Міністерство освіти України
Криворізький державний педагогічний інститут

В.В. Корольский
В.М. Соловйов
С.О. Семеріков

Комп'ютерна підтримка курсу лінійної алгебри

Кривий Ріг, 1998

Корольский В.В., Соловйов В.М., Семеріков С.О.

Комп'ютерна підтримка курсу лінійної алгебри. - Кривий Ріг: КДПІ, 1998 – 26 с.

Рецензенти:

к. ф.-м. н., доц. Рашевський М.О. (КДПІ)

вчитель-методист Теплицький І.О. (Центрально-Міська гімназія)

ВСТУП

Протягом останніх декількох років на кафедрі математики Криворізького педінституту викладається інтегрований курс алгебри та числових систем, для підтримки якого був створений відповідний навчальний посібник [6], у якому висвітлюються основні числові системи, алгебра комплексних чисел та алгебра многочленів від однієї змінної. Крім того, окремим виданням вийшов розділ «Векторні простори. Матриці і детермінанти», на подальший розвиток якого і запропоноване дане видання.

Основним нашим завданням було створити факультативний курс АЧС, в якому розглядаються деякі додаткові питання векторних просторів та лінійної алгебри. Потреба у такому факультативі викликана об'єктивними труднощами, з якими студенти стикаються при векторно-матричних обчисленнях - основним апаратом математичної фізики [1,2].

Наявність комп'ютера дозволяє інтенсифікувати процес обчислень, а сучасні мови програмування - представити його класично, у вигляді операцій над елементами відповідних алгебр. Як відомо, основними об'єктами лінійної алгебри є вектора та матриці, які засобами мови С++ досить легко перетворити на відповідні класи, зберігши при цьому природнє представлення матриці як впорядкованого кортежа арифметичних векторів, а вектора - як впорядкованого

кортежа об’єктів будь-якої природи.

На факультативі розглядаються лише арифметичні вектори, елементами яких є дійсні числа, і саме їх комп’ютерна реалізація дається за взірець для побудови відповідних матричних класів, в яких на обов’язковому рівні повинні бути реалізовані дії додавання, множення, транспонування, обернення, порівняння матриць та знаходження розв’язку СЛАР. На підвищеному рівні до них додаються функції обчислення мінорів, детермінантів, власних значень, алгебраїчних доповнень тощо; крім того, цікавим завданням може бути параметризація побудованих класів, яка дозволяє працювати з такими числовими об’єктами, як комплексні матриці, матриці матриць і таке інше.

Засвоєння цього курсу дає можливість розв’язувати складні задачі декількома рядками програми, що, в свою чергу, підвищує ефективність праці. На відміну від відомих пакетів для символьних обчислень, які теж можуть це робити, студент при побудові класів сам проводить всю підготовчу роботу, детально розбираючись в лінійній алгебрі та відповідному розділі чисельних методів, що сприяє підвищенню культури обчислень.

МЕТОДИЧНІ ОСОБЛИВОСТІ РЕАЛІЗАЦІЇ ОБ'ЄКТІВ ЛІНІЙНОЇ АЛГЕБРИ МОВОЮ ПРОГРАМУВАННЯ С++

У загальному випадку вектори та матриці - це певні алгебраїчні структури, яким у мові С++ можуть відповідати такі класи:

```
class vector
{
  long m;
  long double *vec;
 public:
  vector(char *);
  vector();
  vector(long);
  vector(long,long double *);
  vector(vector &);
  ~vector();
  friend vector operator+(vector &,vector &);
  friend vector operator-(vector &,vector &);
  friend long double operator*(vector &,vector &);
  friend vector operator*(long double &,vector &);
  friend vector operator*(vector &,long double &);
  friend ostream &operator<<(ostream &,vector &);
  friend istream &operator>>(istream &,vector &);
  vector operator=(vector &);
  vector operator-();
  vector operator~();//норма вектора
  long double operator!();//модуль вектора
  friend long operator==(vector &,vector &);
  friend long operator!=(vector &,vector &);
  long double &operator[](long a);
  long getm() { return m; }
};


class matrix
{
  long m,n;
```

```
  vector *mtr;
 public:
  matrix(char *);
  matrix();
  matrix(long,long);
  matrix(long,long,long double *);
  matrix(matrix &);
  ~matrix();
  friend matrix operator+(matrix &,matrix &);
  friend matrix operator-(matrix &,matrix &);
  friend matrix operator*(matrix &,matrix &);
  friend matrix operator*(long double &,matrix &);
  friend matrix operator*(matrix &,long double &);
  friend ostream &operator<<(ostream &,matrix &);
  friend istream &operator>>(istream &,matrix &);
  friend matrix operator%(matrix &,matrix &);
  matrix operator=(matrix &);
  matrix operator~();
  matrix operator-();
  friend long operator==(matrix &,matrix &);
  friend long operator!=(matrix &,matrix &);
  vector &operator[](long a);
  long getm() { return m; }
  long getn() { return n; }
};
```

Розглянемо докладніше структуру цих класів.

При комп'ютерній реалізації будемо використовувати найдовший (десятибайтовий) стандартний тип даних, найбільшим числом у якому є

```
const long double MAX_LONGDOUBLE=1.7976931348e308;
```

Створювати вектор можна по-різному. Наприклад, якщо він знаходиться на зовнішньому пристрої у форматі m d_1 d_2 ... d_m, де m - розмірність вектора, а d_i - його компоненти, то матимемо наступний конструктор:

```
vector::vector(char *f)
```

```
{
  ifstream fp=f;
  if(!fp)
  {
    cerr<<"Не можна відкрити файл "<<f<<"\n";
    exit(0);
  }
  fp>>m;
  if(m<=0)
  {
    cerr<<"Розмірність вектора некоректна\n";
    exit(0);
  }
  vec=new long double[m];
  if(vec==NULL)
  {
    cerr<<"Не вистачає пам'яті\n";
    exit(0);
  }
  for(long i=0;i<m&&fp>>vec[i];i++);
}
```

У випадку, коли нам відома лише розмірність вектора, але не відомі його компоненти, припускаємо, що даний вектор є нульовим:

```
vector::vector(long a):m(a)
{
  if(m<=0)
  {
    cerr<<"Розмірність вектора некоректна\n";
    exit(0);
  }
  vec=new long double[m];
  if(vec==NULL)
  {
    cerr<<"Не вистачає пам'яті\n";
    exit(0);
  }
  for(long i=0;i<m;vec[i++]=0);
}
```

Нарешті, нам можуть бути відомі як розмірність, так і компоненти вектора:

```
vector::vector(long a,long double *v):m(a)
{
  if(m<=0)
  {
    cerr<<"Розмірність вектора некоректна\n";
    exit(0);
  }
  vec=new long double[m];
  if(vec==NULL)
  {
    cerr<<"Не вистачає пам'яті\n";
    exit(0);
  }
  for(long i=0;i<m;i++)
    vec[i]=v[i];
}
```

Є ще один випадок, коли ми нічого не можемо сказати про розмірність і компоненти вектора - при створенні матриці.

```
vector::vector():m(1)
{
  vec=new long double[m];
  if(vec==NULL)
  {
    cout<<"Не вистачає пам'яті\n";
    exit(0);
  }
  *vec=0;
}
```

У реальних розрахунках можуть використовуватися вектори великих розмірностей, тому розміщуються вони у вільній пам’яті комп’ютера, а коли необхідність у них зникає - знищуються.

```
vector::~vector()
{
  delete []vec;
}
```

Необхідність в індексації вектора виникає у двох випадках:

- при отриманні компоненти вектора за її номером і
- при зміні не всього вектора, а лише однієї його компоненти.

При цьому, звичайно, слід враховувати можливість помилкового задання номеру компоненти: дозволений діапазон значень [0,m).

```
long double &vector::operator[](long a)
{
  static long double error=MAX_LONGDOUBLE;
  if(a>=0&&a<m)
    return vec[a];
  else
  {
    cerr<<"Індекс "<<a<<" знаходиться за межами вектора\n";
    return error;
  }
}
```

Створюючи вектор, можна побіжно ініціювати його даними з вже існуючого:

```
vector::vector(vector &e
{
  vec=new long double[m];
  if(vec==NULL)
  {
    cout<<"Не вистачає пам'яті\n";
    exit(0);
  }
  for(long i=0;i<m;i++)
```

```
    vec[i]=ex[i];
}
```

Додавання векторів є алгебраїчною операцією лише тоді, коли вектори однакової розмірності. Результатом додавання є вектор тієї ж розмірності, що й вихідні, компонентами якого є сума відповідних компонент вихідних векторів.

```
vector operator+(vector &f,vector &s)
{
  if(f.m!=s.m)
  {
    cout<<"Розмірність векторів не співпадає\n";
    exit(0);
  }
  vector temp(f.m);
  for(long i=0;i<f.m;i++)
    temp[i]=f[i]+s[i];
  return temp;
}
```

Введемо декілька допоміжних унарних операцій:

* «мінус» - операція співставлення даному вектору іншого, рівного за довжиною, співнапрямленого, але протилежного за напрямком:

```
vector vector::operator-()
{
  vector temp(m);
  for(long i=0;i<m;i++)
    temp[i]=-(*this)[i];
  return temp;
}
```

* нормування вектора - ділення вектора на його модуль, що дає співнапрямлений з даним одиничний вектор:

```
vector vector::operator~()
{
  vector temp(m);
  long double scalp=(*this)*(*this);
  for(long i=0;i<m;i++)
    temp[i]=(*this)[i]/sqrtl(scalp);
  return temp;
}
```

Операція, яку алгебраїчною назвати не можна - це, скоріше, приклад дуже розповсюдженого тернарного відношення «скалярний добуток двох векторів»:

```
long double operator*(vector &f,vector &s)
{
  if(f.m!=s.m)
  {
    cout<<"Помножити не можна!\n";
    exit(0);
  }
  for(long i=0,temp=0;i<f.m;i++)
    temp+=f[i]*s[i];
  return temp;
}
```

Модуль вектора - відношення, яке ми означимо не зовсім стандартно, а саме як квадратний корінь із скалярного добутку вектора на самого себе:

```
long double vector::operator!()
{
  return sqrtl((*this)*(*this));
}
```

Важливим при підрахунках є тернарне відношення «множення числа на вектор», яке можна разглядати ще як бінарне відношення «подовження вектора» у деяке число разів:

```
vector operator*(long double &ld,vector &v) {
  vector temp=v;
  for(long i=0;i<v.getm();i++)
    v[i]*=ld;
  return temp;
}
```

Тернарне відношення «множення числа на вектор» є комутативним, тому через нього можна означити і множення вектора на число:

```
vector operator*(vector &v,long double &ld) {
  return ld*v;
}
```

Маючи визначені бінарну операцію додавання векторів та унарну отримання вектора, протилежного до даного, можна на векторній мові, не звертаючись до компонентів векторів, означити операцію віднімання як оборотну до додавання:

```
vector operator-(vector &f,vector &s) {
  return f+(-s);
}
```

При переписуванні одного вектора у інший можливі два випадки:

1. якщо розмірність обох векторів співпадає, то просто замінюємо компоненти першого вектора компонентами другого;
2. в усіх інших випадках знищуємо перший вектор та створюємо знову, використовуючи компоненти другого як дані для ініціалізації:

```
vector vector::operator=(vector &x) {
  if(m!=x.m)
  {
```

```
    delete []vec;
    m=x.m;
    vec=new long double[m];
    if(vec==NULL)
    {
      cout<<"Не вистачає пам'яті\n";
      exit(0);
    }
  }
  for(long i=0;i<m;i++)
    vec[i]=x[i];
  return *this;
}
```

Два вектора будемо вважати рівними, якщо вони мають однакові довжини і їх відповідні компоненти співпадають:

```
long operator==(vector &f,vector &s)
{
  if(f.m!=s.m)
    return 0;
  for(long i=0;i<f.m;i++)
    if(f[i]!=s[i])
      return 0;
  return 1;
}
```

Нерівність векторів означимо через рівність і операцію заперечення:

```
long operator!=(vector &f,vector &s)
{
  return !(f==s);
}
```

Потужний механізм введення/виведення C++ дозволяє у природній формі виводити вектори на будь-який пристрій відображення інформації:

```
ostream &operator<<(ostream &os,vector &x)
{
  for(long i=0;i<x.m;i++)
    os<<x[i]<<" ";
  return os;
}

istream &operator>>(istream &is,vector &x)
{
  for(long i=0;i<x.m;i++)
    is>>x[i];
  return is;
}
```

Будемо вважати, що дійсна матриця - це вектор, елементами якого є не числові об'єкти, а арифметичні вектори дійсних чисел. В зв'язку з цим між векторним та матричним класами є багато спільного, проте інколи можна спостерігати і суттєві відмінності.

Візьмемо, скажімо, класичний файловий метод зберігання даних. Якщо для вектора досить було вказати одне службове число - його довжину (розмірність), то для матриці їх потрібно вже два, причому перше з цих чисел буде визначати кількість векторів у матриці, а друге - розмірність кожного з цих векторів. У файлових операціях з векторами нам допоможе вищезазначена дія введення вектору з деякого пристрою:

```
matrix::matrix(char *f)
{
  ifstream fp=f;
  if(!fp)
  {
    cerr<<"Не можна відкрити файл "<<f<<"\n";
    exit(0);
  }
  fp>>m>>n;
  if(m<=0||n<=0)
  {
```

```
    cerr<<"Розмірність матриці некоректна\n";
    exit(0);
  }
  mtr=new vector[m];
  if(mtr==NULL)
  {
    cerr<<"Не вистачає пам'яті\n";
    exit(0);
  }
  for(long i=0;i<m;i++)
    mtr[i]=vector(n);
  for(i=0;i<m&&fp>>mtr[i];i++);
}
```

Знаючи лише розміри матриці, ми можемо вважати її нульовим елементом даного розміру і сконструювати відповідним чином:

```
matrix::matrix(long a,long b):m(a),n(b)
{
  if(m<=0||n<=0)
  {
    cerr<<"Розмірність матриці некоректна\n";
    exit(0);
  }
  mtr=new vector[m];
  if(mtr==NULL)
  {
    cerr<<"Не вистачає пам'яті\n";
    exit(0);
  }
  for(long i=0;i<m;i++)
    mtr[i]=vector(n);
}
```

Отримавши про матрицю усі можливі відомості, є сенс, сконструювавши її, ініціювати відповідними елементами:

```
matrix::matrix(long a,long b,long double *mt):
```

```
m(a),n(b)
{
  if(m<=0||n<=0)
  {
    cerr<<"Розмірність матриці некоректна\n";
    exit(0);
  }
  mtr=new vector[m];
  if(mtr==NULL)
  {
    cerr<<"Не вистачає пам'яті\n";
    exit(0);
  }
  for(long i=0,l=0;i<m;i++)
    mtr[i]=vector(n);
  for(i=0;i<m;i++)
    for(long j=0;j<n;j++)
      mtr[i][j]=mt[l++];
}
```

Обернений випадок (коли ми не знаємо ані розмірів, ані елементів матричних векторів) розв'язується досить просто - створенням матриці з одного єдиного вектора довжиною в один елемент за відомим принципом «аби було»:

```
matrix::matrix():m(1),n(1)
{
  mtr=new vector[1];
  if(mtr==NULL)
  {
    cout<<"Не вистачає пам'яті\n";
    exit(0);
  }
}
```

Знищення матриці автоматично знищує усі її вектори:

```
matrix::~matrix() {
  delete []mtr;
```

```
}
```

Як і вектори, матриці теж буває потрібно проіндексувати; результатом цієї дії, природньо, буде відповідний вектор:

```
vector &matrix::operator[](long a)
{
  static vector error;
  error[0]=MAX_LONGDOUBLE;
  if(a>=0&&a<m)
    return mtr[a];
  else
  {
    cerr<<"Індекс "<<a<<" знаходиться за межами ма-
триці\n";
    return error;
  }
}
```

Якщо в пам’яті ЕОМ вже є якась матриця, з неї можна зняти відбиток-копію:

```
matrix::matrix(matrix &ex) : m(ex.m), n(ex.n)
{
  mtr=new vector[m];
  if(mtr==NULL)
  {
    cout<<"Не вистачає пам'яті\n";
    exit(0);
  }
  for(long i=0;i<m;i++)
    mtr[i]=ex[i];
}
```

Додавання матриць як операція визначена лише у тому випадку, коли обидві матриці - доданки мають однакову кількість строк та стовбців - тоді додавання виконується почленно; наприклад:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} + \begin{pmatrix} 1 & -2 & 0 \\ 3 & -6 & -10 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 3 \\ 7 & -1 & -4 \end{pmatrix}$$

Таким чином, додавання матриць однакової розмірності зводиться до додавання відповідних векторів:

```
matrix operator+(matrix &f,matrix &s)
{
  if(f.m!=s.m||f.n!=s.n)
  {
    cout<<"Розмірність матриць не співпадає\n";
    exit(0);
  }
  matrix temp(f.m,f.n);
  for(long i=0;i<f.m;i++)
    temp[i]=f[i]+s[i];
  return temp;
}
```

Як і для векторів, означимо унарний «мінус»:

```
matrix matrix::operator-()
{
  matrix temp(m,n);
  for(long i=0;i<m;i++)
    temp[i]=-(*this)[i];
  return temp;
}
```

На відміну від скалярного добутку векторів, множення матриць є бінарною алгебраїчною операцією, проте лише для окремих видів матриць: добуток квадратних матриць існує завжди; для прямокутних матриць це поняття теж застосовується, але має сенс лише у тому випадку, коли кількість стовбців першої матриці дорівнює кількості строк другої. Зрозуміло, що на множині прямокутних матриць ця операція не є комутативною.

Множення виконується за слідуючим правилом: елемент на перетину строки j і стовбця k матриці-добутку ми отримаємо як суму почленних добутків елементів строки j першого множника на елементи стовбця k другого множника. Наприклад:

$$\begin{pmatrix} 1 & 1 & 0 \\ -1 & -2 & 1 \\ 2 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 0 & 1 & 4 \\ 1 & 3 & 2 \\ 5 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 4 & 6 \\ 3 & -9 & -15 \\ 5 & 2 & 9 \end{pmatrix}$$

```
matrix operator*(matrix &f,matrix &s)
{
  if(f.n!=s.m)
  {
    cout<<"Помножити не можна!\n";
    exit(0);
  }
  matrix temp(f.m,s.n);
  for(long j=0;j<s.n;j++)
    for(long i=0;i<f.m;i++)
      for(long k=0;k<f.n;k++)
         temp[i][j]+=f[i][k]*s[k][j];
  return temp;
}
```

Корисною є унарна операція збільшення матриці в деяке число раз:

```
matrix operator*(matrix &f,long double &s)
{
  matrix temp=f;
  for(long i=0;i<f.m;i++)
    temp[i]=temp[i]*s;
  return temp;
}

matrix operator*(long double &f,matrix &s)
{
  return s*f;
```

```
}
```

Віднімання традиційно реалізуємо через додавання та унарний мінус, використавши оборотність додавання матриць:

```
matrix operator-(matrix &f,matrix &s)
{
  return f+(-s);
}

matrix matrix::operator=(matrix &x)
{
  if(m!=x.m||n!=x.n)
  {
    delete []mtr;
    m=x.m,n=x.n;
    mtr=new vector[m];
    if(mtr==NULL)
    {
      cout<<"Не вистачає пам'яті\n";
      exit(0);
    }
  }
  for(long i=0;i<m;i++)
    mtr[i]=x[i];
  return *this;
}
```

Дуже часто буває потрібна унарна операція транспонування:

```
matrix matrix::operator~()
{
  matrix temp(n,m);
  for(long j=0;j<n;j++)
    for(long i=0;i<m;i++)
      temp[j][i]=mtr[i][j];
  return temp;
}
```

Порівнюючи матриці, ми спочатку враховуємо, чи однакової вони розмірності, а потім у разі необхідності порівнюємо вектори, що складають їх рядки:

```
long operator==(matrix &f,matrix &s)
{
  if(f.m!=s.m||f.n!=s.n)
    return 0;
  for(long i=0;i<f.m;i++)
    if(f[i]!=s[i])
      return 0;
  return 1;
}


long operator!=(matrix &f,matrix &s)
{
  return !(f==s);
}
```

За аналогією, дозволимо виводити матриці на зовнішні пристрої:

```
ostream &operator<<(ostream &os,matrix &x)
{
  for(long i=0;i<x.m;i++)
    os<<x[i]<<"\n";
  return os;
}

istream &operator>>(istream &is,matrix &x)
{
  for(long i=0;i<x.m;i++)
    is>>x[i];
  return is;
}
```

Нехай треба розв’язати СЛАР:

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

де x_k - невідомі, a_{ij} - коефіцієнти.

У матричній формі систему лінійних алгебраїчних рівнянь (СЛАР) можна представити у вигляді

A*X=B,

де **A** - матриця коефіцієнтів при невідомих, **X** - вектор-стовбчик цих самих невідомих, а **B** - вектор-стовбчик вільних членів, взятих з оберненими знаками.

Якщо система має розв'язок, то він буде таким:

$$\mathbf{A^{-1}*A*X=A^{-1}*B}$$

$$\mathbf{(A^{-1}*A)*X=A^{-1}*B}$$

$$\mathbf{E*X=A^{-1}*B}$$

$$\mathbf{X=A^{-1}*B,}$$

а так як результат декількох матричних перетворень теж є перетворенням, то надалі будемо записувати його у формі

X=A%B,

тобто введемо нову операцію з таким позначенням, результатом якої буде розв'язок СЛАР.

Суттєвим моментом є вибір методу розв'язування. Для невеликих систем (≤200) можна застосовувати прямі методи (наприклад, Гаусса); для реальних розрахунків можна застосувати метод ортогоналізації [3]. Останній є дуже

ілюстративним - майже всі операції у ньому є векторними перетвореннями елементів відповідних матриць:

Метод ортогоналізації, на відміну від багатьох інших, не вимагає перевірок збіжності і суттєвих перетворень вихідної системи. Як допоміжний засіб метод використовується в інших обчислювальних алгоритмах, наприклад, у задачах наближення функцій, що задані таблично.

Сутність метода при розв'язанні СЛАР:
Перенесемо вільні члени у вихідній системі в ліву частину рівнянь і покладемо

$$x_{n+1}=1$$

Одержимо систему у вигляді

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

Суми у лівих частинах рівнянь можна інтерпретувати як скалярні добутки векторів $\overrightarrow{a_i}$ і $\vec{x}$; у цьому випадку шуканим розв'язком системи буде деякий вектор $\vec{X}$ в (n+1)-вимірному просторі, ортогональний базису, що утворений системними векторами $\overrightarrow{a_i}$. Так як сам базис у загальному випадку не ортонормований, то необхідна додаткова процедура його ортогоналізації.

Для пошука розв'язку системи додамо до векторів $\overrightarrow{a_i}$ лінейно незалежний від них вектор $\overrightarrow{a_{n+1}} = (0,0,\dots,0,1)$ і у (n+1)-вимірному векторному просторі будемо будувати його ортонормований базис.

Для поворота будь-якого вектора у просторі на кути, що забезпечують його ортогональність деякому ортонормованому базису векторів, досить відняти від нього вектори, співнапрямлені з ортами базиса і рівні за модулем проекціям вектора, що ортогоналізується, на ці орти (тобто скалярним добуткам вектора і відповідних орт).

Таким чином, обчислювальний алгоритм розв'язання СЛАР методом ортогоналізації складається з наступних процедур:

- нормуємо вектор $\overrightarrow{a_i}$ за модулем діленням його компонент на квадратний корінь з суми квадратів цих компонент;
- починаючи з другого вектора та до (n+1)-го послідовно здійснюємо ортогоналізацію з попередніми з подальшим нормуванням за модулем, повторюючи цю процедуру декілька разів (3-5) для попередження можливої нестійкості процеса ортогоналізації і підвищення точності обчислень у випадках близьких до нуля значень визначника вихідної системи;
- останню строку розширеної матриці поділимо на елемент $a_{(n+1)(n+1)}$, який повинен за визначенням бути рівний 1, отримавши таким чином шуканий розв'язок в n перших елементах.

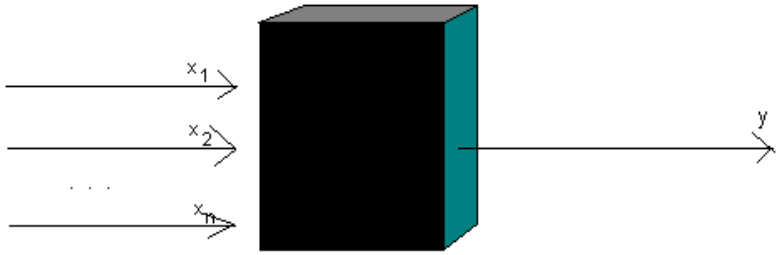
```
matrix operator%(matrix &f,matrix &s)
{
  matrix mtr2(f.m+1,f.m+1),res(f.m,1);
  for(long i=0;i<f.m;i++)
    for(long j=0;j<f.n;j++)
      mtr2[i][j]=f[i][j];
  for(i=0;i<f.m;i++)
    mtr2[i][f.m]=-s[i][0];
  mtr2[f.m][f.m]=1;

  mtr2[0]=~mtr2[0];//norm(0);
 //для впевненості повторюємо процес ортонормування
3 рази
  for(long k=0;k<3;k++)
  {
    for(long l=1;l<f.m+1;l++)
    {
      for(i=0;i<l;i++)
      {
         long double p=mtr2[l]*mtr2[i];
```

```
          for(long j=0;j<(f.m+1);j++)
            mtr2[l][j]-=p*mtr2[i][j];
        }
        mtr2[l]=~mtr2[l];//norm(l);
      }
    }

    for(i=0;i<f.m;i++)
      res[i][0]=mtr2[f.m][i]/mtr2[f.m][f.m];
    return res;
}
```

Розглянемо тепер класичну задачу МНК (методу найменших квадратів). Нехай є деякий об'єкт дослідження невідомої природи, але відомо, що при зміні декількох параметрів його реакція теж змінюється; це так званий «чорний ящик».



Відомо також, що ці зміни описуються деякою функцією. Вибравши для спрощення представлення цієї функції у вигляді полінома, ми, «освітлюючи» цей об'єкт, отримуємо з нього «сірий ящик». Якщо **X** - це дані на вході (ма-

триця), а **Y** - на виході (матриця-стовбець), то вектор-стовбець **A** коефіцієнтів регресії отримується у вигляді [5]:

$$X^T*X*A=X^T*Y$$

$$\underbrace{(X^T*X)^{-1}*(X^T*X)}_{E}*A=(X^T*X)^{-1}*(X^T*Y)$$

$$A=(X^T*X)^{-1}*(X^T*Y)$$

А це означає, що розв'язок такої задачі при застосуванні комп'ютерних типів векторів та матриць запишеться одним єдиним рядком:

matrix a=(~x*x)%(~x*y);

ЗАКЛЮЧЕННЯ

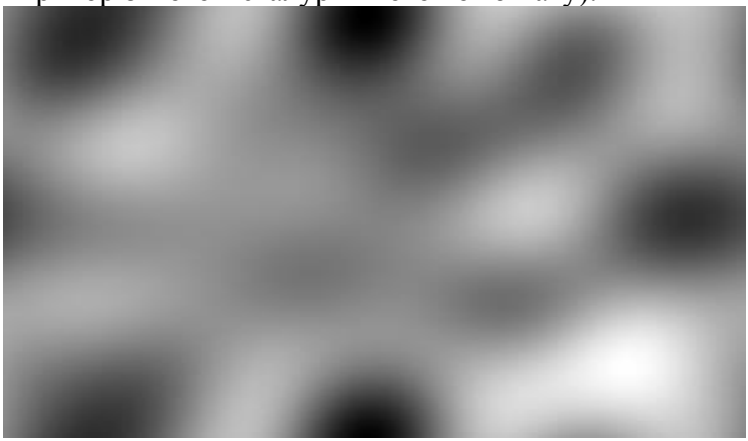
Крім розробки наведеного вище мінімального набору класів для векторно-матричних обчислень, авторами був створений розширений варіант бібліотеки класів, використання якої не тільки значно зменшує обсяг програм, але й у багатьох випадках обсяг і час обчислень завдяки оптимізованим за швидкістю алгоритмам, що має не абияке значення при великому парку застарілих комп'ютерів. Вона дозволяє маніпулювати будь-якими числовими об'єктами довільної точності, накладаючи єдине обмеження - загальний обсяг одночасово існуючих у програмі векторно-матричних об'єктів не повинен перевищувати обсягу вільного місця на резидентному диску.

Якщо, наприклад, Ви розв'язуєте секулярне рівняння, маючи при цьому вхідним параметром дійсний гамільтоніан розміром 1000x1000 чисел 10-байтової точності, то на виході одержується матриця хвильових функцій того ж розміру та енергетичний вектор, що у найпростішому випадку потребує 19,09 Мб пам'яті і близько 85 хвилин для обчислення. Наші дослідження показали, що ідентичні результати можна досягти при обсягу пам'яті у 2,5 рази менше і всього за 50 хвилин на системі 386SX-40 з 4Mb RAM шляхом неповної діагоналізації матриці із чисел 4-байтової точності з подальшим уточненням одержаних значень модифікованим методом Віландта.

Схема організації віртуальної пам'яті на резидентному диску, яку ми використовуємо, менш ефективна, ніж та, що використовується оболонкою Windows 3.1 з розширенням Win32s, тому при роботі програм у flat-моделі пам'яті її бажано відключати (до речі, використання безсегментної моделі пам'яті та перехід до 32-розрядного програмування на 15-20 відсотків збільшує швидкість обчислень,

що дозволяє розглядати нові 32-розрядні операційні системи як бажане середовище для громіздких обчислень).

Механізми бібліотеки дозволяють представити матричні дані у візуальній формі як у повно-кольоровому варіанті (16 млн. кольорів), так і у градаціях сірого, що дозволяє, наприклад, матрицю електронної щільності представити у вигляді зрозумілого і зручного для друку малюнка (див., наприклад, мал. нижче). Інші її особливості - методи для статистичної обробки експериментальних даних та регресії - дозволили застосувати бібліотеку для такої області, як управління вимірювальним обладнанням (ізотопна лабораторія Криворізького металургійного комбінату).



Література:

1. Гринчишин Я.Т. TURBO PASCAL: Чисельні методи в фізиці та математиці: Навч. посібник - Тернопіль, 1994.
2. Иванов В. В. Методы вычислений на ЭВМ - К.: Наук. думка, 1986.
3. Полищук А.П. Информатика. Персональный компьютер и его программирование (C, C++, Паскаль) - Кривой Рог, 1997.
4. Смирнов В. И. Курс высшей математики. Т.3, ч.1-М.: Наука, 1974.
5. Статистическая обработка результатов экспериментов на микро-ЭВМ и программируемых калькуляторах. - Л.: Энергоиздат, 1991.
6. Уткіна С. В., Наришкіна Л. С. Алгебра і числові системи: Навч. посібник. - К.: Вища школа, 1993