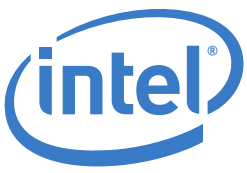
Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики
Студенческая учебно-исследовательская лаборатория Интел в МГУ

А.Н.Панкратов, М.И.Пятков, Р.К.Тетуев, Л.И.Куликова

Алгоритмы спектрального анализа с использованием библиотек Intel IPP и MKL

Москва - 2011

Методическое пособие содержит оригинальные алгоритмы для решения задач аппроксимации данных и спектральных преобразований с использованием классических ортогональных базисов. Материал содержит алгоритмы вычисления ортогональных функций высокого порядка и суммирования рядов, вычисления нулей ортогональных многочленов, линейной интерполяции, вычисления коэффициентов разложения и дифференцирования рядов. Пособие можно рассматривать как усовершенствование и развитие аналогичных процедур из пакета Numerical Recipes. Даны методические указания по оптимизации указанных процедур и примеры на языке C с использованием библиотек для высокопроизводительных вычислений Intel IPP и MKL.



Пособие подготовлено в студенческой лаборатории Интел-МГУ при поддержке компании Интел.

Содержание

Лекция 1. Вычисление функций

Алгоритм устойчивого вычисления функций Лагерра и Эрмита высокого порядка

Ортогональные многочлены удовлетворяют однородным разностным уравнениям вида

$$T_{i+1}(x) = (a_i + b_i x)T_i(x) + c_i T_{i-1}(x) \tag{1}$$

где a_i, b_i, c_i - коэффициенты, не зависящие от x. В силу однородности уравнения (1), ему удовлетворяют как полиномы, так и соответствующие им функции. Для прямого вычисления полиномов по этой формуле необходимо задать начальные условия, которые для классических ортогональных многочленов в стандартной форме имеют вид $T_0 = 1, T_{-1} = 0$. Разностное уравнение (1) связывает значения функций соседнего порядка в одной точке и является одним из способов вычисления значения всех функций базиса в одной точке x наиболее рационально, т.е. с минимальным числом операций.

Полиномы Лагерра $L_i^{\alpha}(x)$ ортогональны на полубесконечном интервале $(0,\infty)$ с весовой функцией $\rho(x) = x^{\alpha}e^{-x}$. Функции Лагерра определяются как $L_i^{\alpha}(x)x^{\alpha/2}e^{-x/2}$ и ортогональны с единичной весовой функцией. Значения функций Лагерра определяются двумя множителями - значением полинома, осциллирующего и неограниченно возрастающего, и значением корня из весовой функции, экспоненциально убывающей с ростом аргумента. При достаточно больших x эти множители в конечном машинном представлении приводят к переполнению и исчезновению порядка соответственно. Однако их произведение - функция Лагерра - является "хорошей"величиной для машинного представления. Для решения этой проблемы на каждом шаге итерационного процесса (1) вычисляемые величины $T_j(x), T_{j-1}(x)$ умножаются на величину 2^{k_j}, где k_j порядок двоичного представления $T_j(x)$. Иными словами, порядок машинного представления $T_j(x)$ обнуляется, а порядки величин $T_{j+1}(x)$ и $T_{j-1}(x)$ изменяются на величину k_j. В результате, на каждом шаге итерационного процесса (1), модифицированного описанной нормировкой, можно получить значение функции Лагерра соответствующего порядка по формуле

$$l_j(x) = \exp\left((\ln 2)\left(\sum_{i=0}^{j} k_i\right) - \frac{x}{2}\right) T_j(x)$$

Поскольку такое усовершенствование алгоритма добавляет лишь целочисленные операции к исходному алгоритму, то эффективность вычисления функций практически не снижается, а устойчивость вычислений повышается. Данный алгоритм позволяет за N шагов рекуррентного процесса получить значения всех функций с нулевого до $N-1$ порядка включительно, что обеспечивает высокое быстродействие вычислительных процедур. Ограничение на порядок вычисляемых функций в этом алгоритме не обнаружено (алгоритм протестирован для функций Лагерра и Эрмита с $N = 10000$). Реализация алгоритма для функций Лагерра и Эрмита на языке C++ представлена в листинге 1.

```
1  double lagfun(int m,double t,double alf)
2  {
3    double d=1,dd=0,tmp;
4    int c=0,cc;
5    for (int j=1;j<=m;j++){
6      tmp=(((2*j-1)+alf-t)*d-(j-1+alf)*dd)/j;
7      tmp=frexp(tmp,&cc);
8      dd=ldexp(d,-cc);
9      d=tmp;
```

```
10        c+=cc;
11      }
12      return d*exp(.5*alf*log(t)+c*log(2.0)-.5*t);
13    }
```

Листинг 1: Алгоритм вычисления функции Лагерра

Описанный алгоритм опубликован в работе [5] и является приемлемым не только для функций Лагерра, но и для функций Эрмита. Как известно [6] рекуррентный алгоритм не является устойчивым в случае ортогональных многочленов дискретного аргумента [3], к которым относятся семейства многочленов Майкснера, Кравчука, Шарлье, Хана и Чебышева дискретного аргумента.

Задания

1. Построить устойчивый алгоритм вычисления функций Хана, Майкснера, Кравчука, Шарлье, Чебышева дискретного аргумента. Указание: для устойчивого решения разностного уравнения использовать не задачу с начальными условиями, а краевую задачу.

Лекция 2. Квадратурные формулы Гаусса

Общие сведения

Рассмотрим задачу аппроксимации функции линейной комбинацией базисных функций, или рядом:

$$a(x) = \sum_{k=0}^{N-1} a_k T_k(x) \tag{2}$$

где a_k - коэффициенты разложения, а $T_k(x)$ - система базисных функций. В пространстве введено скалярное произведение:

$$(a(x), b(x)) = \int_{-1}^{1} a(x)b(x)\rho(x)dx \tag{3}$$

где $\rho(x) > 0$ - некоторая весовая функция. Коэффициенты разложения по этой системе функций удовлетворяют системе линейных уравнений

$$\sum_{i=0}^{N-1} a_i (T_i, T_j) = (f, T_j)$$

Такая система получается в результате скалярного умножения уравнения (2) на систему базисных функций $T_i(x)$. Матрица этой системы состоит из попарных скалярных произведений базисных функций и называется матрицей Грама. В случае ортогональной системы матрица Грама становится диагональной и позволяет использовать явные формулы для вычисления коэффициентов разложения:

$$a_i = \frac{(f, T_i)}{(T_i, T_i)} \tag{4}$$

В пространстве коэффициентов разложения скалярное произведение (3) приобретает следующий вид:

$$(a, b) = \sum_{i=0}^{N-1} a_i b_i \tag{5}$$

где строчными символами a и b обозначаются векторы коэффициентов разложения функций $a(x)$ и $b(x)$ соответственно. Соотношение (5) вводит скалярное произведение в пространстве коэффициентов разложения и устанавливает изоморфизм между исходным функциональным пространством и пространством коэффициентов разложения. Рассмотрим дискретный аналог скалярного произведения в пространстве функций дискретного аргумента

$$(a(x), b(x)) = \int_{-1}^{1} a(x)b(x)\rho(x)dx = \sum_{j=1}^{K} a(x_j)b(x_j)w_j \tag{6}$$

Равенство между интегралом и суммой имеет место только при определенных условиях на подынтегральную функцию и определенном выборе узлов x_i и весов w_i. Наивысшую алгебраическую точность имеют квадратуры Гаусса [2]. При специальном выборе узлов и весов (общее число параметров $2K$) можно удовлетворить условию, чтобы квадратурная формула была точна для подынтегральной функции, представляющей собой

многочлен степени не выше $2K-1$, умноженный на весовую функцию. Узлы квадратурной формулы являются нулями ортогонального многочлена степени K, соответствующего заданной весовой функции. Формула (6) обеспечивает изоморфизм между пространством линейных комбинаций (2) и пространством функций дискретного аргумента на специально выбранной сетке. При таком изоморфизме, сеточной функции, заданной в узлах полинома степени K, ставится в соответствие единственным образом или просто полином степени $K-1$, или полином степени $K-1$, умноженный на корень из весовой функции. Таким образом, смысл квадратурных формул Гаусса состоит в том, что они позволяют ортогональной системе функций непрерывного аргумента поставить в соответствие ортогональную систему функций дискретного аргумента (на неравномерной сетке).

Алгоритм

В пакете процедур Numerical Recipes [9] представлены алгоритмы вычисления узлов и весов для всех семейств классических ортогональных многочленов. В листинге 2 представлен модифицированный аналог процедуры для вычисления узлов (нулей многочлена Лежандра) и весов квадратурной формулы Гаусса с единичным весом. При этом вычисление многочлена Лежандра и его производной вынесено в отдельную процедуру.

```
1  double legfun(int m,double t,double* pder) // значение полинома Лежандра
2  {
3    double p1,p2,p3;
4    p1=1.0;
5    p2=0.0;
6    for (int j=1;j<=m;j++)
7    {
8      p3=p2;
9      p2=p1;
10     p1=((2.0*j-1.0)*t*p2-(j-1.0)*p3)/j;
11   }
12   *pder=m*(t*p1-p2)/(t*t-1.0);              // и его производной
13   return p1;
14 }
15
16 void gauleg(double t[], double w[], int n)
17 {
18   int m;
19   double z1,z2,p,pder;
20   m=(n+1)/2;
21
22   #pragma omp parallel for private (z1,z2,p,pder)
23   for (int i=1;i<=m;i++)
24   {
25
26     z2=cos(M_PI*(i-0.25)/(n+0.5));          //начальное приближение
27     do
28     {
29       p = legfun(n,z2,&pder);
30       z1=z2;
31       z2=z1-p/pder;                         //итерация Ньютона
32     } while (fabs(z2-z1) > EPS);
33
34     t[i-1]=-(t[n-i]=z2);
35     w[n-i]=w[i-1]=2.0/((1.0-z2*z2)*pder*pder); //веса квадратурной
```

```
    формулы
36  }
37 }
```

Листинг 2: Алгоритм вычисления узлов и весов квадратурной формулы Гаусса

При использовании этой процедуры возникает необходимость подбора параметра точности вычисления корней *EPS*. Этого можно избежать, если наряду с методом Ньютона также использовать метод секущих. В частности, можно заменить строки **27-32** на следующие (с добавлением определений для недостающих переменных):

```
p2 = legfun(n,z2,&pder);
do{
  z1 = z2;
  p1 = p2;
  z2 = z1 - p1/pder;
  p2 = legfun(n,z2,&pder);
}while((p1*p2>0)&&(z1!=z2));

while((p1*p2<0)&&(z=z1*p2/(p2-p1)-z2*p1/(p2-p1),is_in(z,z1,z2)))
{
  p = legfun(n,z,&pder);
  if (p1*p<0){
    z2=z;
    p2=p;
  }else{
    z1=z;
    p1=p;
  }
}
```

В отличие от исходной процедуры, в данной процедуре реализованы два метода вычисления нулей для того, чтобы комбинировать эти методы для достижения точности или эффективности процедуры. Кроме того, выход из каждого цикла осуществляется без использования параметра точности *EPS*, что позволяет автоматически вычислять корни полинома с наивысшей точностью, которую позволяет процедура вычисления значений полинома, рассмотренная в предыдущем разделе. Выход из цикла итераций по методу секущей осуществляется с использованием вспомогательной процедуры:

```
bool is_in(double z, double z1, double z2)
{
  if(z1<z2) {
    if(z1<z && z<z2)
    return true;
  }
  else if (z2<z && z<z1)
  return true;
  return false;
}
```

Распараллеливание процесса вычисления корней возможно, поскольку существует аналитическое начальное приближение для каждого корня. Распараллеливание в этом случае проведено с помощью директив стандарта OpenMP. Однако, для других ортогональных многочленов, например, Лагерра и Эрмита, начальное приближение может быть неизвестно и распараллеливание станет невозможно.

Задания

1. Доказать, что циклы для вычисления корней конечны
2. Исследовать и оптимизировать процесс вычисления корней многочленов с помощью отладчика

Лекция 3. Интерполяция

Задача интерполяции

Предположим, что мы имеем значения функции $f(x)$ на множестве точек $x_1,\ldots,x_n$, но у нас нет аналитического выражения, которое позволяет нам вычислить ее значение в произвольной точке. Если точка лежит в пределах от x_1 до x_n, то способ вычисления значения функции называют ***интерполяцией***.

Рассмотрим один из простейших способов интерполяции - линейную интерполяцию. Линейная интерполяция — интерполяция алгебраическим двучленом $P_1(x) = ax+b$ функции $f(x)$, заданной в двух точках x_0 и x_1 (Рис. 3). В случае, когда заданы значения в нескольких точках, функция заменяется кусочно-линейной функцией. Значение интерполируемой точки вычислияется следующим образом

$$f_{real} \approx f_{interp}(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0} \cdot (x - x_0) \tag{7}$$

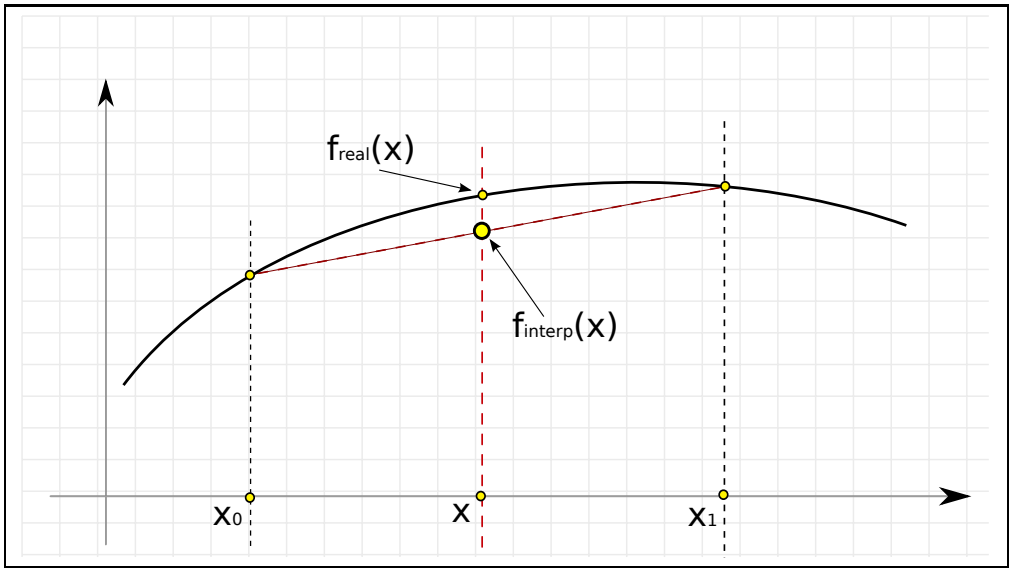


Рис. 1: Разбиение области определения на фрагменты.

```
1  /*
2  double *x1 — массив узлов интерполяции
3  double *y1 — значения функции в узлах интерполяции
4  int n      — размер массивов x1,y1
5  double *x  — массив узлов в которых требуется вычислить значение функции.
6  double *y  — массив интерполированных значений функции
7  int m      — размер массивов x,y
8  Массивы x1,x должны быть упорядочены.
9  */
10
11 void linear_interpolation(double x1[], double y1[], int n,
12                            double x[],  double y[],  int m)
13
14 {
15   int i, j = 0; // i counts points, j counts intervals
16
17   //Проверка на наличие узлов
```

```
18      if (n < 1)
19      {
20        for (i = 0; i < m; i++)
21        y[i] = 0;
22        return;
23      }
24
25      //Цикл по точкам в которых нужно интерполировать
26      for (i = 0; i < m; i++)
27      {
28        if (x[i] < x1[0]) //Если точка не входит в интервал
29        y[i] = 0;
30        else
31        {
32          //Ищем интервал в который входит точка
33          while (!((x[i] >= x1[j]) &&
34          (x[i] <= x1[j+1])) &&
35          (j < (n - 1)))
36          j++;
37          if (j < (n - 1)) //Интерполяция
38          y[i] = ((y1[j+1] - y1[j])  /
39          (x1[j+1] - x1[j])) *
40          (x[i] - x1[j]) + y1[j];
41          else
42          y[i] = 0;
43        }
44      }
45    }
```

Листинг 3: Интерполяция кусочно-линейной функцией

Векторный алгоритм линейной интерполяции

Данный алгоритм достаточно можно ускорить, используя принцип компьютерных вычислений *Single Instruction Multiple Data (SIMD)*. Инструкции, реализованные на процессорном уровне, позволяют за один такт выполнить несколько одинаковых операций. В нашем конкретном случае нас интересуют команды работы с векторами и матрицами. Векторизацию алгоритма проведем средствами *Intel Integrated Performance Primitives (Intel IPP)*. Данная библиотека предоставляет готовый набор команд для обработки различных данных(звук, изображение) и содержит нужные нам примитивы для работы с векторами.

Алгоритм, представленный в листинге 3 в том виде в каком он есть сейчас, не пригоден для векторизации или распараллеливания. Для того, чтобы это исправить, введем новую функцию ***linit***, которая возвращает для массива точек, в которых требуется узнать значение функции, индексы левого и правого узла интерполяции.

```
1   double x0[]  -- Узлы интерполяции
2   int n,       -- Количество узлов интерполяции
3   double x[] -- Точки в которых требуется интерполяция
4   int x1[],  -- Индексы левых узлов интеполяции для интервала
5   int x2[],  -- Индексы правых узлов для интервала
6   int m      -- Количество точек в которых требуется интерполяция
7
8   void linit(double x0[],
9              int n,
10             double x[],
```

```
11              int x1[],
12              int x2[],
13              int m)
14  {
15    int j = 0;
16    int i;
17
18    for (i = 0; i < m; i++)
19    {
20      if (x[i] < x0[0])
21
22      {
23        x1[i] = -1;
24        x2[i] = 0;
25      }
26      else
27      {
28        while (!( (x[i] >= x0[j]) &&
29                  (x[i] <= x0[j+1])) &&
30               (j < (n-1)))
31          j++;
32
33        if (j < (n-1))
34        {
35          x1[i] = j;
36          x2[i] = j+1;
37        }
38        else
39        {
40          x1[i] = n - 1;
41          x2[i] = -1;
42        }
43      }
44    }
45  }
```

Листинг 4: Вычисление интервала в котором лежит интерполируемая точка

После того, как мы узнали в каких интервалах расположены интерполируемые точки, процедуру интерполяции можно распараллелить по точкам или векторизовать. Векторизация средствами Intel IPP будет выглядеть следующим образом.

```
1   void lint(double x0[],
2             double y0[],
3             int n,
4             double x[],
5             double y[],
6             int x1[],
7             int x2[],
8             int m)
9   {
10    Ipp64f *xleft, *xright, *yleft, *yright; /* edge arrays */
11    Ipp64f *temp; /* Temporary array */
12
13    //Выделение памяти
14    xleft = new double[m];
15    xright = new double[m];
16    yleft = new double[m];
17    yright = new double[m];
```

```
18      temp = new double[m];
19
20      if ((xleft == NULL)  ||
21          (xright == NULL) ||
22          (yleft == NULL)  ||
23          (yright == NULL))
24        return -1;
25      int i;
26
27      //Инициализация массивов для левых и правых узлов интерполяции
28      for (i = 0; i < m; i++)
29      {
30       //Если точка вне сетки интерполяции экстраполируем ее нулем
31        if (x1[i] == -1)
32        {
33          xleft[i] = -1;;
34          xright[i] = 1;
35          yleft[i] = 0;
36          yright[i] = 0;
37        }
38        else if(x2[i] == -1)
39        {
40          xleft[i] = 1;
41          xright[i] = -1;
42          yleft[i] = 0;
43          yright[i] = 0;
44        }
45        else
46        {
47          xleft[i] = x0[x1[i]];
48          xright[i] = x0[x2[i]];
49          yleft[i] = y0[x1[i]];
50          yright[i] = y0[x2[i]];
51        }
52      }
53
54      //Formula:
55      //    y = (dy/dx)(x-x0) + y0
56      //
57
58      ippsSub_64f_A53(x, xleft, y, m);           // y = x - xleft
59      ippsSub_64f_A53(yright, yleft, temp, m); // temp = DY
60      ippsMul_64f_A53(y, temp, y, m);            // y = DY*(x-xleft)
61      ippsSub_64f_A53(xright, xleft, temp, m); // temp = DX
62      ippsDiv_64f_A53(y, temp, y, m);            // y = (DY*(x-xleft))/DX
63      ippsAdd_64f_A53(y, yleft, y, m);      //y = (DY*(x-xleft))/DX+yleft
64
65      free(xleft);
66      free(xright);
67      free(yleft);
68      free(yright);
69      free(temp);
70    }
```

Листинг 5: Векторная линейная интерполяция. IPP реализация.

Реализация для Intel MKL отличается не сильно, в частности строки **28–52** можно переписать используя команду инициализации массива которой нет в Intel IPP:

```
vdPackV(m, x0, x1, xleft);
vdPackV(m, x0, x2, xright);
vdPackV(m, y0, x1, yleft);
vdPackV(m, y0, x2, yright);
```

Код, отвечающий за интерполяцию, отличается только названием команд и их синтаксисом.

```
vdSub(m, x, xleft, y);          // y = x - xleft
vdSub(m, yright, yleft, temp); // temp = DY
vdMul(m, y, temp, y);           // y = DY*(x-xleft)
vdSub(m, xright, xleft, temp); // temp = DX
vdDiv(m, y, temp, y);           // y = (DY*(x-xleft))/DX
vdAdd(m, y, yleft, y);          // y = (DY*(x-xleft))/DX+yleft
```

Так как линейная интерполяция является частным случаем интерполяционной формулы Лагранжа и интерполяционной формулы Ньютона, приемы оптимизации, представленные в данной главе, будут в некоторой степени справедливы и для других типов интерполяции.

Задания

1. Реализовать матричный алгоритм интерполяции с использованием разреженных матриц из библиотеки MKL

2. Сравнить эффективность алгоритмов интерполяции

Лекция 4. Вычисление коэффициентов разложения

Разложение функции по коэффициентам

Некоторые системы базисных функций можно задать в явном виде, например рассмотрим как полиномы Чебышева I-го рода, которые обозначаются как $T_n(x)$, задаются подобным образом:

$$T_n(x) = \cos(n \arccos x) \tag{8}$$

Многочлены Чебышева I-го рода также могут быть заданы с помощью рекуррентного соотношения:

$$T_{n+1} = 2xT_n(x) - T_{n-1}(x) \tag{9}$$

Где первые два полинома равны $T_0(x) = 1$ и $T_1(x) = x$, соответственно. Данный многочлен, характеризуется как многочлен, который меньше всего отклоняется от нуля на интервале $[-1,1]$. Таким образом, если аппроксимировать функцию подобным полиномом, можно получить достаточно хорошее приближение уже на первых нескольких коэффициентах.

Рассмотрим задачу приближения функции, полиномами Чебышева I рода, заданной аналитическим способом на интервале $[a,b]$. Реализацию при помощи явной формулы мы рассматривать не будем, она достаточно подробно описана в книге *Numerical Recipies* [9]. Основной акцент будет сделан на рекуррентном алгоритме. Алгоритм достаточно прост и имеет широкие возможности по его оптимизации, как средствами распараллеливания, так и средствами векторизации.

Рекуррентный алгоритм

Формула вычисления коэффициентов разложения выглядит следующим образом:

$$c_j = \frac{2}{n} \sum_{i=1}^{n} f(x_i) T_j(x_i) \tag{10}$$

Начнём с реализации алгоритма разложения функции на языке высокого уровня.

```
1  /*
2    double a,b       - интервал задания функции
3    double *c        - итоговый массив коэффициентов разложения
4    int n            - размер сетки Гаусса для полиномов Чебышева
5    int m            - количество коэффициентов разложения
6    (*func)(double)  - аппроксимируемая функция
7  */
8  void chebft(double a, double b,
9              double *c, int n, int m,
10             double (*func)(double))
11 {
12   double T0, T1, Tn;
13   double *x, *y;
14
15   x = new double[n];
16   y = new double[n];
17
18   double bma = 0.5*(b-a);
19   double bpa = 0.5*(b+a);
20
21   for (int i = 0; i < n; i++)
```

```
22    {
23      x[i] = -cos(M_PIro-*(i+0.5)/n);          //Корни полинома Чебышева
24      y[i] = (*func)(x[i]*bma+bpa)/(n/2.);   //Значение функции в корнях
25    }
26
27    for(int j = 0; j < m; j++)
28      c[j] = 0.0;
29
30    for(int i = 0; i < n; i++)
31    {
32      T0 = y[i];
33      T1 = x[i]*y[i];
34      c[0]+=T0;
35      c[1]+=T1;
36      for(int j = 2; j < m; j++){
37        Tn = 2*x[i]*T1-T0;
38        T0 = T1;
39        T1 = Tn;
40        c[j]+=Tn;
41      }
42    }
43
44    delete [] y;
45    delete [] x;
46  }
```

Листинг 6: Вычисление спектральных коэффициентов путем разложения функции по базису Чебышева I рода

Рассмотрим основные моменты более подробно. Процедура **chebft** принимает на вход функцию заданную на интервале $[a,b]$ и возвращает массив спектральных коэффициентов. Строки **18-25** отвечают за перенос сетки Гаусса полиномов Чебышева с интервала $[-1,1]$ на интервал $[a,b]$. Рекуррентное выражение представлено с **30-42** строку, внешний цикл идет по точкам, внутренний по коэффициентам. Первой оптимизацией является то, что в строках **32-33** полиномы сразу же умножаются на функцию, таким образом мы избегаем дополнительного умножения во внутреннем цикле в строке **40**, ограничиваясь суммированием, т.к. полином в строке **37** уже умножен на функцию.

Векторно-рекуррентный алгоритм

Сначала необходимо выделить память для хранения значений полиномов:

```
  //double T0, T1, Tn;
  double *T0 = ippsMalloc_64f(n);
  double *T1 = ippsMalloc_64f(n);
  double *Tn = ippsMalloc_64f(n);
```

Строки **15-25**, листинга 6, отвечающие за вычисление узлов и значений функции на сетке Гаусса, оставим без изменений. Обнуляем массив коэффициентов:

```
  // for(int j = 0; j < m; j++)
  //   c[j] = 0.0;
  ippsZero_64f(c, m);
```

убираем внешний цикл по сетке Гаусса, заменяя векторными поэлементными произведениями места со скалярными:

```
  // for(int i = 0; i < n; i++)
```

```
ippsCopy_64f(y, T0, n);              //    T0 -> y[i];
ippsMul_64f_A53(x, y, T1, n);        //    T1 -> x[i]*y[i];
```

суммируем ряд:

```
ippsSum_64f(T0, n, &c[0]);           //    c[0]+=T0;
ippsSum_64f(T1, n, &c[1]);           //    c[1]+=T1;
```

Цикл по коэффициентам остается, однако рекуррентное выражение в цикле расписывается в виде виде векторных поэлементных произведений.

```
for(int j = 2; j < m; j++)
{
                                     // Tn = 2*x[i]*T1-T0;
  ippsMulC_64f(x, 2, Tn, n);         // 2*x[i] -> Tn
  ippsMul_64f_A53(Tn, T1, Tn, n);    // T1*Tn  -> Tn
  ippsSub_64f_A53(Tn, T0, Tn, n);    // Tn-T1  -> Tn

  ippsSum_64f(Tn, n, &c[j]);         // c[j]+=Tn;
  TSwp = T0;
  T0   = T1;
  T1   = Tn;
  Tn   = TSwp;
}
```

Оптимизация, предпринятая в листинге 6, позволила избежать дополнительного векторного умножения полинома на функцию, во внутреннем цикле. Используемые векторные команды описаны в официальной документации [10].

Матричный алгоритм

Одно из наиболее явных улучшений предыдущего подхода является однократное вычисление полиномов и запоминание их в матрице. После чего можно передавать вычисленную матрицу как параметр. В общем виде подход выглядит следующим образом:

$$\begin{pmatrix} T_0(x_0) & \dots & T_0(x_{n-1}) \\ \vdots & \ddots & \vdots \\ T_m(x_0) & \dots & T_m(x_{n-1}) \end{pmatrix} \times \begin{pmatrix} f(x_0) \\ \vdots \\ f(x_{n-1}) \end{pmatrix} = \begin{pmatrix} c_0 \\ \vdots \\ c_m \end{pmatrix} \tag{11}$$

Ниже представлена реализация для вычисления матрицы полиномов и отдельной функцией умножение матрицы полиномов на функцию.

```
1  void chebftpm(double *mp, int n, int m)
2  {
3    //Инициализация сетки и значений функций
4
5    ippsSet_64f(1, &mp[0], n);        // mp[0] =  1
6    ippsCopy_64f(x, &mp[n], n);       // mp[n] =  x
7
8    for(int j = 2; j < m; j++)
9    {
10     // Tn = 2*x[i]*T1-T0;
11     // 2*x[i] -> mp[j*n]
12     ippsMulC_64f(x, 2, &mp[j*n], n);
13     // T1*Tn  -> mp[j*n]
14     ippsMul_64f_A53(&mp[j*n], &mp[(j-1)*n], &mp[j*n], n);
15     // Tn-T1  -> mp[j*n]
16     ippsSub_64f_A53(&mp[j*n], &mp[(j-2)*n], &mp[j*n], n);
17   }
18
19   //Очистка памяти
20 }
```

Листинг 7: Вычисление матрицы полиномов

```
1  void chebftmat(double a, double b, double *c, int n, int m,
2                 double(*func)(double), double *mp)
3  {
4    //Инициализация сетки и значений функций
5
6    ippsZero_64f(c, m);
7
8    int mpWidth  = n;                    //
9    int mpHeight = m;                    //  Служебные переменные
10   int mpStride2 = sizeof(double);      //  для матрицы полиномов
11   int mpStride1 = n*sizeof(double);    //
12
13   int yWidth  = 1;                     //
14   int yHeight = n;                     //  Служебные переменные
15   int yStride2 = sizeof(double);       //  для массива значений функции
16   int yStride1 = sizeof(double);       //
17
18   int cStride2 = sizeof(double);       //  Служебные переменные
19   int cStride1 = sizeof(double);       //  для массива коэффициентов
20
```

```
21      //mp[i][j] * y[j] -> c[j]
22      ippmMul_mm_64f(mp, mpStride1, mpStride2, mpWidth, mpHeight,
23                     y,  yStride1,  yStride2,  yWidth,  yHeight,
24                     c,  cStride1,  cStride2);
25  }
```

Листинг 8: Вычисление коэффициентов разложения

Положительные стороны матричного подхода:

1. Вычисления выполняются заранее
2. Возможность композиции с оператором линейной интерполяции с одной сетки на другую
3. Задача сводится к линейной алгебре

Отрицательные стороны матричного подхода:

1. Интенсивное потребление памяти
2. Накладные расходы на доступ к памяти при работе на SMP-системах

Векторно-рекуррентный алгоритм с фиксированной глубиной векторизации

Последняя модификация алгоритма направлена на эффективное использование процессорного кэша. Суть идеи заключается в том, чтобы разбить область определения на p частей размером l каждая, как показано на Рис. [2], то можно записать формулу получения коэффициентов разложения следующим образом:

$$c_j = \frac{2}{n}\sum_{i=1}^{n} f(x_i)T_j(x_i) \Rightarrow \frac{2}{n}\sum_{k=0}^{p-1}\sum_{i=k\cdot l}^{(k+1)\cdot l} f(x_i)T_j(x_i) \tag{12}$$

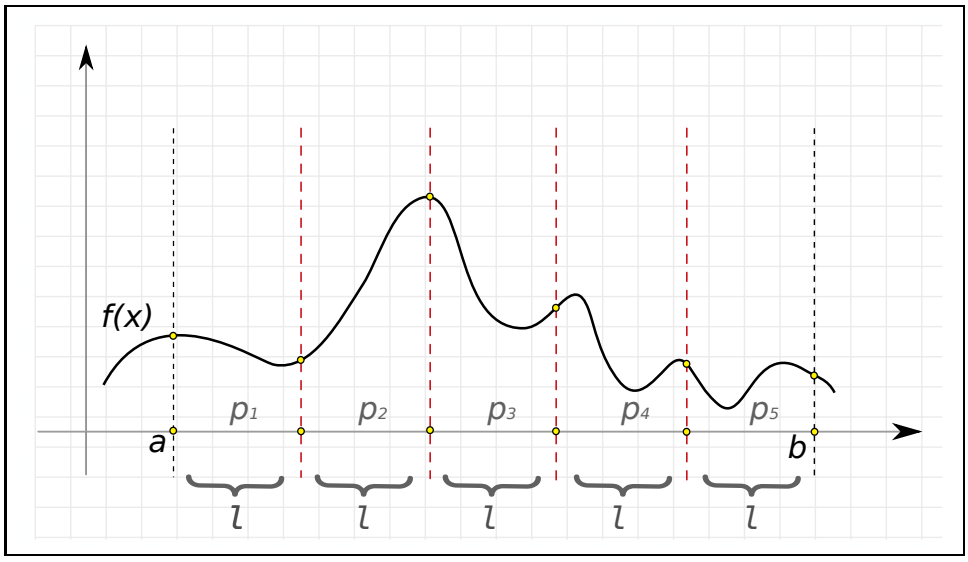


Рис. 2: Разбиение области определения на фрагменты.

Несмотря на то, что мы добавили еще один цикл по p, теперь мы можем подобрать размер фрагмента l таким образом, чтобы он полностью помещался в кэш. В коде, представленном в листинге 9, количество фрагментов p задаются константой, поэтому при

разных значениях n необходимо подбирать значение p, чтобы l приблизительно соответствовало некоторому размеру.

```
1  void chebftippvs(double a, double b,
2                   double *c, int n, int m,
3                   double (*func)(double))
4  {
5    int p       = 20;  //Колво- фрагментов
6    int l       = n/p; //Размер фрагмента
7
8    //Выделение памяти
9
10   double c0tmp, c1tmp;  //Переменные для промежуточных значений сумм
11
12   //Инициализация сетки и значений функции
13
14   for (int i = 0; i < p; i++)                 //Цикл по векторам
15   {
16
17     ippsCopy_64f(y+i*l, T0, l);               //  T0 = y[i];
18     ippsMul_64f_A53(x+i*l, y+i*l, T1, l);     //  T1 = x[i]*y[i];
19     ippsSum_64f(T0, l, &c0tmp);
20     ippsSum_64f(T1, l, &c1tmp);
21     c[0]+=c0tmp;                              //  c[0]+=T0;
22     c[1]+=c1tmp;                              //  c[1]+=T1;
23
24     for(int j = 2; j < m; j++)
25     {
26                                               // Tn =  2*x[i]*T1-T0;
27       ippsMulC_64f(x+i*l, 2, Tn, l);          // 2*x[i] -> Tn
28       ippsMul_64f_A53(Tn, T1, Tn, l);         // T1*Tn  -> Tn
29       ippsSub_64f_A53(Tn, T0, Tn, l);         // Tn-T1  -> Tn
30       ippsSum_64f(Tn, l, &c0tmp);             //
31       c[j]+=c0tmp;                            // c[j]+=Tn
32       TSwp = T0;
33       T0   = T1;
34       T1   = Tn;
35       Tn   = TSwp;
36     }
37 }
38
39 //Очистка памяти
40 }
```

Листинг 9: Векторно-рекуррентный алгоритм с фиксированной глубиной аппроксимации. Длина фрагмента аппроксимации l равна $\frac{n}{20}$.

Комбинирование различных техник оптимизации позволяет найти компромисс между переносимостью, наличием/отсутсвием параллельности и простотой алгоритма. В данной главе были рассмотрены различные алгоритмы вычисления спектральных коэффициентов путем разложения функции, заданной аналитически, по базису Чебышева I-го рода. Для других ортогональных базисов все будет аналогично, за исключением того, что для большинства из них нет явных формул для получения корней полиномов.

Задания

1. Найти оптимальный размер фрагмента сетки для наиболее эффективного выпол-

нения алгоритма с фиксированной глубиной векторизации

2. Исследовать алгоритмы при многопоточном вычислении коэффициентов разложения

Лекция 5. Суммирование рядов

Алгоритмы

Восстановление функции происходит по следующему выражению:

$$f(x) \approx \left[\sum_{k=0}^{N-1} c_k T_k(x)\right] - \frac{1}{2}c_0 \tag{13}$$

Метод восстановления функции мы получаем из рекуррентного алгоритма вычисления коэффициентов разложения. Этот алгоритм, представленный на листинге 10, также можно векторизовать, объединяя точки сетки, в которых необходимо вычислить сумму ряда. Но в отличие от алгоритма вычисления коэффициента разложения, этот алгоритм можно распараллелить с помощью директив стандарта OpenMP.

```
1  void chebev(double a,double b,
2              double *c, int m,
3              double *x, double *y, int n)
4  {
5    double T0, T1, Tn;
6    #pragma omp parallel for private(T0, T1, Tn)
7    for(int i = 0; i < n; i++)
8    {
9      t=(2.0*x[i]-a-b)/(b-a);
10     T0 = 1.0;
11     T1 = t;
12     y[i]=c[0]*0.5+c[1]*T1;
13     for(int j = 2; j < m; j++)
14     {
15       Tn = 2*t*T1-T0;
16       T0 = T1;
17       T1 = Tn;
18       y[i]+=c[j]*Tn;
19     }
20   }
21 }
```

Листинг 10: Рекуррентная процедура суммирования ряда

Существует элегантная схема вычисления суммы ряда, члены которого связаны рекуррентным соотношением, которая называется рекуррентное соотношение Кленшоу , она достаточно подробно описана в книге *Numerical Recipies* [9]. Для перехода к рекуррентному соотношению Кленшоу в листинге 10 достаточно заменить соответствующие строки на строки из листинга 11.

```
10     d = 0.0;
11     dd = 0.0;
12     for (int j = m-1; j > 0; j--)
13     {
14       sv=d;
15       d=2.0*t*d-dd+c[j];
16       dd=sv;
17     }
18     y[i]= t*d-dd+0.5*c[0];
```

Листинг 11: Суммирование ряда рекуррентным соотношением Кленшоу

Задания

1. Сравнить рекуррентный алгоритм и алгоритм Кленшоу
2. Реализовать матричный алгоритм суммирования
3. Реализовать рекуррентный алгоритм суммирования с использованием средств векторизации

Лекция 6. Дифференцирование рядов

Общие сведения

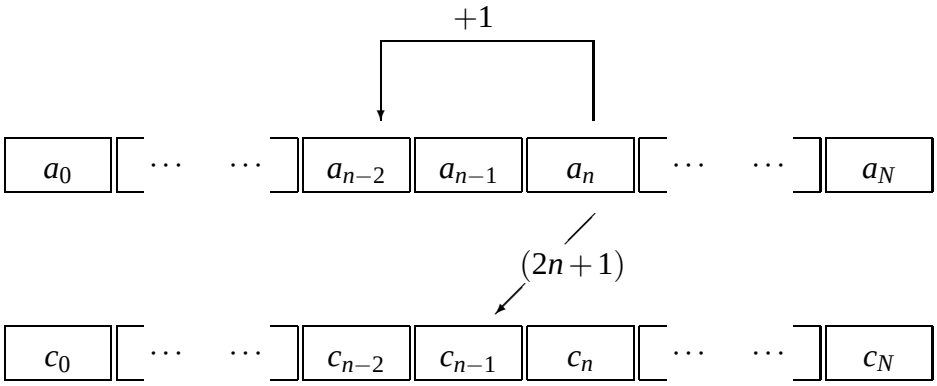
Разложение исследуемых сигналов в обобщенный ряд Фурье по ортогональным полиномам приводит не только к качественной аппроксимации сигнала (что показано в предыдущих лекциях), но и позволяет вычислять его производные.

$$f'(t) = \left(\sum_{i=0}^{N} a_i \varphi_i(t)\right)' = \sum_{i=0}^{N} a_i \varphi_i'(t) = \sum_{i=0}^{N-1} c_i \varphi_i(t)$$

Для вычисления производной необходимо выразить коэффициенты разложения производной c_i через коэффициенты исходного сигнала a_i. Общая схема алгоритма для преобразования коэффициентов разложения предложена в работе [7].

Алгоритм

Рассмотрим схему алгоритма преобразования коэффициентов разложения при дифференцировании на примере полиномов Лежандра.



Для этого подставим формулу для дифференцирования полинома Лежандра

$$P'_{i+1}(t) = P'_{i-1}(t) + (2i+1)P_i(t)$$

в старший член ряда, таким образом понижая степень ряда

$$f'(t) = \sum_{i=0}^{N+1} a_i P'_i(t) = (2N+1)a_{N+1}P_N(t) + a_{N+1}P'_{N-1}(t) + \sum_{i=0}^{N} a_i P'_i(t)$$

Из этого следует алгоритм преобразования коэффициентов ряда, который может быть представлен в виде двух независимых вычислительных этапов, получивших название ***каскада*** и ***диффузии*** спектра. На этапе каскада каждый коэффициент разложении исходного сигнала увеличивается на значение коэффициента с номером c_{j+2}. При этом ход вычислений производится в направлении от старшего коэффициента к младшему. Второй шаг процедуры, диффузия, сводится к сдвигу всего спектра на одну позицию влево и к умножению значений спектра на $(2i+1)$.

Изложенная процедура вычисления приведена в таблице 1 и листинге 12.

```
1  void legder(double a,double b,double c[],double cder[],int n)
2  {
3    double con=2.0/(b-a);
4
5    //инициализация
6    for(int i = 0; i < n; i++)
7          cder[i] = c[i];
8
9     //каскад
10    for(int i = n-2; i > 0; i--)
11          cder[i-1] += cder[i+1];
12
13    //диффузия
14    for(int i = 0; i < n-1; i++)
15          cder[i] = cder[i+1]*(2.0*i+1.0);
16    cder[n-1]=0;
17
18    //масштабирование
19    for (int i=0;i<n-1;i++)
20          cder[i] *= con;
21 }
```

Листинг 12: Каскадный алгоритм дифференцирования ряда Лежандра

С другой стороны, можно вывести формулы для явного преобразования коэффициентов

$$c_n = \sum_{k=0}^{N} \sqrt{(2k+1)(2n+1)}\, a_k$$

которые приводят к матричному алгоритму, представленному на листинге 13.

```
1 void legder1(flop a,flop b,flop alf,flop bet,flop c[],flop cder[],int n)
2  {
3     flop con=2.0/(b-a);
4     for (int i=0;i<n-1;i++)
5     {
6           cder[i]=0;
7           for (int j=i+1; j<n; j+=2 )
8             cder[i]+=sqrt((2.0*j+1)*(2.0*i+1))*c[j];
9           cder[i] *= con;
10    }
11 }
```

Листинг 13: Матричный алгоритм дифференцирования ряда Лежандра

Задания

1. Реализовать матричный алгоритм дифференцирования с использованием IPP или MKL
2. Сравнить эффективность каскадного и матричного алгоритмов дифференцирования
3. Вывести и реализовать алгоритм дифференцирования ряда Чебышева I-го рода

Список литературы

1. Ф.Ф.Дедус, С.А.Махортых, М.Н.Устинин, А.Ф.Дедус Обобщенный спектрально-аналитический метод обработки информационных массивов. Задачи анализа изображений и распознавания образов. - М.: Машиностроение, 1999.

2. А.Ф.Никифоров, В.Б.Уваров Специальные функции математической физики. - М.:Наука, 1984

3. А.Ф.Никифоров, С.К.Суслов, В.Б.Уваров Классические ортогональные полиномы дискретной переменной - М.: Наука, 1985

4. Ф.Ф.Дедус, Л.И.Куликова, А.Н.Панкратов, Р.К.Тетуев Классические ортогональные базисы в задачах аналитического описания и обработки информационных сигналов. - Москва: Издательский отдел Факультета ВМиК МГУ имени М.В.Ломоносова, 2004.

5. А.Н.Панкратов, А.К.Британков Обобщенный спектрально-аналитический метод: проблемы описания цифровых данных семействами ортогональных полиномов// Вестник Нижегородского государственного университета им. Н.И.Лобачевского. Серия Радиофизика. Вып. 1(2). - Н.Новгород: Изд-во ННГУ, 2004, с.5-14

6. А.Ф.Никифоров, М.В.Скачков Методы вычисления q-полиномов// Математическое моделирование, 2001, том 13, номер 8, с.85-94

7. Р.К.Тетуев, Ф.Ф.Дедус Классические ортогональные базисы и их применение - М: 11-й Формат, препринт, 2007

8. П.К.Суетин Классические ортогональные многочлены. - М.:Наука, 1979

9. W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery Numerical Recipes in C. The Art of Scientific Computing. - Cambridge University Press, 1992

10. Intel® Integrated Performance Primitives for Intel® Architecture Reference Manual.