z_1)} \end{aligned}$$

Находятся два корня этого многочлена

$$z_{1,2} = \frac{B \pm \sqrt{B^2 - 4AC}}{2A}$$

и за новую точку z_2 берется тот из них, который расположен ближе к предыдущей точке z_2 . При этом первая точка предыдущей тройки отбрасывается. По новым точкам тем же порядком строится очередная новая точка и т.д. Построенная таким образом последовательность точек всегда сходится к некоторому корню.

Процесс повторяется до тех пор, пока значение полинома для новой точки z_2 (или расстояние между старой и новой точками z_2) не будет меньше заданной малой величины $P(z_2) < \text{eps}$.

После того, как корень найден, порядок полинома понижается с использованием схем (4.6) или (4.9), соответственно, для действительного корня (мнимая часть корня равна нулю) или для случая комплексно сопряженных корней (мнимая часть корня отлична от нуля). Затем поиск корня продолжается уже для полинома меньшего порядка.

Это повторяется пока текущая размерность полинома больше двух. В противном случае, корни полинома второго или первого порядка находятся просто по известным формулам.

Замечание. Все переменные (кроме коэффициентов полинома) в алгоритме должны быть описаны как комплексные. Соответственно и все операции над ними должны учитывать их природу.

На рис. 4.11. приведены корни полинома, вычисленные методом парабол в рамках программы, представленной в предыдущем параграфе. При реализации алгоритма для комплексных чисел был определен новый тип переменной – комплексное число, состоящее из действительной и комплексной частей. Все операции с комплексными числами, используемые в алгоритме, такие как

$$(a + ib) \pm (c + id) = (a \pm c) + i(b \pm d)$$

$$(a + ib)(c + id) = (ac - bd) + i(ad + cb)$$

$$\frac{(a + ib)}{(c + id)} = \frac{(ac + bd) + i(cb - ad)}{(c^2 + d^2)}$$

$$\sqrt{a \pm ib} = \pm \left[\sqrt{\frac{r+a}{2}} \pm i \sqrt{\frac{r-a}{2}} \right], r = \sqrt{a^2 + b^2}$$

были оформлены в виде соответствующих подпрограмм.

Листинги на языке Visual Basic этих подпрограмм и процедуры, реализующей метод парабол, приведены ниже:

‘складывает два комплексных числа’

```
Sub addc (a As complex, b As complex, c As complex)
    a.r = b.r + c.r
    a.c = b.c + c.c
End Sub
```

‘вычитание для двух комплексных чисел’

```
Sub subc (a As complex, b As complex, c As complex)
```

a.r = b.r - c.r

a.c = b.c - c.c

End Sub

'умножение двух комплексных числа

Sub mulc (aa As complex, b As complex, c As complex)

aa.r = b.r * c.r - b.c * c.c

aa.c = b.r * c.c + b.c * c.r

End Sub

'деление для двух комплексных чисел

Sub divc (a As complex, b As complex, c As complex)

Dim d#

d = c.r * c.r + c.c * c.c

If d = 0# Then

 a.r = 0

 a.c = 0

 Exit Sub

End If

 a.r = (b.r * c.r + b.c * c.c) / d

 a.c = (c.r * b.c - b.r * c.c) / d

End Sub

'корень квадратный из комплексного числа

Sub sqrtc (a As complex, b As complex)

Dim r#

r = Sqr(b.r * b.r + b.c * b.c)

a.r = Sqr((r + b.r) / 2)

a.c = Sqr((r - b.r) / 2)

If b.c < 0 Then

 a.c = -a.c

End If

End Sub

Sub muller ()

ReDim zn(3)

ReDim l(3)

Dim dz#, dz1#, dz2#, normp#

Dim it%, txt\$

Dim z0 As complex

Dim z1 As complex

Dim z2 As complex

Dim z3 As complex

Dim z4 As complex

```

Dim z5 As complex
Dim aa As complex
Dim bb As complex
Dim cc As complex
Dim dd As complex
Dim l0 As complex
Dim l1 As complex
Dim l2 As complex
nr = 0      'число найденных корней
Do While n > 2
    'наличие нулевого корня
    If Abs(a(n)) < .0001 Then
        root(nr).r = 0
        root(nr).c = 0
        nr = nr + 1
        n = n - 1
        GoTo Lab
    End If
    '
    zn(0).r = -1
    zn(0).c = 0
    zn(1).r = 1
    zn(1).c = 0
    zn(2).r = 0      '
    zn(2).c = 1      '0
    For i = 0 To 2      'значения полинома в трех точках
        Call polinomc(zn(i))
        l(i).r = p.r
        l(i).c = p.c
    Next i
    normp = Sqr(l(2).r * l(2).r + l(2).c * l(2).c)
    it = 0
    Do While normp > eps
        '
        Call subc(z0, zn(0), zn(2))  'zn(0)-zn(2) -> z0
        Call subc(z1, zn(0), zn(1))  'zn(0)-zn(1) -> z1
        Call mulc(z3, z0, z1)       'z0*z1 -> z3
        Call divc(l0, l(0), z3)     'l(0)/z3 -> l0  L0

        Call subc(z0, zn(1), zn(0))  'zn(1)-zn(0) -> z0
        Call subc(z1, zn(1), zn(2))  'zn(1)-zn(2) -> z1
        Call mulc(z3, z0, z1)       'z0*z1 -> z3
        Call divc(l1, l(1), z3)     'l(1)/z3 -> l1  L1

```

```

Call subc(z0, zn(2), zn(1))    'zn(2)-zn(1) -> z0
Call subc(z1, zn(2), zn(0))    'zn(2)-zn(0) -> z1
Call mulc(z3, z0, z1)          'z0*z1 -> z3
Call divc(l2, l(2), z3)        'l(2)/z3 -> l2  L2
'
Call addc(z0, l0, l1)
Call addc(aa, z0, l2)          'A  aa=l0+l1+l2
'
Call addc(z3, zn(2), zn(1))    'zn(2)+zn(1) -> z3
Call mulc(z0, l0, z3)          'l0*z3 -> z0  L0*(z(2)+z(1))
Call addc(z3, zn(0), zn(2))    'zn(0)+zn(2) -> z3
Call mulc(z1, l1, z3)          'l1*z3 -> z1  L1*(z(0)+z(2))
Call addc(z3, zn(1), zn(0))    'zn(1)+zn(0) -> z3
Call mulc(z2, l2, z3)          'l2*z3 -> z2  L2*(z(1)+z(0))
Call addc(z3, z0, z1)
Call addc(bb, z3, z2)          'B  bb=z0+z1+z2
'
Call mulc(z3, zn(2), zn(1))    'zn(2)*zn(1) -> z3
Call mulc(z0, l0, z3)          'L0*z3 -> z0
Call mulc(z3, zn(2), zn(0))    'zn(2)*zn(0) -> z3
Call mulc(z1, l1, z3)          'L1*z3 -> z1
Call mulc(z3, zn(0), zn(1))    'zn(0)*zn(1) -> z3
Call mulc(z2, l2, z3)          'L2*z3 -> z2
Call addc(z3, z0, z1)
Call addc(cc, z3, z2)          'C  cc=z0+z1+z2
'
Call mulc(z3, bb, bb)          'B*B -> z3
Call mulc(z4, aa, cc)          'A*C -> z4
z4.r = 4 * z4.r
z4.c = 4 * z4.c               '4*A*C -> z4
Call subc(dd, z3, z4)          'D  dd=z3-z4
'
Call sqrtc(z3, dd)             'sqr(dd) -> z3
aa.r = 2 * aa.r
aa.c = 2 * aa.c               '2A
Call addc(z4, bb, z3)
Call subc(z5, bb, z3)
Call divc(z0, z4, aa)          'z0  два корня
Call divc(z1, z5, aa)          'z1

```

'находим ближайший корень

```

Call subc(z3, zn(2), z0)      'zn(2)-z0 -> z3
Call subc(z4, zn(2), z1)      'zn(2)-z1 -> z4
dz1 = Sqr(z3.r * z3.r + z3.c * z3.c)

```

```

dz2 = Sqr(z4.r * z4.r + z4.c * z4.c)
zn(0) = zn(1)
zn(1) = zn(2)
l(0) = l(1)
l(1) = l(2)
If dz1 < dz2 Then
    dz = dz1
    zn(2) = z0
Else
    dz = dz2
    zn(2) = z1
End If
it = it + 1
If it > k Then
    MsgBox "При поиске корня исчерпаны все итерации !", 48, "Сообщение"
    Exit Do    '? do or sub
End If
Call polinomc(zn(2))
l(2).r = p.r      'значение полинома в третей точке
l(2).c = p.c
normp = Sqr(l(2).r * l(2).r + l(2).c * l(2).c)
Loop
root(nr) = zn(2)
nr = nr + 1
'
If Abs(zn(2).c) < .0001 Then
    Call extr_root1(zn(2).r)
Else
    Call extr_root2(zn(2))
    root(nr) = zn(2)      'сохраняем второй комплексно-сопряженный корень
    root(nr).c = -root(nr).c
    nr = nr + 1
End If
Lab:
Loop
'корни линейного или квадратичного полинома
If n = 1 Then
    root(nr).r = -a(1) / a(0)
    root(nr).c = 0#
    nr = nr + 1
Else
    Call fine_root2
    root(nr).r = rr1
    root(nr).c = rc1

```

```

nr = nr + 1
root(nr).r = rr2
root(nr).c = rc2
nr = nr + 1
End If
End Sub

```

‘вычисление значения полинома в точке x

```
Sub polinomc (x As complex)
```

```
    Dim i%, r1 As Double
```

```
    p.r = a(0)
```

```
    p.c = 0
```

```
    For i = 1 To n
```

```
        r1 = p.r
```

```
        p.r = x.r * p.r - x.c * p.c + a(i)
```

```
        p.c = x.r * p.c + x.c * r1
```

```
    Next i
```

```
End Sub
```

Здесь используются переменными типа – complex, введенного следующим оператором

```
Type complex
```

```
    r As Double
```

```
    c As Double
```

```
End Type
```

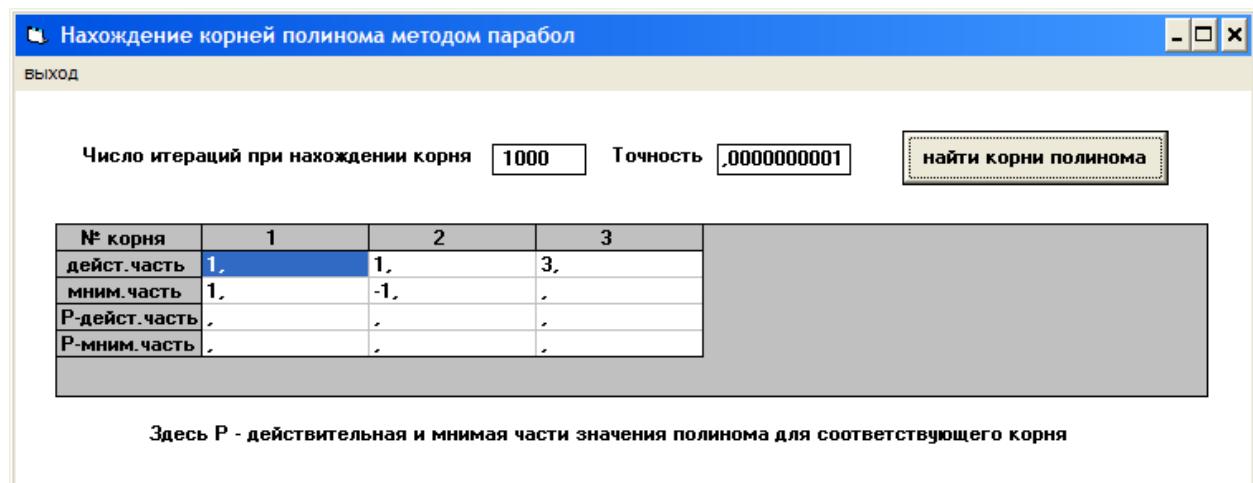


Рис. 4.11. Корни полином $x^3 - 5x^2 + 8x - 6$, определенные методом парабол. Форма вызывается пунктом меню «метод парабол» на главной форме рис.4.1

Заметим, что в рамках пакета Mixima результаты, отображенные на рис.4.11, получаются с помощью одной внутренней команды (здесь %i – обозначение мнимой единицы)

```

solve(x^3-5*x^2+8*x-6,[x]);
(%o18) [x = 1 - %i, x = %i + 1, x = 3]

```

В заключение приведем листинг программы, которая реализует метод парабол на C++ в среде программирования CodeBlocks с использованием собственного класса комплексных чисел.

```

#include <iostream>
#include <windows.h>
#include <math.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

using namespace std;

class complex
{
    double rn,in; //по-умолчанию - private
public:
    //3 конструктора
    complex(){rn=0;in=0;}
    complex(double r,double i){rn=r;in=i;}
    //другой вариант передачи параметров конструктору
    complex(double r):rn(r),in(0){}
    //его методы
    double modulo(){return sqrt(rn*rn+in*in);}
    double argument(){ return atan2(in,rn);}
    double get_rn(){return rn;}
    double get_in(){return in;}
    void get(double &r,double &i) //получить значения комп.числа
    {
        r=rn;i=in;
    }
    void set(double r,double i) //присвоить значения комп.числа
    {
        rn=r;in=i;
    }
    //переопределение арифметических операций
    complex operator+(complex n2)
    {
        complex sum(rn+n2.rn,in+n2.in);
        return sum;
    }
    complex operator-(complex n2)
    {
        complex sub(rn-n2.rn,in-n2.in);
        return sub;
    }
    complex operator*(complex n2)
    {

```

```

complex mul(rn*n2.rn-in*n2.in,rn*n2.in+in*n2.rn);
    return mul;
}
complex operator/(complex n2)
{
    double zn=n2.rn*n2.rn+n2.in*n2.in;
    complex del((rn*n2.rn+in*n2.in)/zn,(in*n2.rn-rn*n2.in)/zn);
    return del;
}
bool operator==(complex c){
    bool fl=FALSE;
    if ((rn==c.rn)&&(in==c.in))fl=TRUE;
    return fl;
}
//квадратный корень
void sq2(double &r,double &i)
{
    double buf;
    buf=sqrt(rn*rn+in*in);
    r=sqrt((buf+rn)/2);
    i=sqrt((buf-rn)/2);
    if (in<0) i=-i;
}

//вычисляет комплексное значение (rp,ip) полинома степени n в точке x -> вызов x.pol(...)
void pol(int n,double pa[],double &rp,double &ip)
{
    int i;
    double r,xr,xi;
    rp=pa[0];ip=0;
    for(i=1;i<n+1;i++)
    {
        r=rp;
        rp=rn*rp-in*ip+pa[i];
        ip=rn*ip+in*r;
    }
}
};

void dev1(int &n,double pa[],double a);
void dev2(int &n,double pa[],double p,double g);
void muller(int n,double pa[],int iter,double eps,complex xp[]);

//деление полинома на (x-a)
void dev1(int &n,double pa[],double a)
{
    int i;
    for(i=1;i<n;i++)pa[i]=pa[i]+pa[i-1]*a;
    n=n-1;
}
//деление полинома на (x^2+p*x+g)
void dev2(int &n,double pa[],double p,double g)

```

```

{
    int i;
    pa[1]=pa[1]-pa[0]*p;
    for(i=2;i<n-1;i++)pa[i]=pa[i]-pa[i-1]*p-pa[i-2]*g;
    n=n-2;
}
//корень для полинома 1-го порядка
void pol1(int &n,double pa[],double &rx,double &ix)
{
    rx=-pa[1]/pa[0]; ix=0; n=n-1;
}
//корни для полинома 2-го порядка
void pol2(int &n,double pa[],double rx[],double ix[])
{
    double d,zn;
    int fl;
    d=pa[1]*pa[1]-4*pa[0]*pa[2];zn=2*pa[0];
    if(d>=0)fl=1;d=sqrt(abs(d));
    if(fl==1)
    {
        rx[0]=(-pa[1]+d)/zn;ix[0]=0;
        rx[1]=(-pa[1]-d)/zn;ix[1]=0;
    }
    else
    {
        rx[0]=-pa[1]/zn;ix[0]=d/zn;
        rx[1]=rx[0];ix[1]=-ix[0];
    }
    n=n-2;
}
//метод парабол - находит все корни полинома
void muller(int n,double pa[],int iter,double eps,complex xp[],int &nr)
{
    complex l[3],zn[3];
    double dz,dz1,dz2,norm;
    complex z0,z1,z2,z3,z4,z5;
    complex aa,bb,cc,dd;
    complex l0,l1,l2;
    int i,it;
    double r,m,rx[2],ix[2]; //действ. и компл. составляющие компл. чисел и корней
    double p,q;
    while (n>2) //цикл по поиску корней
    {
        //наличие нулевого корня
        if(abs(pa[n])<0.00001)
        {
            xp[nr].set(0,0);
            nr=nr+1;n=n-1;
            goto Label;
        }
        //значение полинома в 3-х точках
        zn[0].set(-1,0);zn[1].set(1,0);zn[2].set(0,1);
    }
}

```

```

for(i=0;i<3;i++)
{
    zn[i].pol(n,pa,r,m); l[i].set(r,m);
}
norm=l[2].modulo();

it=0;
while (norm>eps) // поиск очередного корня
{
    z0=zn[0]-zn[2]; z1=zn[0]-zn[1]; z3=z0*z1; l0=l[0]/z3; //L0
    z0=zn[1]-zn[0]; z1=zn[1]-zn[2]; z3=z0*z1; l1=l[1]/z3; //L1
    z0=zn[2]-zn[1]; z1=zn[2]-zn[0]; z3=z0*z1; l2=l[2]/z3; //l2
    aa=l0+l1+l2; //A
    z3=zn[2]+zn[1]; z0=l0*z3; //L0*(z(2)+z(1))
    z3=zn[0]+zn[2]; z1=l1*z3; //L1*(z(0)+z(12))
    z3=zn[1]+zn[0]; z2=l2*z3; //L2*(z(1)+z(0))
    bb=z0+z1+z2; //B
    z3=zn[2]*zn[1]; z0=l0*z3; //L0*z(2)*z(1)
    z3=zn[0]*zn[2]; z1=l1*z3; //L1*z(0)*z(12)
    z3=zn[1]*zn[0]; z2=l2*z3; //L2*z(1)*z(0)
    cc=z0+z1+z2; //C
    z3=bb*bb; z4=aa*cc; z4=z4*4;
    dd=z3-z4; //D
    dd.sq2(r,m); z3.set(r,m); //z3=sqrt(dd)
    aa=aa*2; //2A
    z4=bb+z3; z5=bb-z3;
    z0=z4/aa; z1=z5/aa; //два корня
    //находим корень, ближайий к точке zn[2]
    z3=zn[2]-z0;dz1=z3.modulo();
    z4=zn[2]-z1;dz2=z4.modulo();
    //сдвигаем точки и значения полиномов в них
    zn[0]=zn[1]; zn[1]=zn[2]; l[0]=l[1];l[1]=l[2];
    if(dz1<dz2){dz=dz1;zn[2]=z0;}else{dz=dz2;zn[2]=z1;}
    it=it+1;
    if(it>iter)
    {
        cout<<"\nПри поиске корня исчерпаны все итерации";
        return;
    }
    zn[2].pol(n,pa,r,m); l[2].set(r,m); //новое значение полинома в 3-ий точке
    norm=l[2].modulo();
} //while 2
xp[nr]=zn[2]; nr=nr+1;
zn[2].get(r,m);
if(fabs(m)<0.00001) //действительный корень
{
    dev1(n,pa,r); //выделяем этот корень
}
else //комплексные корни
{
    xp[nr].set(r,-m);nr=nr+1; //сохраняем второй корень
    p=-2*r,q=r*r+m*m; dev2(n,pa,p,q); //выделяем корни
}

```

```

        }
        Label:;
    } //while 1
//корни линейного или квадратичного уравнений
if(n==1)
{
    pol1(n,pa,r,m); xp[nr].set(r,m); nr=nr+1;
}
else
{
    pol2(n,pa,rx,ix);
    xp[nr].set(rx[0],ix[0]); nr=nr+1;
    xp[nr].set(rx[1],ix[1]); nr=nr+1;
}
}

// главная программа
int main()
{
    double fn,r,m;
    double a,p,q;
    int n,i,j,nr;
    int iter=1000;
    double eps=0.00001;
    char s[2];

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

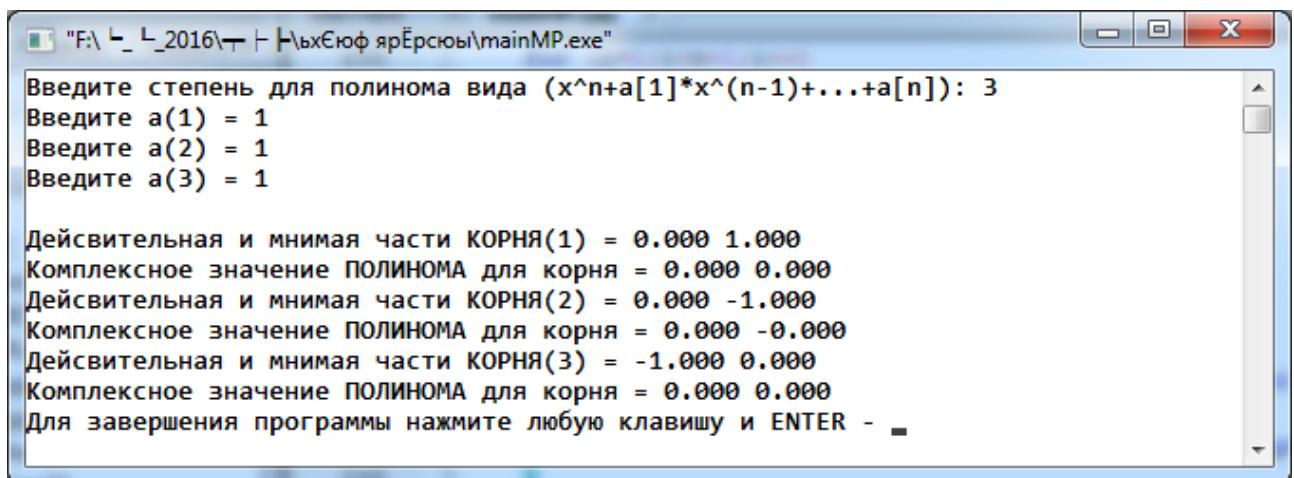
//ПОЛИНОМ !!!!
cout<<"Введите степень для полинома вида (x^n+a[1]*x^(n-1)+...+a[n]): ";
cin>>n;
double *pa=new double[n+1]; pa[0]=1;
double *bufpa=new double[n+1]; bufpa[0]=1;
complex *xp=new complex[n];
for (i=1;i<n+1;i++)
{
    cout<<"Введите a("<<i<<") = "; cin>>pa[i];
    bufpa[i]=pa[i];
}
nr=0;
muller(n,pa,iter,eps,xp,nr);
for (i=0;i<nr;i++)
{
    xp[i].get(r,m);
    printf("\nДействительная и мнимая части КОРНЯ(%d) = %.3f %.3f",i+1,r,m);
    xp[i].pol(n,bufpa,r,m);
    printf("\nКомплексное значение ПОЛИНОМА для корня = %.3f %.3f",r,m);
}

cout<<"\nДля завершения программы нажмите любую клавишу и ENTER - ";
cin>>s;

```

```
    return 0;  
}
```

Результаты ее работы представлены в следующем скриншоте.



Вопросы и задания для самоконтроля

- Почему при вычислении значения полинома схема Горнера является предпочтительной по сравнению с другими способами?
 - В каких случаях используются схемы деления полинома на линейный и квадратичный множитель?
 - Каким образом можно грубо оценить местоположение корней полинома на комплексной плоскости?
 - Опишите, каким образом можно с помощью приведенной выше программы последовательно найти все корни полинома.
 - В чем отличие методов наискорейшего и покоординатного спусков?
 - Составьте алгоритм, реализующий метод парабол.
 - Попробуйте найти все корни полинома с помощью пакета Maxima:
$$x^3 - 5x^2 + 8x - 6.$$

Задания для практикума

В рамках практикума предлагается разработать какую-либо программу аналогичную тем, что упоминаются в данном пособии. Возможно использования математического пакета Maxima.

Привем список указанных программ с кратким описанием их возможностей:

1. *Нормы* – приложение позволяет рассчитать три нормы произвольной матрицы A , введенной пользователем. Поскольку программа в процессе своей работы определяет все собственные числа вспомогательной матрицы $A^T A$, то с ее помощью можно вручную найти меру обусловленности исходной матрицы.
2. *ITER_MET* – в приложении реализованы три итерационных метода, с помощью которых можно найти решения произвольной системы уравнений. Исходную систему можно автоматически преобразовать в равносильную систему с положительно определенной симметричной матрицей. В приложении исходную систему можно ввести вручную или сформировать случайным образом с помощью датчика равномерно распределенных псевдослучайных чисел.
3. *CONVERT* – приложение позволяет находить для произвольной матрицы обратную и значение определителя.
4. *ORTGAUSS* – приложение позволяет найти решение произвольной системы уравнений методом оптимального исключения или первым методом ортогонализации.
5. *KV_ROOT* – приложение позволяет найти решение произвольной системы уравнений с симметричной положительно определенной матрицей методом квадратного корня или вторым методом ортогонализации. В приложении можно преобразовать обычную систему в равносильную с симметричной положительно определенной матрицей.
6. *REGUL* – приложение строит решение для произвольной системы и ее регуляризованного прототипа. Может быть использована для поиска нормального решения для плохо обусловленных систем.
7. *LMDMAX* – в приложении реализована итерационная процедура нахождения максимального по модулю собственного числа, которую в рамках данного приложения можно использовать и для нахождения минимального по модулю собственного числа, а также для определения границ отрезка, на котором располагаются все действительные собственные числа симметричной матрицы.
8. *M_DANIL* – в приложении для нахождения характеристического полинома произвольной матрицы используется метод Данилевского. Предусмотрено решение этой задачи и в случае, когда полином распадается на несколько полиномов. Сами собственные значения матрицы находятся методом парабол.
9. *KRILOV* – приложение позволяет определить собственные значения произвольной матрицы через ее характеристический полином, коэффициенты которого находятся методом Крылова.
10. *ВРАЩЕНИЕ* – в приложении методом вращения решается полная проблема собственных значений произвольной симметричной матрицы. В приложении

исходная матрица может быть сформирована случайным образом с помощью датчика равномерно распределенных псевдослучайных чисел.

11. *МПАРАБОЛ* – приложение позволяет: локализовать местоположение корней полинома на комплексной плоскости; рассчитать значение полинома в произвольной точке; выделить из него действительный корень, а также найти методом парабол все его корни.

К заданию можно добавить вычислительные исследования:

1. Исследовать скорость сходимости итерационных методов для конкретной системы линейных уравнений.
2. Исследовать точность различных прямых методов при решении конкретной системы линейных уравнений.
3. Исследовать зависимость нормального решения плохообусловленной системы от значения коэффициента регулирования.
4. Найти все собственные числа конкретной матрицы, используя все возможные методы. Сравнить результаты.
5. Найти графически местоположение на комплексной плоскости корней конкретного полинома и определить их точное значение.

Список литературы

1. Турчак Л.И. Основы численных методов. М., Наука, 1987
2. Крылов В.И., Бобков В.В., Монастырский П.И. Вычислительные методы. Т.1, М., Физматгиз, 1976
3. Калиткин Н.Н. Численные методы. М., Наука, 1978
4. Волков Е.А. Численные методы. М., Наука, 1987
5. Демидович Б.П., Марон И.А. Основы вычислительной математики. М., Наука, 1966
6. Бахвалов Н.С. Численные методы. М., Наука, 1973
7. Воеводин В.В. Вычислительные основы линейной алгебры. М., Наука, 1977
8. Фаддеев Д.К., Фаддеева В.Н. Вычислительные методы линейной алгебры. М., Наука, 1963
9. Бахвалов Н.С., Жидков Н.П., Кобельков Г.М. Численные методы. М., Бином, 2008
10. Козин Р.Г. Алгоритмы численных методов линейной алгебры. М., Логос, 2009

Приложение. Некоторые сведения о математическом пакете Maxima.

Maxima – свободно распространяемый математический пакет (можно свободно скачать из Интернета), имеющий большое число внутренних функций, которые без громоздкого программирования позволяют выполнять различные сложные математические вычисления с графическим оформлением результатов. Экран пакета показан на рис. 1 и 2.

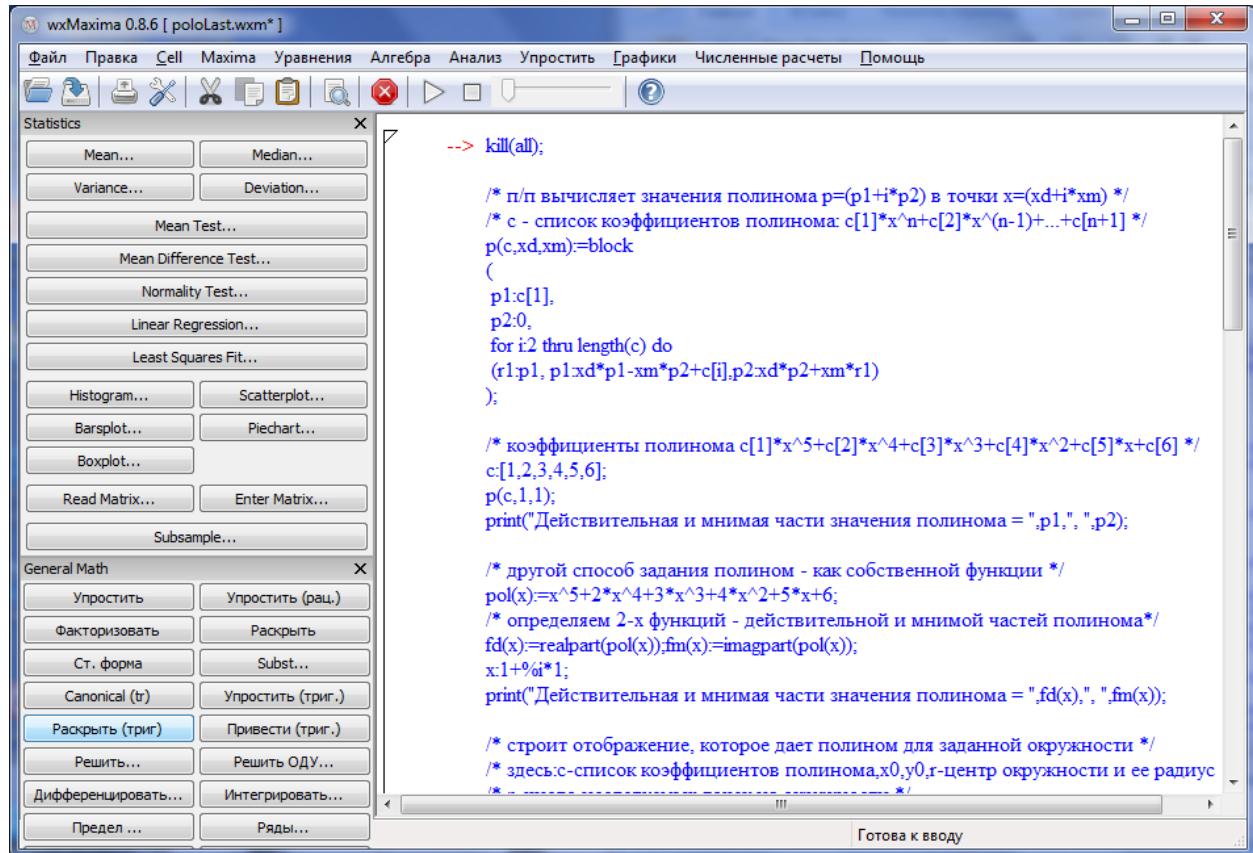


Рис. 1. Экран пакета в режиме отображением двух панелей (слева) с перечнем некоторых внутренних функций и с набранной программой (справа).

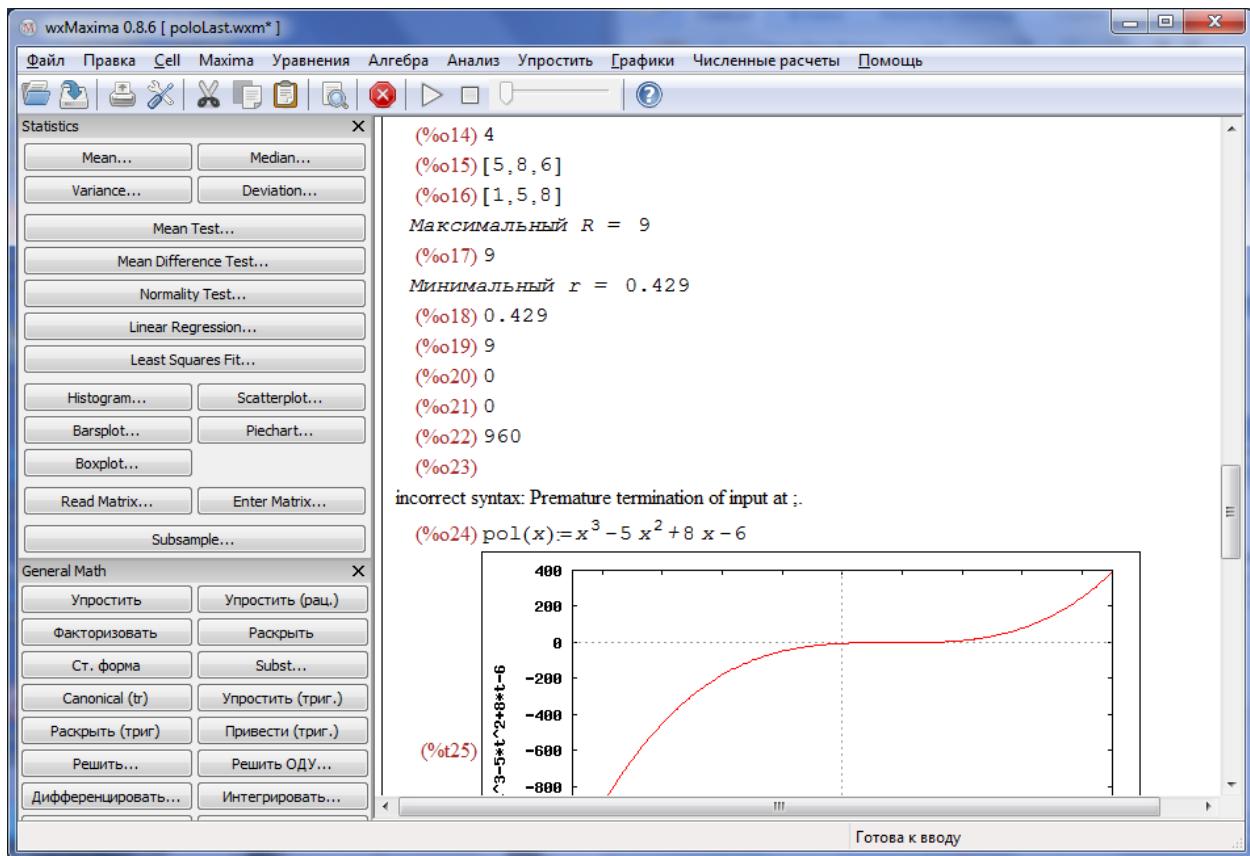


Рис. 2. Экран пакета с результатами выполнения набранной программы.

Далее описываются некоторые операции и внутренние функции пакета, которые использовались в листингах, приведенных в пособии. Более подробно со всеми особенностями пакета можно ознакомиться, обращаясь к «помощи» самого пакета.

Простейшие операции и операторы:

: - операция присвоение; := - определение функции; =, >, <, >=, <=, # - операции сравнение; **and**, **or** – логические операции.

, или ; - закрывают каждую команду, (ком1,ком2,...) – задание блока команд.

if условие **then** команда/блок команд **else** команда/блок команд – условный оператор;

for i :i1 **thru** i2 **step** di **do** команда или (блок команд) – простейший вариант оператора цикла;

while условие **do** команда или (блок команд) – условный оператор цикла;

print("text",v1,..,vn) – выводит значения аргументов на экран;

%pi, **%i** – встроенное число «пи» и мнимая единица.

Пример.

Код программы.

n:5;

if n>=0 and n<=10 then (print("есть вхождение"),n:10)

```
else print("нет вхождения");
for i thru 3 do print(i);
```

Результат.

(%o14) 5

есть вхождение

(%o15) 10

```
1
2
3
```

Функции для работы с матрицами:

m:matrix([..],[..],[..]) – определение произвольной матрицы;

m:addcol(m,b) – добавить к матрице справа столбец;

m:addrow(m,b) – добавляет к матрице снизу строку;

submatrix(i1,...,in,m) – удаление из матрицы m i1,...,in строк;

submatrix(m,j1,...,jn) – удаление из матрицы m j1,...,jn столбцов;

A:copy(m) – копирует матрицу;

determinant(m) – вычисляет определитель матрицы;

rank(m) – определяет ранг матрицы;

invert(m) – находит обратную матрицу;

transpose(m) - строит транспонированную матрицу;

tracematrix(m) – находит след матрицы **m**;

ident(n) – генерация единичной матрицы размерности n*n;

zeromatrix(n,m) – генерация нулевой матрици размерности n*m;

A.B / A.b – «точка» операция перемножение матриц или матрицы-столбца;

genmatrix(f,n,m) – генерация матрицы из значений функции f[i,j] для i=1,n; j=1,m;

write_data(M,"c:/MaximaData/datamatrix.csv",comma) – сохранение матрицы M в файле datamatrix.csv;

TR:read_matrix("c:/MaximaData/datamatrix.csv") – чтение сохраненной матрицы из файла в матрицу TR;

eigenvalues(m) – определяет собственные числа матрицы и их кратность.

Пример.

Код программы.

```
kill(all);          /* предварительно все вычистить */
numer:true;        /* выводить числа в действительном виде */
fpprintprec:5;     /* в числах остался только 5 цифр */
y0:matrix([1],[4]);
Y:copy(y0);
y1:matrix([5],[9]);
y2:matrix([6],[3]);
Y:addcol(y1,Y); Y:addcol(Y,y2);
```

```

r:[7,2,3];
Y:addrw(Y,r);
print("определитель = ",determinant(Y));
m:invert(Y);
mt:transpose(Y);
Z:zeromatrix(2,2);
E:ident(2);
f[i,j]:=random(2.0); /* функция-шаблон – генерирует случайное число из диапозона [0,2] */
print("сгенерированная случайная матрица:");
G:genmatrix(f,2,2);
a:matrix([1,2,0,0,0,0],[4,5,0,0,0,0],[3,4,5,0,0,0],[2,3,4,0,0,0],[3,4,2,1,2,1],[6,5,4,3,2,1]);
eigenvalues(a); /* собственные значения матрицы и их кратность */

```

Результат.

$$(\%o4) \begin{bmatrix} 1 \\ 4 \end{bmatrix}$$

$$(\%o5) \begin{bmatrix} 5 \\ 9 \end{bmatrix}$$

$$(\%o6) \begin{bmatrix} 6 \\ 3 \end{bmatrix}$$

$$(\%o7) \begin{bmatrix} 5 & 1 \\ 9 & 4 \end{bmatrix}$$

$$(\%o8) \begin{bmatrix} 5 & 1 & 6 \\ 9 & 4 & 3 \end{bmatrix}$$

$$(\%o9) [7, 2, 3]$$

$$(\%o10) \begin{bmatrix} 5 & 1 & 6 \\ 9 & 4 & 3 \\ 7 & 2 & 3 \end{bmatrix}$$

определитель = -36

$$(\%o12) \begin{bmatrix} -0.167 & -0.25 & 0.583 \\ 0.167 & 0.75 & -1.0833 \\ 0.278 & 0.0833 & -0.306 \end{bmatrix}$$

$$(\%o13) \begin{bmatrix} 5 & 9 & 7 \\ 1 & 4 & 2 \\ 6 & 3 & 3 \end{bmatrix}$$

(%o14) $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$

(%o15) $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

(%o16) $f_{i,j} := \text{random}(2.0)$

(%o17) сгенерированная случайная матрица:

(%o18) $\begin{bmatrix} 1.8276 & 0.39 \\ 0.965 & 0.711 \end{bmatrix}$

Исходная матрица и списки ее собственных значений и их кратность

(%o57) $\begin{bmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 4 & 5 & 0 & 0 & 0 & 0 \\ 3 & 4 & 5 & 0 & 0 & 0 \\ 2 & 3 & 4 & 0 & 0 & 0 \\ 3 & 4 & 2 & 1 & 2 & 1 \\ 6 & 5 & 4 & 3 & 2 & 1 \end{bmatrix}$

(%o58) $[[-0.464, 6.4641, 0, 3, 5], [1, 1, 2, 1, 1]]$ -

Другие функции:

block([л1,..,лп],команда1,..,командам) – определение тела подпрограммы, где л1,..,лп – локальные переменные, команда1,..,командам – команды, выполняемые подпрограммой;

return(а) – выход из подпрограммы с передачей значения а;

create_list(f(i,j),i,ib,ie,j,jb,je) – формирует список из значений функции f(i,j) для i=ib,ie; j=jb,je;

append(L1,L2) – добавление справа к списку L1 списка L2;

length(m) – длина списка, размерность матрицы;

lmax(L), lmin(L) – определение максимального и минимального элемента списка L:

abs(n) – абсолютное значение числа n;

floor(n) – возвращает целую часть действительного числа;

round(n) – округляет действительное число до ближайшего целого;

mod(n,m) – возвращает остаток от деления n на m;

sqrt(x) – корень от числа x;

signum(n) – возвращает: 1 при n>0, -1 при n<0, 0 при n=0;

random(n) – генерация псевдослучайного числа из интервала (0,n);

sum(f(i),i,ib,ie) – суммирование значений функции для i=ib,ie;

integrate(f(x),x,a,b) – вычисляет определенный интеграл от функции f(x) на отрезке [a,b];

factorial(n) – n!;

product(f(i),i,n1,n2) - $\prod_{i=n1}^{n2} f(i)$.

Пример.

Код программы.

```
mul_matr(A,B):=block /* подпрограмма перемножант матрицы A и B */
(
    [n,buf,i,j,k],
    n:length(A),
    AB:zeromatrix(n,n), /* глобальная переменная */
    for i thru n do
        for j thru n do
            for k thru n do AB[i,j]:=AB[i,j]+A[i,k]*B[k,j]
);
A:matrix([1,2,3],[4,5,6],[8,9,7]);
n:length(A);
/* вводим функцию для расчета кубической нормы матрицы M*/
nrm(M):=lmax(create_list(sum(abs(M[i,j]),j,1,length(M)),i,1,length(M)));
normKub:nrm(A);
A_1:invert(A);
mul_matr(A,A_1); /* умножаем обратную матрицу на исходную */
AB;
/* тестирование встроенных функций */
a:floor(5.1); b:floor(5.8); c:round(6.4); d:round(6.7); e:mod(5.1,2);
```

Результат ее работы:

(%o22) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 8 & 9 & 7 \end{bmatrix}$

(%o23) 3

(%o24) $nrm(M) := lmax\left(create_list\left(\sum_{j=1}^{length(M)} |M_{i,j}|, i, 1, length(M) \right) \right)$

(%o25) 24

(%o26) $\begin{bmatrix} -2.1111 & 1.4444 & -0.333 \\ 2.2222 & -1.8889 & 0.667 \\ -0.444 & 0.778 & -0.333 \end{bmatrix}$

(%o27) done

(%o28) $\begin{bmatrix} 1.0 & 0.0 & 0.0 \\ -4.44089 \cdot 10^{-16} & 1.0 & -2.22045 \cdot 10^{-16} \\ 0.0 & -8.88178 \cdot 10^{-16} & 1.0 \end{bmatrix}$

(%o29) 5

(%o30) 5

(%o31) 6

(%o32) 7

(%o33) 1.1

Команды, связанные с решением обыкновенных уравнений:

solve([eq1,..,eqn],[x1,..,xn]) – находит решение системы уравнений eq1,..,eqn, относительно переменных x1,..,xn;

globalsolve:true – значения решения системы уравнений непосредственно присваиваются указанным в команде переменным x1,..,xn;

to_poly_solve([pol(x)=0],[x]) – находит корни полинома pol(x);

realpart(cn) – выделяет действительную часть **rp** комплексного числа **cn=rp+%i*imp;**

imagpart(cn) – выделяет мнимую часть **imp** комплексного числа **cn=rp+%i*imp;**

ev(x,s[i]) – возвращает значение i-го корня полинома из именного списка решения, который строит команда **s:solve([p(x)=0],[x])**.

Пример.

Код программы.

cn:=1+%*i****2;**

realpart(cn); imagpart(cn);

globalsolve:true;

```

solve([x1+2*x2+3*x3=1,3*x1+5*x2-7*x3=2,2*x1-7*x2+3*x3=3],[x1,x2,x3]);
print(" Компоненты вектора невязки для решения = ",x1+2*x2+3*x3-1,
",",3*x1+5*x2-7*x3-2,",", 2*x1-7*x2+3*x3-3);
pol(z):=z^4+2*z^3+3*z^2+4*z-5;
s:solve([pol(x)=0],[x]);
cr:ev(x,s[3]);
print(" Действительная и мнимая части третьего корня = ", realpart(cr), ",",
imagpart(cr));
listR:makelist(ev(x,s[i]),i,1,4);
print(" Список корней полинома = ", listR);
listR[2];

```

Результат.

(%o29) $2 \%i + 1$

(%o30) 1

(%o31) 2

(%o33) $[[x_1:1.0116, x_2:-0.11, x_3:0.0694]]$

Компоненты вектора невязки для решения = 0.0 , $4.44089 \cdot 10^{-16}$, $4.44089 \cdot 10^{-16}$

(%o35) $\text{pol}(z):=z^4+2 z^3+3 z^2+4 z-5$

(%o36) $[x=-2.0591, x=0.684, x=-1.8579 \%i - 0.312, x=1.8579 \%i - 0.312]$

(%o37) $-1.8579 \%i - 0.312$

Действительная и мнимая части третьего корня = -0.312 , -1.8579

(%o39) $[-2.0591, 0.684, -1.8579 \%i - 0.312, 1.8579 \%i - 0.312]$

Список корней полинома = $[-2.0591, 0.684, -1.8579 \%i - 0.312, 1.8579 \%i - 0.312]$

(%o40) $[-2.0591, 0.684, -1.8579 \%i - 0.312, 1.8579 \%i - 0.312]$

(%o41) 0.684

Некоторые графические команды:

wxplot2d(исходные данные и опции) / **plot2d(..)** и **wxplot3d(..)** / **plot3d(..)** – первая форма команды выводит двумерный или трехмерный график в основное окно, а вторая форма – в дополнительном. Во втором случае трехмерный график можно вращать.

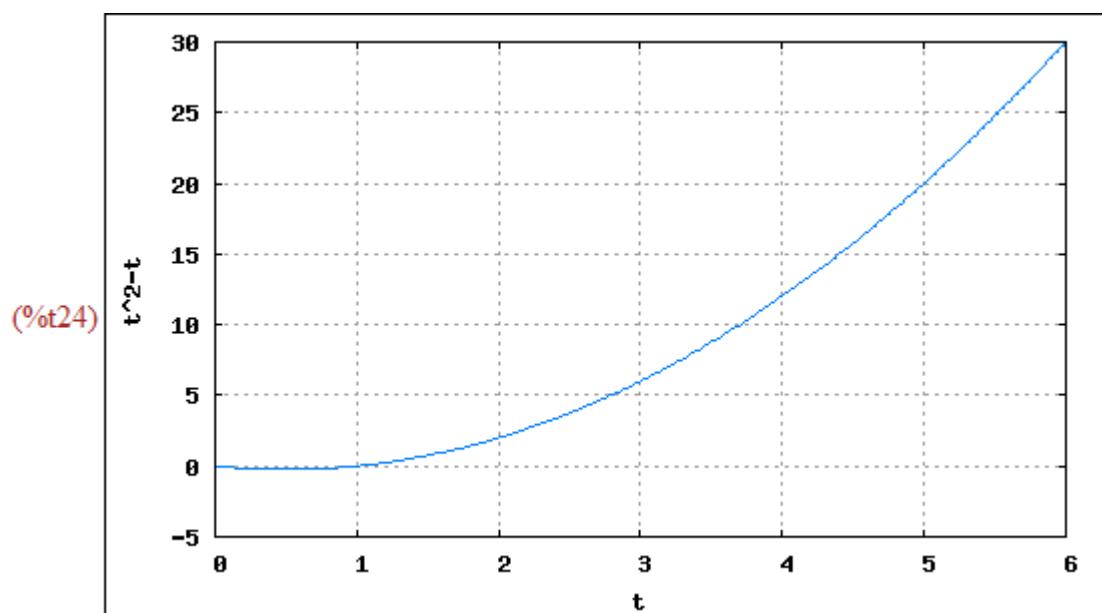
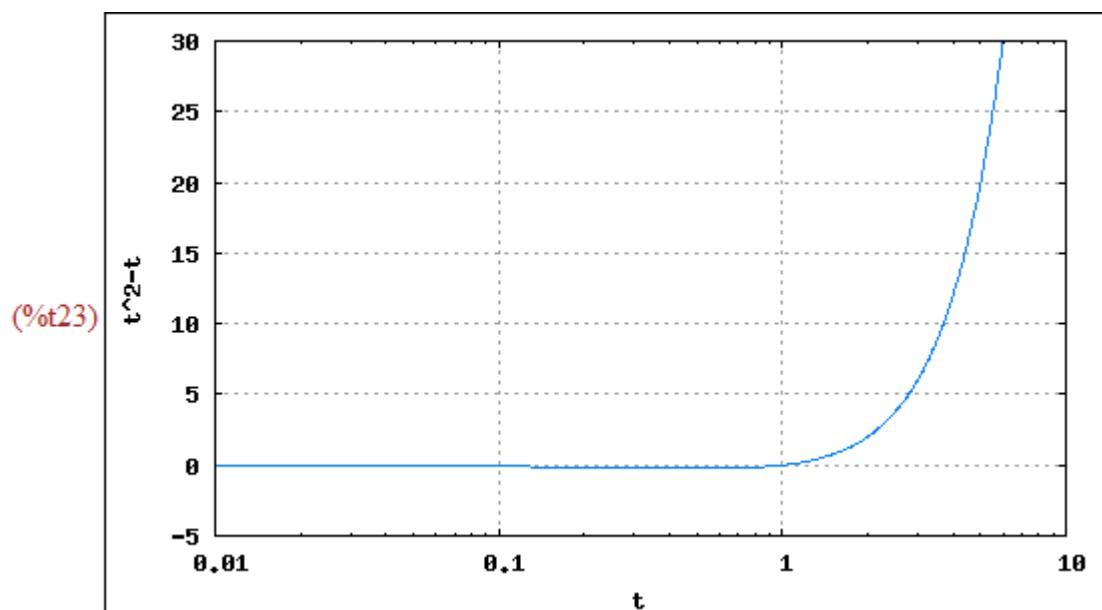
Приведем несколько вариантов этих команд с результатами их выполнения:

Пример 1.

Параметрическое задание кривых:

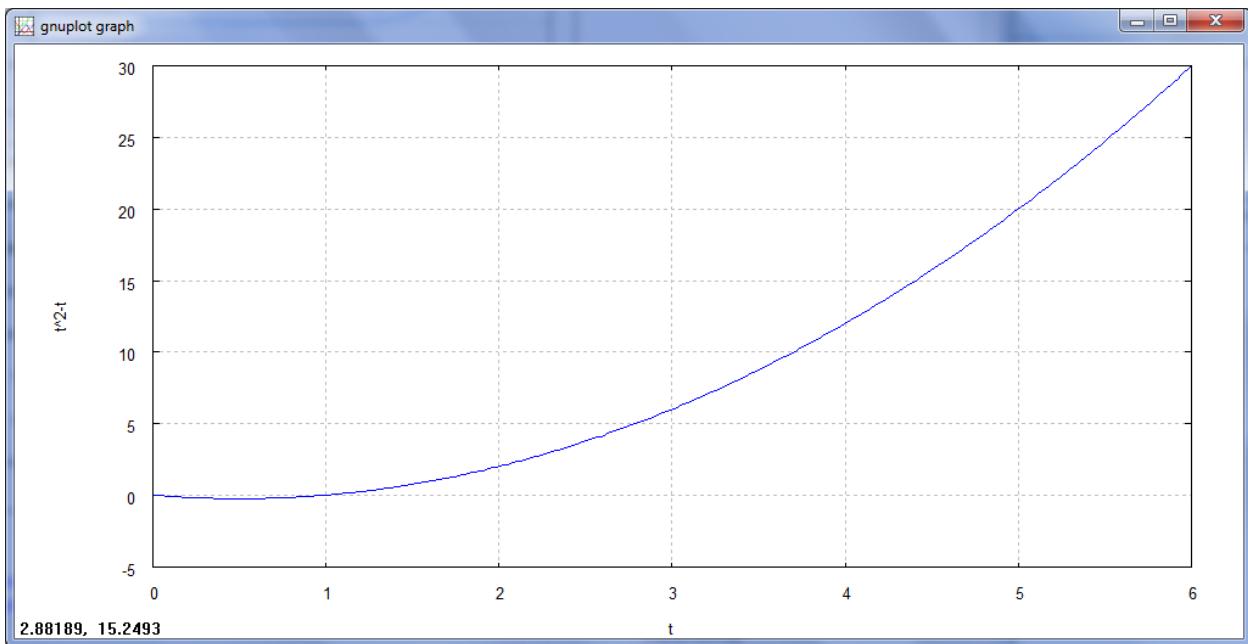
wxplot2d([['parametric, t, t^2-t, [t, 0, 6], [nticks, 300]]],
[gnuplot_preamble, "set grid;"], [logx]); - логарифмический график, при построении использовать 300 точек, выводить размерную сетку.

wxplot2d([['parametric, t, t^2-t, [t, 0, 6], [nticks, 300]]],
[gnuplot_preamble, "set grid;"]);



Пример 2.

```
plot2d([[parametric, t, t^2-t, [t, 0, 6], [nticks, 300]]],[gnuplot_preamble, "set grid;"]);  
- выводит график в отдельном окне.
```



Пример 3.

```
wxplot2d([discrete, [2,3,4,5], [3,6,7,8]],[gnuplot_preamble, "set grid;"]);
```

- при построении используются значения из двух списков.

```
wxplot2d([2^x-x-10],[x,-12,5],[gnuplot_preamble, "set grid;"]);
```

- строится график заданной функции на указанном интервале.

```
t:[4,7,16,31]; cf:[0.2131,0.05691,0.01237,0.00304];
```

```
cmax:[0.191,0.05094,0.009029,0.0000274];
```

```
cmid:[0.05743,0.01313,0.002089,0.000022];
```

```
plot2d([discrete, t,cf], [gnuplot_preamble, "set grid;"],
```

[style, lines], [color, blue], [point_type, asterisk],

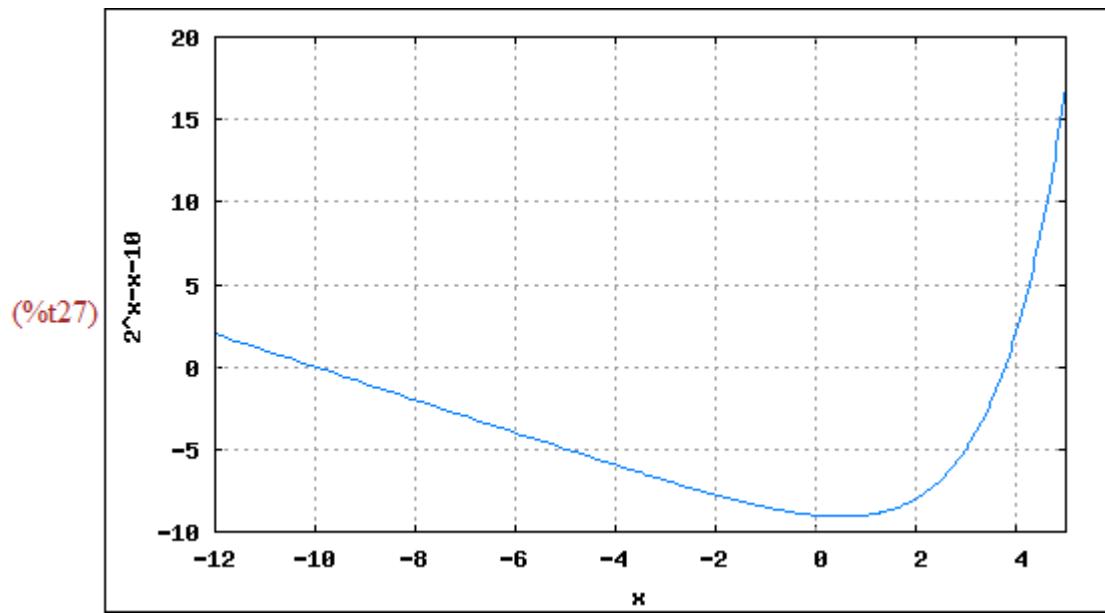
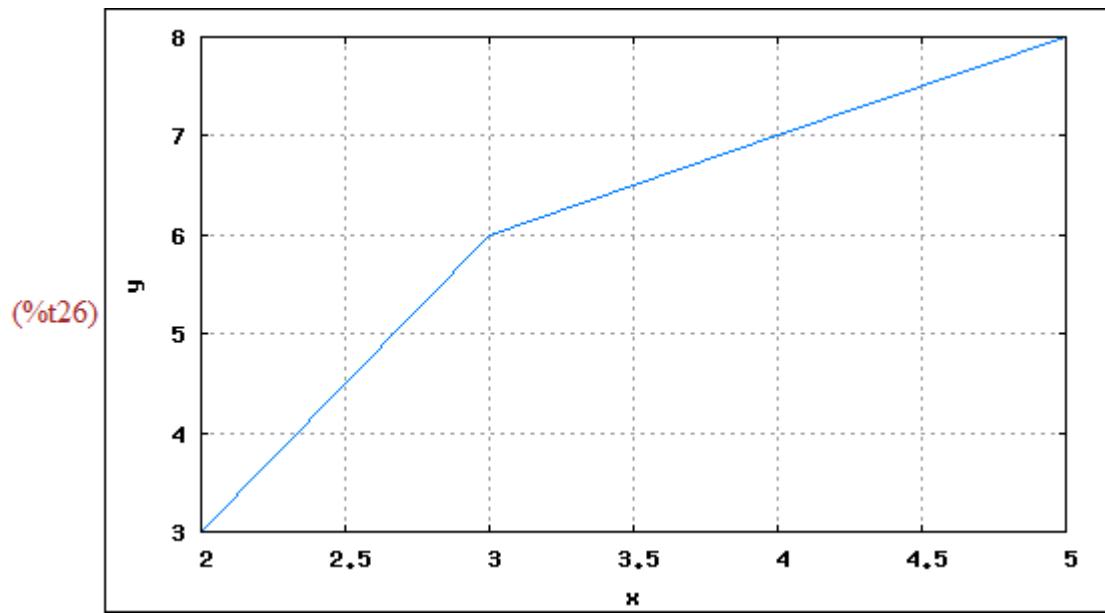
[xlabel, "число источников"], [ylabel, "значение функционала"]);

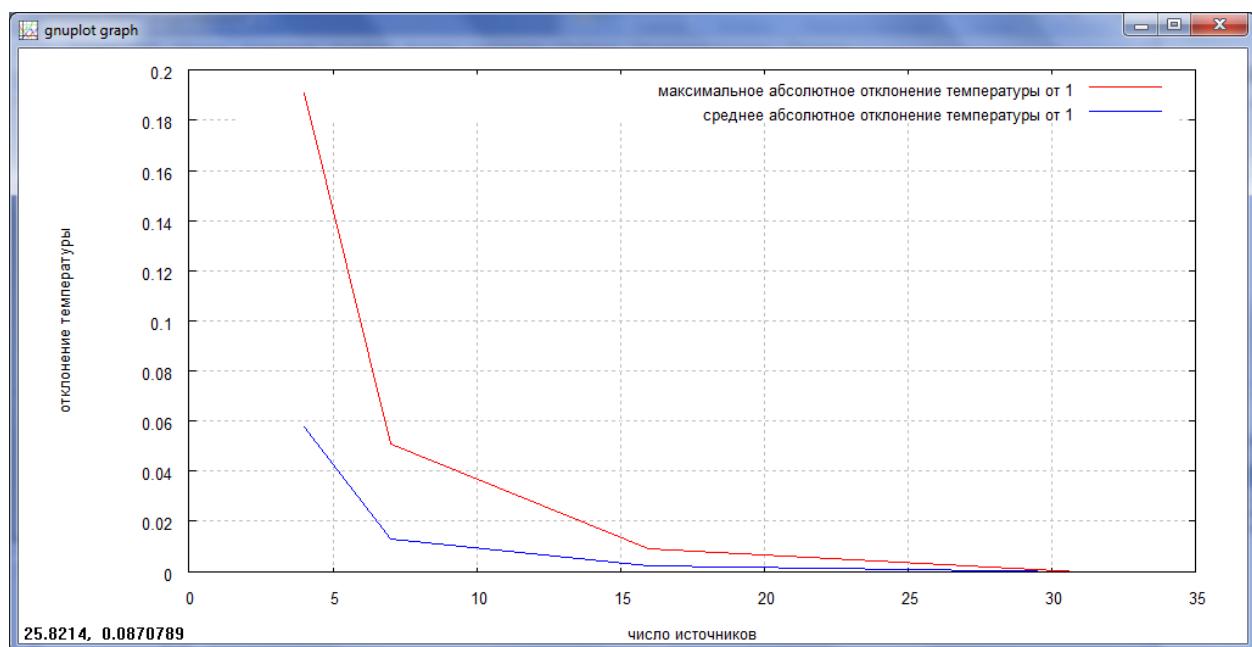
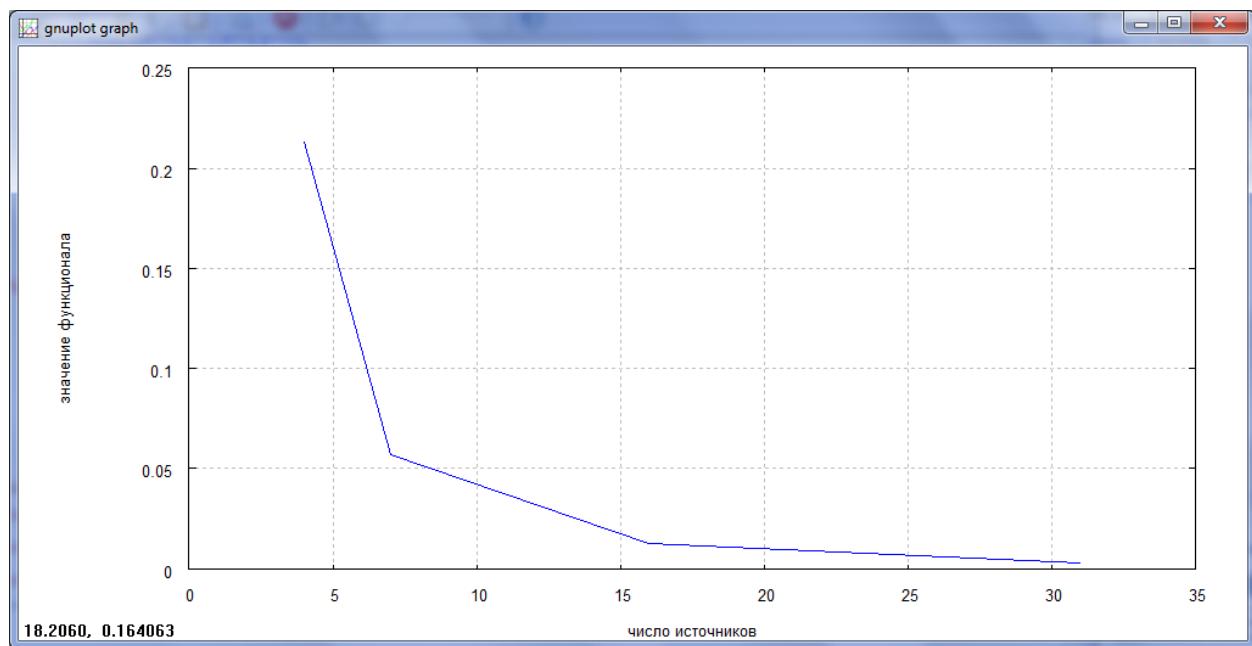
```
plot2d([[discrete, t,cmax],[discrete,t,cmid]], [gnuplot_preamble, "set grid;"],
```

[style, lines, lines], [color, red, blue], [point_type, asterisk],

[legend, "максимальное абсолютное отклонение температуры от 1", "среднее
абсолютное отклонение температуры от 1"],

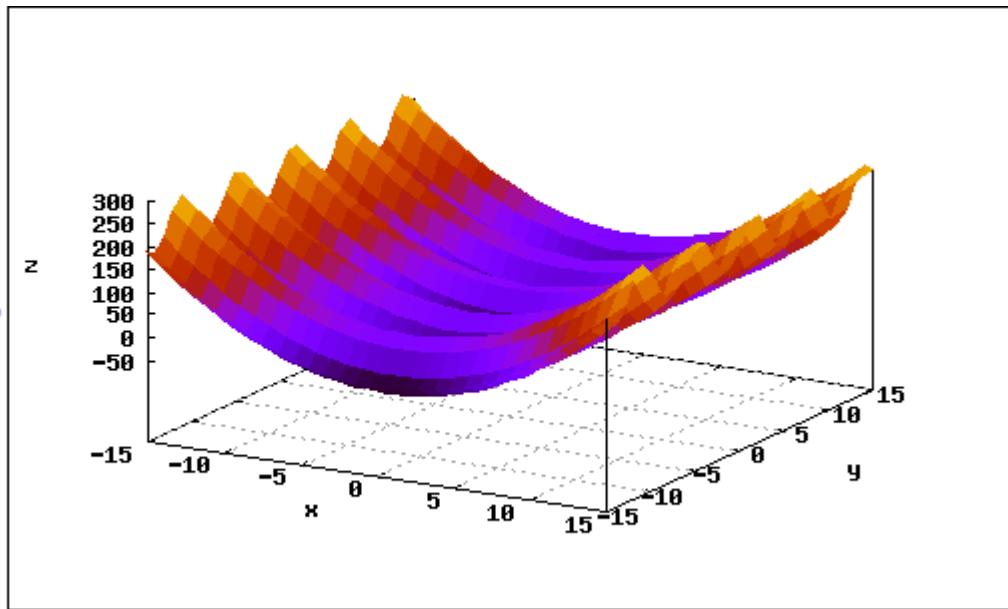
[xlabel, "число источников"], [ylabel, "отклонение температуры"]); - построение
двух графиков одновременно.





Пример 4. Построение трехмерного графика.

`wxplot3d(x^2+50*sin(y), [x,-15,15], [y,-15,15], [plot_format,gnuplot],
[gnuplot_preamble, "set grid; set pm3d at s; unset surf; unset colorbox"]);`
- строится трехмерный график заданной функции над указанной областью.



Пример 5. Построение трехмерных графиков в отдельном окне:

```
plot3d(x^2+50*sin(y), [x,-15,15], [y,-15,15], [plot_format,gnuplot],
[gnuplot_preamble, "set grid; set pm3d at s; unset surf; unset colorbox"]);
```

```
plot3d([x^2+y^2,x*y, [x,-5,5], [y,-5,5]], [plot_format,gnuplot]);
```

- строятся две поверхности.

```
plot3d ( log ( x^2*y^2 ), [x, -2, 2], [y, -2, 2], [z, -8, 4],[palette, false], [color, magenta, blue]);
```

- наложено ограничение на область значений функции z.

```
plot3d ( log ( x^2*y^2 ), [x, -2, 2], [y, -2, 2],[palette, false], [color, magenta, blue]);
```

- ограничение снято.

