SP++ Guide

张明

2010-05

目录

1 SP++导引

1.1 SP++概述

SP++(Signal Processing in C++)是一个信号处理的 C++开源库，提供了线性代数与信号处理领域中常用的一些基本算法的 C++实现。

SP++全部以 C++类模板方法实现，所有程序都以头文件形式组织而成，所以不需要用户进行本地编译，只要将相关的头文件包含在项目中即可使用。对于一般的函数直接在头文件中定义；对于类”XXX”，分为两部分，即声明文件”XXX.h”和实现”XXX-impl.h”。所有的函数和类均位于名字空间”itlab”中，所以使用本库时要进行命名空间声明：”using namespace itlab”。

SP++最早发表于开源中国社区上，博客地址为：http://my.oschina.net/zmjerry，在Google Code上也可以找到该库，地址为：http://code.google.com/p/tspl/，因为名字冲突，故项目名称为TSPL(Template Signal Processing Library)。

1.2 SP++安装

1 将 SP++压缩包解压到某一路径下，如 D:\Program Files\SP++。

2 打开CodeBlocks，在Settings->Compler and Debugger / Search directories / Compiler 中加入D:\Program Files\SP++\include，如图 1.1所示。

3 同第二步，在Linker中加入D:\Program Files\SP++\lib，如图 1.2所示。

4 在 Settings->Compler and Debugger / Link libraries 中加入 D:\Program Files\SP++\libfftw3f.a与D:\Program Files\SP++\libfftw3.a，如图 1.3所示。

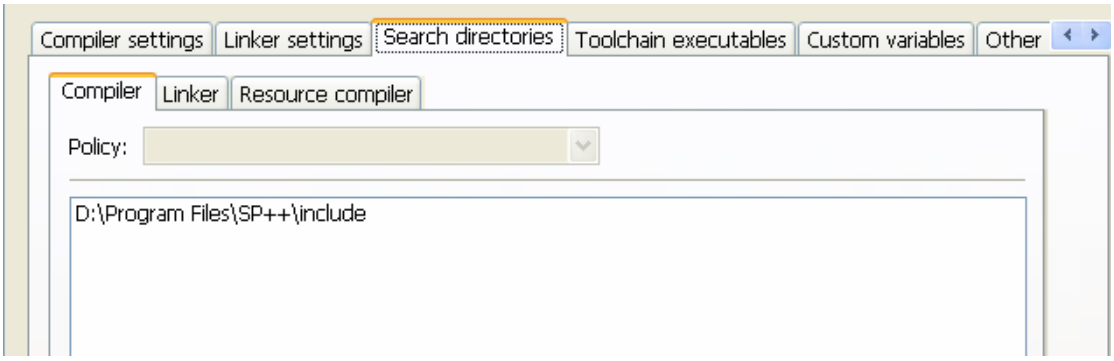

图 1.1

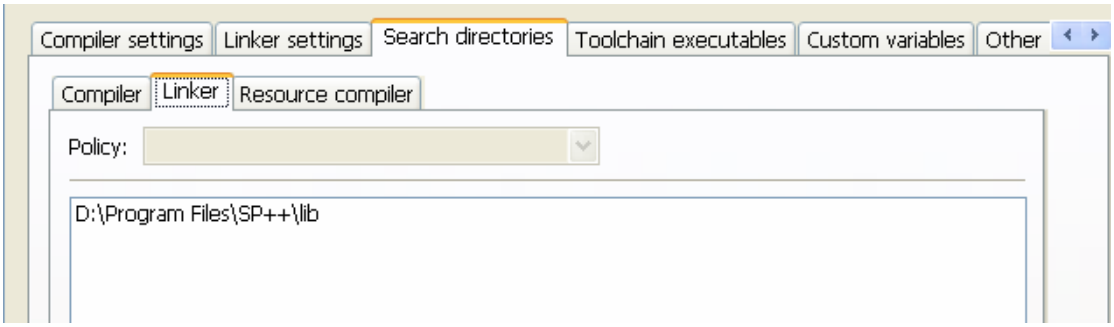

图 1.2

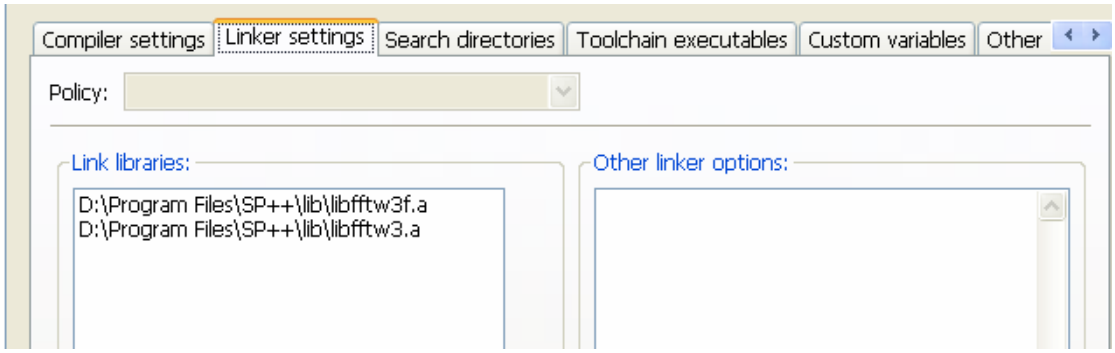

图 1.3

5 在 CodeBlocks 中建立 C++ 工程，例如 SP++Test，代码如下：

```
1.  /*************************************************************************
2.   *                              fir_test.cpp
3.   *
4.   * FIR class testing.
5.   *
6.   * Zhang Ming, 2010-03
7.   *************************************************************************/
8.
9.
10.  #define BOUNDS_CHECK
11.
12.  #include <iostream>
13.  #include <fir.h>
14.
15.
16.  using namespace std;
17.  using namespace itlab;
18.
19.
20.  int main()
21.  {
22.      string  wType = "Hamming";
```

```
23.      string  fType = "bandstop";
24.      double  fs = 1000,
25.              fpass1 = 100,
26.              apass1 = -3,
27.              fstop1 = 200,
28.              fstop2 = 300,
29.              astop1 = -20,
30.              fpass2 = 400,
31.              apass2 = -3;
32.      FIR fir( fType, wType );
33.      fir.setParams( fs, fpass1, apass1, fstop1, fstop2, astop1, fpass2, apass2 );
34.
35.      fir.design();
36.      fir.dispInfo();
37.
38.      cout << endl;
39.      return 0;
40.  }
```

运行结果为：

```
1.                      Filter selectivity      :  bandstop
2.                      Window type             :  Hamming
3.                      Sampling Frequency (Hz) :  1000
4.                      Lower passband frequency (Hz) :  100
5.                      Lower passband gain      (dB) :  -3
6.                      Lower stopband frequency (Hz) :  200
7.                      Upper stopband frequency (Hz) :  300
8.                      Stopband gain            (dB) :  -20
9.                      Upper passband frequency (Hz) :  400
10.                     Upper passband gain      (dB) :  -3
11.
12.
13.                                 Filter Coefficients
14.
15.     N    [     N + 0           N + 1           N + 2           N + 3     ]
16.    ===   ================================================================
17.     0    -3.85192764e-003  8.42472981e-003  3.11153775e-003 -2.03549729e-003
18.     4    -3.55185234e-002  2.30171390e-002 -1.41331145e-001  2.61755439e-001
19.     8     2.88881794e-002  3.56158158e-001  3.56158158e-001  2.88881794e-002
20.    12     2.61755439e-001 -1.41331145e-001  2.30171390e-002 -3.55185234e-002
21.    16    -2.03549729e-003  3.11153775e-003  8.42472981e-003 -3.85192764e-003
22.
23.
24.           =================== Edge Frequency Response ====================
```

```
25.             Mag(fp1) = -0.85449456(dB)       Mag(fp2) = -0.80635855(dB)
26.             Mag(fs1) = -21.347821(dB)       Mag(fs2) = -21.392825(dB)
```

2 向量类模板

2.1 基本向量类

向量类模板Vector<Type>是专门为线性代数中向量而设计的一个模板类，包含了向量的构造与析构，见表 2.1；向量类的基本属性提取，见表 2.2；向量常用算法的运算符重载，见表 2.3；以及一些其它常用函数，见表 2.4。

为了表示方便，表中所涉及的变量均未注明类型，可根据变量名称粗略地判断其类型，如习惯用 v 来表示向量，用 a 来表示数组，用 n 来表示元素个数等等。具体的函数声明与定义可以参见”vector.h”和”vector-impl.h”。

表 2.1 向量类的构造与析构函数

Operation	Effect
Vector<Type> v	创建一个空向量
Vecto<Type> v1(v2)	创建向量 v2 的拷贝 v1
Vector<Type> v(x)	创建常数向量
Vector<Type> v(n,a)	通过数组创建向量
v.~Vectro<Type>()	销毁向量并释放空间

表 2.2 向量类的属性获取

Operation	Effect
v.begin()	获取第一个元素的迭代器
v.end	获取最后一个元素下一位的迭代器
v.dim()	获取向量的维数
v.size()	获取向量的大小
v.resize(n)	重新设置向量的大小
v.reverse()	返回向量的反转

表 2.3 向量类中重载的运算符

Operation	Effect
v1 = v2	向量对向量赋值
v = x	常数对向量赋值
v[i]	0 偏移下标访问
v(i)	1 偏移下标访问
-v	全部元素取反
v + x	向量与常数之和
x + v	常数与向量之和
v += x	向量自身加常数

v - x	向量与常数之差
x - v	常数与向量之差
v -= x	向量自身减常数
v * x	向量与常数之积
x * v	常数与向量之积
v *= x	向量自身乘常数
v / x	向量与常数之商
x / v	常数与向量之商
v /= x	向量自身除以常数
v1 + v2	向量与向量之和
v1 += v2	向量自身加向量
v1 - v2	向量与向量之差
v1 -= v2	向量自身减向量
v1 * v2	向量与向量之积(逐元素)
v1 *= v2	向量自身乘向量(逐元素)
v1 / v2	向量与向量之商(逐元素)
v1 /= v2	向量自身除以向量(逐元素)
>> v	输入向量
<< v	输出向量

表 2.4　向量类的其它函数

Operation	Effect
sum(v)	向量元素之和
norm(v)	向量的 L_2 范数
dotProd(v1,v2)	向量的内积

Vector<Type>类的测试代码：

```
1.  /*****************************************************************************
2.   *                              vector_test.cpp
3.   *
4.   * Vector class testing.
5.   *
6.   * Zhang Ming, 2010-01
7.   *****************************************************************************/
8.
9.
10.  #define BOUNDS_CHECK
11.
12.  #include <complex>
13.  #include <iostream>
14.  #include <constants.h>
15.  #include <vector.h>
16.
```

```
17.
18.  using namespace std;
19.  using namespace itlab;
20.
21.
22.  const int M = 3;
23.
24.
25.  void display( const int *p, int length )
26.  {
27.      for( int i=0; i<length; ++i )
28.          cout << p[i] << "\t" ;
29.      cout << endl;
30.  }
31.
32.
33.  int main()
34.  {
35.      int k;
36.      int arrays[3] = {1,2,3};
37.
38.      Vector<int> v1( M,arrays );
39.      k = 1;
40.      Vector<int> v2( M,k );
41.
42.      Vector<int> v3( M );
43.      k = 0;
44.      v3 = k;
45.
46.      Vector<int> v4( v1 );
47.
48.      cout << "vector v1 : " << v1 << endl;
49.      cout << "vector v2 : " << v2 << endl;
50.      cout << "vector v3 : " << v3 << endl;
51.      cout << "vector v4 : " << v4 << endl;
52.
53.      display( v4, M );
54.      cout << endl;
55.
56.      v4.resize( 5 );
57.      Vector<int>::iterator itr = v4.begin();
58.      while( itr != v4.end() )
59.          *itr++ = 1;
60.      cout << "new vector v4 : " << v4 << endl;
61.      v4 = v1;
```

```
62.     cout << "new vector v4 : " << v4 << endl;
63.     cout << "reversion of vector v4 : " << v4.reverse() << endl;
64.
65.     k = 2;
66.     v3 = k+v1;
67.     cout << "v3 = k + v1 : " << v3 << endl;
68.     v3 += k;
69.     cout << "v3 += k : " << v3 << endl;
70.
71.     v3 = v1-k;
72.     cout << "v3 = v1 - k : " << v3 << endl;
73.     v3 = k-v1;
74.     cout << "v3 = k - v1 : " << v3 << endl;
75.     v3 -= k;
76.     cout << "v3 -= k : " << v3 << endl;
77.
78.     v3 = k*v1;
79.     cout << "v3 = k * v1 : " << v3 << endl;
80.     v3 *= k;
81.     cout << "v3 *= k : " << v3 << endl;
82.
83.     v3 = v1/k;
84.     cout << "v3 = v1 / k : " << v3 << endl;
85.     v3 = k/v1;
86.     cout << "v3 = k / v1 : " << v3 << endl;
87.     v3 /= k;
88.     cout << "v3 /= k : " << v3 << endl;
89.
90.     v3 = v1+v2;
91.     cout << "v3 = v1 + v2 : " << v3 << endl;
92.     v3 += v1;
93.     cout << "v3 += v1 : " << v3 << endl;
94.
95.     v3 = v1-v2;
96.     cout << "v3 = v1 - v2 : " << v3 << endl;
97.     v3 -= v1;
98.     cout << "v3 -= v1 : " << v3 << endl;
99.
100.    v3 = v1*v2;
101.    cout << "v3 = v1 * v2 : " << v3 << endl;
102.    v3 *= v1;
103.    cout << "v3 *= v1 : " << v3 << endl;
104.
105.    v3 = v1/v2;
106.    cout << "v3 = v1 / v2 : " << v3 << endl;
```

```
    v3 /= v1;
    cout << "v3 /= v1 : " << v3 << endl;

    k = dotProd( v1, v2 );
    cout << "inner product of v1 and v2 : " << endl;
    cout << k << endl << endl;

    cout << "L2 norm of v3 : ";
    cout << norm( v3 ) << endl << endl;

    complex<double> z = -1.0;
    Vector< complex<double> > v( M );
    v[0] = polar( 1.0,PI/4 );
    v[1] = polar( 1.0,PI );
    v[2] = complex<double>( 1.0,-1.0 );
    Vector< complex<double> > u = v*z;

    cout << "complex vector v : " << v << endl;
    cout << "complex vector u = -v : " << u << endl;

    Vector< Vector<double> > v2d1( M );
    for( int i=0; i<M; ++i )
    {
        v2d1[i].resize( M );
        for( int j=0; j<M; ++j )
            v2d1[i][j] = double( i+j );
    }

    cout << "two dimention vector v2d1 : " << endl;
    Vector< Vector<double> >::const_iterator itrD2 = v2d1.begin();
    int rowNum = 0;
    while( itrD2 != v2d1.end() )
        cout << "the " << rowNum++ << "th row : " << *itrD2++ << endl;

    Vector< Vector<double> > v2d2 = v2d1+v2d1;
    cout << "two dimention vector v2d2 = v2d1 + v2d1 : " << v2d2 << endl;

    return 0;
}
```

运行结果：

```
vector v1 : size: 3 by 1
1
2
```

```
3

vector v2 : size: 3 by 1
1
1
1

vector v3 : size: 3 by 1
0
0
0

vector v4 : size: 3 by 1
1
2
3

1       2       3

new vector v4 : size: 5 by 1
1
1
1
1
1

new vector v4 : size: 3 by 1
1
2
3

reversion of vector v4 : size: 3 by 1
3
2
1

v3 = k + v1 : size: 3 by 1
3
4
5

v3 += k : size: 3 by 1
5
6
7
```

```
49.
50.  v3 = v1 - k : size: 3 by 1
51.  -1
52.  0
53.  1
54.
55.  v3 = k - v1 : size: 3 by 1
56.  1
57.  0
58.  -1
59.
60.  v3 -= k : size: 3 by 1
61.  -1
62.  -2
63.  -3
64.
65.  v3 = k * v1 : size: 3 by 1
66.  2
67.  4
68.  6
69.
70.  v3 *= k : size: 3 by 1
71.  4
72.  8
73.  12
74.
75.  v3 = v1 / k : size: 3 by 1
76.  0
77.  1
78.  1
79.
80.  v3 = k / v1 : size: 3 by 1
81.  2
82.  1
83.  0
84.
85.  v3 /= k : size: 3 by 1
86.  1
87.  0
88.  0
89.
90.  v3 = v1 + v2 : size: 3 by 1
91.  2
92.  3
93.  4
```

```
94.
95.  v3 += v1 : size: 3 by 1
96.  3
97.  5
98.  7
99.
100. v3 = v1 - v2 : size: 3 by 1
101. 0
102. 1
103. 2
104.
105. v3 -= v1 : size: 3 by 1
106. -1
107. -1
108. -1
109.
110. v3 = v1 * v2 : size: 3 by 1
111. 1
112. 2
113. 3
114.
115. v3 *= v1 : size: 3 by 1
116. 1
117. 4
118. 9
119.
120. v3 = v1 / v2 : size: 3 by 1
121. 1
122. 2
123. 3
124.
125. v3 /= v1 : size: 3 by 1
126. 1
127. 1
128. 1
129.
130. inner product of v1 and v2 :
131. 6
132.
133. L2 norm of v3 : 1
134.
135. complex vector v : size: 3 by 1
136. (0.707107,0.707107)
137. (-1,1.22465e-016)
138. (1,-1)
```

```
139.
140. complex vector u = -v : size: 3 by 1
141. (-0.707107,-0.707107)
142. (1,-1.22465e-016)
143. (-1,1)
144.
145. two dimention vector v2d1 :
146. the 0th row : size: 3 by 1
147. 0
148. 1
149. 2
150.
151. the 1th row : size: 3 by 1
152. 1
153. 2
154. 3
155.
156. the 2th row : size: 3 by 1
157. 2
158. 3
159. 4
160.
161. two dimention vector v2d2 = v2d1 + v2d1 : size: 3 by 1
162. size: 3 by 1
163. 0
164. 2
165. 4
166.
167. size: 3 by 1
168. 2
169. 4
170. 6
171.
172. size: 3 by 1
173. 4
174. 6
175. 8
```

2.2 常用函数的向量版本

为了便于数值计算，将一些信号处理中经常使用的数学函数进行了重载，使之能够运用于向量和矩阵对象（有关矩阵的内容见下一章），如表 2.5所示。

表 2.5　常用函数的向量与矩阵版本

Operation	Effect
vabs(rv)	实向量的绝对值
vabs(cv)	复向量的绝对值
mabs(rA)	实矩阵的绝对值
mabs(cA)	复矩阵的绝对值
vsin(v)	向量的正弦函数
vcos(v)	向量的余弦函数
vexp(v)	向量的指数函数
vpow(v)	向量的幂函数
gauss(v,u,r)	向量的 Gauss 函数
vreal(cv)	复向量的实部
vimag(cv)	复向量的虚部
mreal(cA)	复矩阵的实部
mimag(cA)	复矩阵的虚部

常用向量函数的测试代码：

```
/*****************************************************************************
 *                              mathfunc_test.cpp
 *
 * Math functions for vector and matrix testing.
 *
 * Zhang Ming, 2010-01
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <utilities.h>
#include <mathfunc.h>


using namespace std;
using namespace itlab;


int main()
{

    int     N = 11;
    float   a = 0,
            b = 2*PI;
```

```
27.     Vector<float> x = linspace( a, b, N );
28.     cout << x << endl;
29.
30.     cout << "Sine function of x : " << vsin(x) << endl;
31.     cout << "Cosine function of x : "<< vcos(x) << endl;
32.     cout << "Exponential function of x : " << vexp(x) << endl;
33.     a = 2.0;
34.     cout << "Power function of x : " << vpow(x,a) << endl;
35.
36.     Vector< complex<float> > vc(N);
37.     for( int i=0; i<N; ++i )
38.         vc[i] = complex<float>( sin(x[i]), cos(x[i]) );
39.     cout << "Complex vector vc : " << vc << endl;
40.     cout << "Real part of vc : " << vreal(vc) << endl;
41.     cout << "Imaginary part of vc : " << vimag(vc) << endl;
42.     cout << "Absolute part of vc : " << vabs(vc) << endl;
43.
44.     N = 21;
45.     a = -5;
46.     b = 5;
47.     x = linspace( a, b, N );
48.     cout << "Absolute function of x : " << vabs( x ) << endl;
49.     a = 0.0;
50.     b = 1.0;
51.     cout << "The standard normal distribution : "
52.          << gauss( x, a, b ) << endl;
53.
54.     return 0;
55. }
```

运行结果：

```
1.  size: 11 by 1
2.  0
3.  0.628319
4.  1.25664
5.  1.88496
6.  2.51327
7.  3.14159
8.  3.76991
9.  4.39823
10. 5.02655
11. 5.65487
12. 6.28319
13.
```

```
Sine function of x : size: 11 by 1
0
0.587785
0.951057
0.951056
0.587785
-8.74228e-008
-0.587785
-0.951056
-0.951056
-0.587785
1.74846e-007

Cosine function of x : size: 11 by 1
1
0.809017
0.309017
-0.309017
-0.809017
-1
-0.809017
-0.309017
0.309017
0.809017
1

Exponential function of x : size: 11 by 1
1
1.87446
3.51359
6.58606
12.3453
23.1407
43.3762
81.3068
152.406
285.679
535.492

Power function of x : size: 11 by 1
0
0.394784
1.57914
3.55306
6.31655
```

```
9.86961
14.2122
19.3444
25.2662
31.9775
39.4784

Complex vector vc : size: 11 by 1
(0,1)
(0.587785,0.809017)
(0.951057,0.309017)
(0.951056,-0.309017)
(0.587785,-0.809017)
(-8.74228e-008,-1)
(-0.587785,-0.809017)
(-0.951056,-0.309017)
(-0.951056,0.309017)
(-0.587785,0.809017)
(1.74846e-007,1)

Real part of vc : size: 11 by 1
0
0.587785
0.951057
0.951056
0.587785
-8.74228e-008
-0.587785
-0.951056
-0.951056
-0.587785
1.74846e-007

Imaginary part of vc : size: 11 by 1
1
0.809017
0.309017
-0.309017
-0.809017
-1
-0.809017
-0.309017
0.309017
0.809017
1
```

```
104.
105. Absolute part of vc : size: 11 by 1
106. 1
107. 1
108. 1
109. 1
110. 1
111. 1
112. 1
113. 1
114. 1
115. 1
116. 1
117.
118. Absolute function of x : size: 21 by 1
119. 5
120. 4.5
121. 4
122. 3.5
123. 3
124. 2.5
125. 2
126. 1.5
127. 1
128. 0.5
129. 0
130. 0.5
131. 1
132. 1.5
133. 2
134. 2.5
135. 3
136. 3.5
137. 4
138. 4.5
139. 5
140.
141. The standard normal distribution : size: 21 by 1
142. 1.48672e-006
143. 1.59837e-005
144. 0.00013383
145. 0.000872683
146. 0.00443185
147. 0.0175283
148. 0.053991
```

```
0.129518
0.241971
0.352065
0.398942
0.352065
0.241971
0.129518
0.053991
0.0175283
0.00443185
0.000872683
0.00013383
1.59837e-005
1.48672e-006
```

2.3 Matlab 中的常用函数

Matlab提供了非常丰富的工具箱，其中包含了众多领域中的常用函数，表 2.6列出了信号处理中经常使用的一些函数的C++实现。

表 2.6 Matlab 中常用函数

Operation	Effect
mod(m,n)	取 m 对 n 的非负模值
ceil(m,n)	对 m/n 进行向上取整
isPower2(n)	判断 n 是否为 2 的于整次幂
fastLog2(n)	计算 $\log_2(n)$的快速算法
linspace(a,b,n)	生成等差数组
flip(v)	向量反转
dyadUp(v,oe)	二进上采样
dyadDown(v,oe)	二进下采样
wkeep(v,n,first)	提取向量的部分元素
wkeep(v,n,direct)	提取向量的部分元素
wextend(v,n,direct,mode)	向量延拓
conv(s,f)	卷积运算的时域实现

Matlab 中常用函数的测试代码：

```
/*****************************************************************************
 *                              utilities_test.cpp
 *
 * Utilities class testing.
 *
```

```
 * Zhang Ming, 2010-01
 ******************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <string>
#include <utilities.h>


using namespace std;
using namespace itlab;


const int N = 5;


int main()
{

    Vector<int> v1(N);
    for( int i=1; i<=v1.dim(); i++ )
        v1(i) = i;
    cout << "vector v1 : " << v1 << endl;

    Vector<int> v2(N);
    for( int i=1; i<=v2.dim(); ++i )
        v2(i) = i+N;
    cout << "vector v2 : " << v2 << endl;

    int N = 11;
    double a = 0;
    double b = 1.0;
    Vector<double> x = linspace( a, b, N );
    cout << N << " points linearly spaced from 0 to 1.0"
         << x << endl;

    cout << "Flipping vector v1 from left to right : " << flip(v1) << endl;
    cout << "Dyadic upsampling of vector v1 by zeros at the even position : "
         << dyadUp( v1,0 ) << endl;
    cout << "Dyadic upsampling of vector v1 by zeros at the odd position : "
         << dyadUp( v1,1 ) << endl;
    cout << "Dyadic downsampling of vector v1 by zeros at the even position : "
         << dyadDown( v1,0 ) << endl;
```

```
51.       cout << "Dyadic downsampling of vector v1 by zeros at the odd position : "
52.            << dyadDown( v1,1 ) << endl;
53.
54.       int n = 2;
55.       string dire = "left";
56.       string mode = "zpd";
57.       cout << "Extending vector v1 in left dirction by zeros padding : "
58.            << wextend( v1,n,dire,mode ) << endl;
59.       mode = "ppd";
60.       cout << "Extending vector v1 in left dirction by periodic mode : "
61.            << wextend( v1,n,dire,mode ) << endl;
62.       mode = "sym";
63.       cout << "Extending vector v1 in left dirction by symetric mode : "
64.            << wextend( v1,n,dire,mode ) << endl;
65.
66.       dire = "right";
67.       mode = "zpd";
68.       cout << "Extending vector v1 in right dirction by zeros padding : "
69.            << wextend( v1,n,dire,mode ) << endl;
70.       mode = "ppd";
71.       cout << "Extending vector v1 in right dirction by periodic mode : "
72.            << wextend( v1,n,dire,mode ) << endl;
73.       mode = "sym";
74.       cout << "Extending vector v1 in right dirction by symetric mode : "
75.            << wextend( v1,n,dire,mode ) << endl;
76.
77.       dire = "both";
78.       mode = "zpd";
79.       cout << "Extending vector v1 in both dirction by zeros padding : "
80.            << wextend( v1,n,dire,mode ) << endl;
81.       mode = "ppd";
82.       cout << "Extending vector v1 in both dirction by periodic mode : "
83.            << wextend( v1,n,dire,mode ) << endl;
84.       mode = "sym";
85.       cout << "Extending vector v1 in both dirction by symetric mode : "
86.            << wextend( v1,n,dire,mode ) << endl;
87.
88.       cout << "Keeping the center part of vector v1 : " << wkeep( v1,3,"center" ) << endl;
89.       cout << "Keeping the left part of vector v1 : " << wkeep( v1,3,"left" ) << endl;
90.       cout << "Keeping the right part of vector v1 : " << wkeep( v1,3,"right" ) << endl;
91.       cout << "Keeping the first(2) to first + L(3) elements of vector v1 : "
92.            << wkeep( v1,3,2 ) << endl;
93.
94.       cout << "The convolution of vector v1 and v2 : " << conv( v1,v2 ) << endl;
95.
```

```
96.       cout << "The modulus of 2 divided by 5 is " << mod(2,5) << "." << endl;
97.       cout << "The modulus of -1 divided by 5 is " << mod(-1,5) << "." << endl;
98.       cout << endl;
99.       cout << "The nearest integer >= 10/2 is " << ceil(10,2) << "." << endl;
100.      cout << "The nearest integer >= 10/3 is " << ceil(10,3) << "." << endl;
101.
102.      cout << endl;
103.      cout << "The numbers can be represented by the integer power of 2 "
104.           << "from 0 to 1000 are : " << endl;
105.      for( int i=0; i<1000; ++i )
106.          if( isPower2(i) )
107.              cout << fastLog2(i) << " = log2(" << i << ")" << endl;
108.
109.      return 0;
110. }
```

运行结果：

```
1.   vector v1 : size: 5 by 1
2.   1
3.   2
4.   3
5.   4
6.   5
7.
8.   vector v2 : size: 5 by 1
9.   6
10.  7
11.  8
12.  9
13.  10
14.
15.  11 points linearly spaced from 0 to 1.0size: 11 by 1
16.  0
17.  0.1
18.  0.2
19.  0.3
20.  0.4
21.  0.5
22.  0.6
23.  0.7
24.  0.8
25.  0.9
26.  1
27.
```

```
Flipping vector v1 from left to right : size: 5 by 1
5
4
3
2
1

Dyadic upsampling of vector v1 by zeros at the even position : size: 11 by 1
0
1
0
2
0
3
0
4
0
5
0

Dyadic upsampling of vector v1 by zeros at the odd position : size: 9 by 1
1
0
2
0
3
0
4
0
5

Dyadic downsampling of vector v1 by zeros at the even position : size: 3 by 1
1
3
5

Dyadic downsampling of vector v1 by zeros at the odd position : size: 2 by 1
2
4

Extending vector v1 in left dirction by zeros padding : size: 7 by 1
0
0
1
2
```

```
3
4
5

Extending vector v1 in left dirction by periodic mode : size: 7 by 1
4
5
1
2
3
4
5

Extending vector v1 in left dirction by symetric mode : size: 7 by 1
2
1
1
2
3
4
5

Extending vector v1 in right dirction by zeros padding : size: 7 by 1
1
2
3
4
5
0
0

Extending vector v1 in right dirction by periodic mode : size: 7 by 1
1
2
3
4
5
1
2

Extending vector v1 in right dirction by symetric mode : size: 7 by 1
1
2
3
4
```

```
118.  5
119.  5
120.  4
121.
122.  Extending vector v1 in both dirction by zeros padding : size: 9 by 1
123.  0
124.  0
125.  1
126.  2
127.  3
128.  4
129.  5
130.  0
131.  0
132.
133.  Extending vector v1 in both dirction by periodic mode : size: 9 by 1
134.  4
135.  5
136.  1
137.  2
138.  3
139.  4
140.  5
141.  1
142.  2
143.
144.  Extending vector v1 in both dirction by symetric mode : size: 9 by 1
145.  2
146.  1
147.  1
148.  2
149.  3
150.  4
151.  5
152.  5
153.  4
154.
155.  Keeping the center part of vector v1 : size: 3 by 1
156.  2
157.  3
158.  4
159.
160.  Keeping the left part of vector v1 : size: 3 by 1
161.  1
162.  2
```

```
3

Keeping the right part of vector v1 : size: 3 by 1
3
4
5

Keeping the first(2) to first + L(3) elements of vector v1 : size: 3 by 1
2
3
4

The convolution of vector v1 and v2 : size: 9 by 1
6
19
40
70
110
114
106
85
50

The modulus of 2 divided by 5 is 2.
The modulus of -1 divided by 5 is 4.

The nearest integer >= 10/2 is 5.
The nearest integer >= 10/3 is 4.

The numbers can be represented by the integer power of 2 from 0 to 1000 are :
0 = log2(1)
1 = log2(2)
2 = log2(4)
3 = log2(8)
4 = log2(16)
5 = log2(32)
6 = log2(64)
7 = log2(128)
8 = log2(256)
9 = log2(512)
```

2.4 概率统计中的常用函数

“statistics.h”头文件中提供了随机信号处理中的一些常用算法，如随机变量的均值、方差、偏度、峭度以及概率密度函数等等，同时包含了向量的极大值、极小值和中值等，详见表 2.7。

表 2.7 概率统计中的常用函数

Operation	Effect
min(v)	取向量的最小值
max(v)	取向量的最大值
mid(v)	取向量的中值
mean(v)	取随机变量的均值
var(v)	取随机变量的方差
stdVar(v)	取随机变量的标准差
standard (v)	随机变量的标准化
skew(v)	取随机变量的偏度
kurt(v)	取随机变量的峭度
pdf(v,lambda)	估计随机变量的概率密度函数

概率统计中常用函数的测试代码：

```
/*****************************************************************************
 *                              statistics_test.cpp
 *
 * Statistics routines testing.
 *
 * Zhang Ming, 2010-03
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <statistics.h>


using namespace std;
using namespace itlab;


int main()
{
    Vector<double> xn(9);
    xn[0] = 1.2366;     xn[1] = -0.6313;    xn[2] = -2.3252;
```

```
    xn[3] = -1.2316;    xn[4] = 1.0556;     xn[5] = -0.1132;
    xn[6] = 0.3792;     xn[7] = 0.9442;     xn[8] = -2.1204;

    cout << "The minmum maximum and median value of the sequence." << endl;
    cout << min(xn) << endl << max(xn) << endl << mid(xn) <<endl << endl;
    cout << "The mean, variance and standard variance of the sequence." << endl;
    cout << mean(xn) << endl << var(xn) << endl << stdVar(xn) << endl << endl;

    Vector<double> yn = xn - mean(xn);
    cout << "The skew and kurtosis of the sequence." << endl;
    cout << skew(yn) << endl << kurt(yn) << endl << endl;

    cout << "The PDF of the sequence." << endl;
    cout << pdf(yn) << endl;

    return 0;
}
```

运行结果：

```
The minmum maximum and median value of the sequence.
-2.3252
1.2366
-0.1132

The mean, variance and standard variance of the sequence.
-0.311789
1.82934
1.35253

The skew and kurtosis of the sequence.
-0.282956
-1.67782

The PDF of the sequence.
size: 5 by 1
0.000512486
0.134665
0.333867
0.42622
0.104735
```

3 矩阵类模板

3.1 基本矩阵类

矩阵类模板Vector<Type>是专门为线性代数中矩阵而设计的一个模板类，包含了矩阵的构造与析构，见表 3.1；矩阵类的基本属性提取，见表 3.2；矩阵常用算法的运算符重载，见表 3.3；以及一些其它常用函数，如矩阵之间的（转置）乘法，矩阵与向量间的（转置）乘法，矩阵的转置与逆等等，见表 3.4。具体的函数声明与定义可以参见”matrix.h”和” matrix -impl.h”。

表 3.1 矩阵类的构造与析构函数

Operation	Effect
Matrix<Type> m	创建一个空矩阵
Matrix<Type> m2(m1)	创建矩阵 m2 的拷贝 m1
Matrix<Type> m(x)	创建常数矩阵
Matrix<Type> m(r,c,a)	通过数组创建矩阵
m.~Matrix<Type> ()	销毁矩阵并释放空间

表 3.2 矩阵类的属性获取

Operation	Effect
m.size()	矩阵元素总个数
m.dim(d)	矩阵第 d 维的维数
m.rows()	矩阵的行数
m.cols()	矩阵的列数
m.resize(r,c)	重新分配矩阵大小
m.getRow(r)	获取矩阵的第 r 行
m.getColumn(c)	获取矩阵的第 c 列
m.setRow(v)	设置矩阵的第 r 行
m.setColumn(v)	设置矩阵的第 c 列

表 3.3 矩阵类中重载的运算符

Operation	Effect
A1 = A2	矩阵对矩阵赋值
A = x	常数对矩阵赋值
A[i][j]	0 偏移下标访问
A(i,j)	1 偏移下标访问
-A	全部元素取反
A + x	矩阵与常数之和

x + A	矩阵与向量之和
A += x	矩阵自身加常数
A - x	矩阵与常数之差
x - A	矩阵与向量之差
A -= x	矩阵自身减常数
A * x	矩阵与常数之积
x * A	矩阵与向量之积
A *= x	矩阵自身乘常数
A / x	矩阵与常数之商
x / A	矩阵与向量之商
A /= x	矩阵自身除以常数
A1 + A2	矩阵与矩阵之和
A1 += A2	矩阵自身加矩阵
A1 - A2	矩阵与矩阵之差
A1 -= A2	矩阵自身减矩阵
A1 * A2	矩阵与矩阵之积(逐元素)
A1 *= A2	矩阵自身乘矩阵(逐元素)
A1 / A2	矩阵与矩阵之商(逐元素)
A1 /= A2	矩阵自身除以矩阵(逐元素)
>> A	输入矩阵
<< A	输出矩阵

表 3.4 矩阵类的其它函数

Operation	Effect
prod(A,B,C)	优化版本的矩阵乘法
prod(A,B)	矩阵乘以矩阵
prod(A,b,c)	优化版本的矩阵乘以向量
prod(A,b)	矩阵乘以向量
tranProd(A,B)	矩阵乘以矩阵的转置
tranProd(A,b)	矩阵乘以向量的转置
tranProd(b,c)	向量乘以向量的转置
transpose(A)	矩阵的转置
diag(A)	提取矩阵的对角线
eye(n,x)	产生 n 阶单位矩阵
inverse(A)	矩阵求逆
norm(A)	矩阵的 Frobenius 范数

Matrix<Type>的测试代码：

```
1.  /*****************************************************************************
2.   *                              matrix_test.cpp
3.   *
4.   * Matrix class testing.
5.   *
```

```
6.    * Zhang Ming, 2010-01
7.    ************************************************************************/
8.
9.
10.  #define BOUNDS_CHECK
11.
12.  #include <iostream>
13.  #include <matrix.h>
14.
15.
16.  using namespace std;
17.  using namespace itlab;
18.
19.
20.  const int M = 3;
21.  const int N = 3;
22.
23.
24.  int main()
25.  {
26.      double x;
27.
28.      Matrix<double> m1;
29.      m1.resize( M, N );
30.      x = 1.0;
31.      m1 = x;
32.      cout << "matrix m1 : " << m1 << endl;
33.
34.      x = 2.0;
35.      Matrix<double> m2( M, N, x );
36.      cout << "matrix m2 : " << m2 << endl;
37.
38.      Matrix<double> m3 = m1;
39.      cout << "matrix m3 : " << m3 << endl;
40.      m3.resize( 3, 4 );
41.      for( int i=1; i<=3; ++i )
42.          for( int j=1; j<=4; ++j )
43.              m3(i,j) = double(i*j);
44.
45.      int row = m3.dim(1);
46.      int column = m3.dim(2);
47.      cout << "the row number of new matrix m3 : " << row << endl;
48.      cout << "the column number of new matrix m3 : " << column << endl;
49.      cout << "new matrix m3 : " << m3 << endl;
50.
```

```
51.     cout << "the diagonal matrix of m3 : " << diag( m3 ) << endl;
52.     cout << "the transpose matrix of m3 : " << transpose( m3 ) << endl;
53.
54.     cout << endl << endl << "\t\t\t\tmatrix-scalar operand" << endl << endl;
55.     cout << "scalar x = " << x << endl;
56.     cout << "m1 + x : " << m1+x << endl;
57.     cout << "x + m1 : " << x+m1 << endl;
58.     m1 += x;
59.     cout << "m1 += x : " << m1 << endl;
60.     cout << "m1 - x : " << m1-x << endl;
61.     cout << "x - m1 : " << x-m1 << endl;
62.     m1 -= x;
63.     cout << "m1 -= x : " << m1 << endl;
64.     cout << "m1 * x : " << m1*x << endl;
65.     cout << "x * m1 : " << x*m1 << endl;
66.     m1 *= x;
67.     cout << "m1 *= x : " << m1 << endl;
68.     cout << "m1 / x : " << m1/x << endl;
69.     cout << "x / m1 : " << x/m1 << endl;
70.     m1 /= x;
71.     cout << "m1 /= x : " << m1 << endl;
72.
73.     cout << endl << endl << "\t\t\t\telementwise matrix-matrix operand" << endl << endl;
74.     cout << "m1 + m2 : " << m1 + m2 << endl;
75.     m1 += m2;
76.     cout << "m1 += m2 : " << m1 << endl;
77.     cout << "m1 - m2 : " << m1-m2 << endl;
78.     m1 -= m2;
79.     cout << "m1 -= m2 : " << m1 << endl;
80.     cout << "m1 * m2 : " << m1*m2 << endl;
81.     m1 *= m2;
82.     cout << "m1 *= m2 : " << m1 << endl;
83.     cout << "m1 / m2 : " << m1/m2 << endl;
84.     m1 /= m2;
85.     cout << "m1 /= m2 : " << m1 << endl;
86.
87.     cout << endl << endl << "\t\t\t\tmatrix-matrix operand" << endl << endl;
88.     cout << "m1 * m2 : " << prod(m1, m2) << endl;
89.     cout << "m1' * m2 : " << tranProd(m1, m2) << endl;
90.
91.     cout << endl << endl << "\t\t\t\tmatrix-vector operand" << endl << endl;
92.     Vector<double> v1( 3, 2 );
93.     cout << "vector v1 : " << v1 << endl;
94.     cout << "m1 * v1 : " << prod(m1, v1) << endl;
95.     cout << "m1' * v1 : " << tranProd(m1, v1) << endl;
```

```
96.
97.     cout << "Please input a sqare matrix mtr." << endl;
98.     Matrix<double> mtr;
99.     cin >> mtr;
100.    cout << "the inverse matrix of mtr : " << inverse( mtr ) << endl;
101.
102.    Matrix<int> m4( 4, 5 );
103.    Vector<int> v2(5);
104.    v2[0] = 1;  v2[1] = 2;  v2[2] = 3;  v2[3] = 4;  v2[4] = 5;
105.    for( int i=1; i<=4; ++i )
106.        m4.setRow( i*v2, i );
107.
108.    cout << "matrix m4 : " << m4 << endl;
109.    cout << "column vectors of m4 : " << endl;
110.    for( int j=1; j<=5; ++j )
111.        cout << "the " << j << "th column" << m4.getColumn(j) << endl;
112.
113.    v2.resize(4);
114.    v2[0] = 1;  v2[1] = 2;  v2[2] = 3;  v2[3] = 4;
115.    for( int j=1; j<=5; ++j )
116.        m4.setColumn( j*v2, j );
117.
118.    cout << "row vectors of m4 : " << endl;
119.    for( int i=1; i<=4; ++i )
120.        cout << "the " << i << "th row" << m4.getRow(i) << endl;
121.
122.    return 0;
123. }
```

运行结果：

```
1.  matrix m1 : size: 3 by 3
2.  1 1 1
3.  1 1 1
4.  1 1 1
5.
6.  matrix m2 : size: 3 by 3
7.  2 2 2
8.  2 2 2
9.  2 2 2
10.
11. matrix m3 : size: 3 by 3
12. 1 1 1
13. 1 1 1
14. 1 1 1
```

```
15.
16.  the row number of new matrix m3 : 3
17.  the column number of new matrix m3 : 4
18.  new matrix m3 : size: 3 by 4
19.  1 2 3 4
20.  2 4 6 8
21.  3 6 9 12
22.
23.  the diagonal matrix of m3 : size: 3 by 1
24.  1
25.  4
26.  9
27.
28.  the transpose matrix of m3 : size: 4 by 3
29.  1 2 3
30.  2 4 6
31.  3 6 9
32.  4 8 12
33.
34.
35.                                  matrix-scalar operand
36.
37.  scalar x = 2
38.  m1 + x : size: 3 by 3
39.  3 3 3
40.  3 3 3
41.  3 3 3
42.
43.  x + m1 : size: 3 by 3
44.  3 3 3
45.  3 3 3
46.  3 3 3
47.
48.  m1 += x : size: 3 by 3
49.  3 3 3
50.  3 3 3
51.  3 3 3
52.
53.  m1 - x : size: 3 by 3
54.  1 1 1
55.  1 1 1
56.  1 1 1
57.
58.  x - m1 : size: 3 by 3
59.  -1 -1 -1
```

```
-1 -1 -1
-1 -1 -1

m1 -= x : size: 3 by 3
1 1 1
1 1 1
1 1 1

m1 * x : size: 3 by 3
2 2 2
2 2 2
2 2 2

x * m1 : size: 3 by 3
2 2 2
2 2 2
2 2 2

m1 *= x : size: 3 by 3
2 2 2
2 2 2
2 2 2

m1 / x : size: 3 by 3
1 1 1
1 1 1
1 1 1

x / m1 : size: 3 by 3
1 1 1
1 1 1
1 1 1

m1 /= x : size: 3 by 3
1 1 1
1 1 1
1 1 1


                                elementwise matrix-matrix operand

m1 + m2 : size: 3 by 3
3 3 3
3 3 3
3 3 3
```

```
105.
106. m1 += m2 : size: 3 by 3
107. 3 3 3
108. 3 3 3
109. 3 3 3
110.
111. m1 - m2 : size: 3 by 3
112. 1 1 1
113. 1 1 1
114. 1 1 1
115.
116. m1 -= m2 : size: 3 by 3
117. 1 1 1
118. 1 1 1
119. 1 1 1
120.
121. m1 * m2 : size: 3 by 3
122. 2 2 2
123. 2 2 2
124. 2 2 2
125.
126. m1 *= m2 : size: 3 by 3
127. 2 2 2
128. 2 2 2
129. 2 2 2
130.
131. m1 / m2 : size: 3 by 3
132. 1 1 1
133. 1 1 1
134. 1 1 1
135.
136. m1 /= m2 : size: 3 by 3
137. 1 1 1
138. 1 1 1
139. 1 1 1
140.
141.
142.                                 matrix-matrix operand
143.
144. m1 * m2 : size: 3 by 3
145. 6 6 6
146. 6 6 6
147. 6 6 6
148.
149. m1' * m2 : size: 3 by 3
```

```
6 6 6
6 6 6
6 6 6


                                matrix-vector operand

vector v1 : size: 3 by 1
2
2
2

m1 * v1 : size: 3 by 1
6
6
6

m1' * v1 : size: 3 by 1
6
6
6

Please input a sqare matrix mtr.
3 3
-4 1 3
4 -1 -2
-3 1 1
the inverse matrix of mtr : size: 3 by 3
1 1 2
4 2 5
-0 1 1

matrix m4 : size: 4 by 5
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20

column vectors of m4 :
the 1th columnsize: 4 by 1
1
2
3
4

```

```
the 2th columnsize: 4 by 1
2
4
6
8

the 3th columnsize: 4 by 1
3
6
9
12

the 4th columnsize: 4 by 1
4
8
12
16

the 5th columnsize: 4 by 1
5
10
15
20

row vectors of m4 :
the 1th rowsize: 5 by 1
1
2
3
4
5

the 2th rowsize: 5 by 1
2
4
6
8
10

the 3th rowsize: 5 by 1
3
6
9
12
15
```

```
240.
241.  the 4th rowsize: 5 by 1
242.  4
243.  8
244.  12
245.  16
246.  20
```

3.2 Cholesky 分解

对于一个对称正定矩阵A，可以表示为A=L*L'，其中L为下三角矩阵，这就是对称正定矩阵的Cholesky分解。利用该分解可以解方程组，也可以求解矩阵方程，Cholesky类的具体功能见表 3.5。

表 3.5 Cholesky 分解

Operation	Effect
Cholesky<Real> cho	建立 Cholesky 类
cho.~Cholesky<Real>()	Cholesky 析构函数
cho.isSpd()	判断矩阵是否对称正定
cho.dec(A)	对矩阵 A 进行 Cholesky 分解
cho.getL()	获取下三角矩阵 L
cho.solve(b)	解方程组 Ax = b
cho.solve(B)	矩阵方程 AX = B

Cholesky<Real>类的测试代码：

```
1.   /*****************************************************************************
2.    *                              cholesky_test.cpp
3.    *
4.    * Cholesky class testing.
5.    *
6.    * Zhang Ming, 2010-01
7.    *****************************************************************************/
8.
9.
10.  #define BOUNDS_CHECK
11.
12.  #include <iostream>
13.  #include <cholesky.h>
14.
15.
16.  using namespace std;
```

```
17.  using namespace itlab;
18.
19.
20.  const int N = 5;
21.
22.
23.  int main()
24.  {
25.      Matrix<double> A(N,N), L(N,N), B(N,N);
26.      Vector<double> b(N);
27.
28.      for( int i=1; i<N+1; ++i )
29.      {
30.          for( int j=1; j<N+1; ++j )
31.              if( i == j )
32.              {
33.                  A(i,i) = i;
34.                  B(i,i) = 1;
35.              }
36.              else
37.              {
38.                  if( i < j )
39.                      A(i,j) = i;
40.                  else
41.                      A(i,j) = j;
42.
43.                  B(i,j) = 0;
44.              }
45.
46.          b(i) = i*(i+1)/2 + i*(N-i);
47.      }
48.
49.      Cholesky<double> cho;
50.      cho.dec(A);
51.      if( cho.isSpd() )
52.          L = cho.getL();
53.      else
54.          cout << "Factorization was not complete." << endl;
55.
56.      cout << "The original matrix A : " << A << endl;
57.      cout << "The lower triangular matrix L is : " << L << endl;
58.
59.      Vector<double> x = cho.solve(b);
60.      cout << "The constant vector b : " << b << endl;
61.      cout << "The solution of Ax = b : " << x << endl;
```

```
62.
63.     Matrix<double> C = cho.solve(B);
64.     cout << "The invse matrix of A : " << C << endl;
65.
66.     return 0;
67. }
```

运行结果：

```
1.  The original matrix A : size: 5 by 5
2.  1 1 1 1 1
3.  1 2 2 2 2
4.  1 2 3 3 3
5.  1 2 3 4 4
6.  1 2 3 4 5
7.
8.  The lower triangular matrix L is : size: 5 by 5
9.  1 0 0 0 0
10. 1 1 0 0 0
11. 1 1 1 0 0
12. 1 1 1 1 0
13. 1 1 1 1 1
14.
15. The constant vector b : size: 5 by 1
16. 5
17. 9
18. 12
19. 14
20. 15
21.
22. The solution of Ax = b : size: 5 by 1
23. 1
24. 1
25. 1
26. 1
27. 1
28.
29. The invse matrix of A : size: 5 by 5
30. 2 -1 0 0 0
31. -1 2 -1 0 0
32. 0 -1 2 -1 0
33. 0 0 -1 2 -1
34. 0 0 0 -1 1
```

3.3 LU 分解

对于一个n阶矩阵A，LU分解将其分为一个下三角阵L与一个上三角阵U的乘积，通过借助矩阵的LU分解来求解矩阵的行列式，解方程组以及解矩阵方程等，详见表3.6。对于行数与列数不等的矩阵，LU同样可以分解成功，具体见”lud.h”头文件中的有关说明。

表 3.6 LU 分解

Operation	Effect
LUD<Real> lu	建立 LUD 类
lu.~LUD<Real>()	LUD 析构函数
lu.dec(A)	对矩阵 A 进行三角分解
lu.getL()	获取下三角矩阵 L
lu.getU()	获取上三角矩阵 U
lu.det()	计算矩阵的行列式
lu. isNonsingular()	判断矩阵是否非奇异
lu.solve(b)	解方程组 Ax = b
lu.solve(B)	矩阵方程 AX = B

LUD<Real>类测试代码：

```
/*****************************************************************************
 *                                lud_test.cpp
 *
 * LUD class testing.
 *
 * Zhang Ming, 2010-01
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <lud.h>


using namespace std;
using namespace itlab;


const int M = 3;
const int N = 3;

```

```
24.  int main()
25.  {
26.      Matrix<double> A(M,N), B(M,N);
27.      Vector<double> b(N);
28.
29.      A[0][0] = 1;    A[0][1] = 2;    A[0][2] = 1;
30.      A[1][0] = 2;    A[1][1] = 5;    A[1][2] = 4;
31.      A[2][0] = 1;    A[2][1] = 1;    A[2][2] = 0;
32.
33.      B[0][0] = 1;    B[0][1] = 0;    B[0][2] = 0;
34.      B[1][0] = 0;    B[1][1] = 1;    B[1][2] = 0;
35.      B[2][0] = 0;    B[2][1] = 0;    B[2][2] = 1;
36.
37.      b[0] = 1;   b[1] = 0;   b[2] = 1;
38.
39.      LUD<double> lu;
40.      lu.dec(A);
41.
42.      Matrix<double> L, U;
43.      lu.getL(L);
44.      lu.getU(U);
45.
46.      cout << "The original matrix A is : " << A << endl;
47.      cout << "The unit lower triangular matrix L is : " << L << endl;
48.      cout << "The upper triangular matrix U is : " << U << endl;
49.
50.      Vector<double> x = lu.solve(b);
51.      cout << "The constant vector b : " << b << endl;
52.      cout << "The solution of A * x = b : " << x << endl;
53.
54.      cout << "The invse matrix of A : " << lu.solve(B) << endl;
55.      cout << "The determinant of A : " << endl;
56.      cout << lu.det() << endl << endl;
57.
58.      return 0;
59.  }
```

运行结果：

```
1.  The original matrix A is : size: 3 by 3
2.  1 2 1
3.  2 5 4
4.  1 1 0
5.
6.  The unit lower triangular matrix L is : size: 3 by 3
```

```
1 0 0
0.5 1 0
0.5 0.333333 1

The upper triangular matrix U is : size: 3 by 3
2 5 4
0 -1.5 -2
0 0 -0.333333

The constant vector b : size: 3 by 1
1
0
1

The solution of A * x = b : size: 3 by 1
-1
2
-2

The invse matrix of A : size: 3 by 3
-4 1 3
4 -1 -2
-3 1 1

The determinant of A :
1
```

3.4 QR 分解

对于一个m行n列的矩阵A，QR分解将分主一个m阶的正交矩阵Q和一个m行n列的上三角矩阵R，满足A=Q*R，QR分解总是存在的，即使对于秩亏矩阵。利用矩阵的QR分解可以求解超定方程组的最小二乘解等，具体见表 3.7。

表 3.7 QR 分解

Operation	Effect
QRD<Real> qr	建立 LUD 类
qr.~QRD<Real>()	LUD 析构函数
qr.dec(A)	对矩阵 A 进行正交三角分解
qr.getQ()	获取正交矩阵 Q
qr.getR()	获取上三角矩阵 R
qr. isFullRank()	判断矩阵是否满秩
qr.solve(b)	方程组最小二乘解 Ax = b

qr.solve(B)	矩阵方程最小二乘解 AX = B

QRD<Real>类的测试代码：

```
/*****************************************************************************
 *                              qrd_test.cpp
 *
 * QRD class testing.
 *
 * Zhang Ming, 2010-01
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <qrd.h>


using namespace std;
using namespace itlab;


const int M = 4;
const int N = 3;


int main()
{
    Matrix<double> A(M,N);
    A[0][0] = 1;    A[0][1] = 0;    A[0][2] = 0;
    A[1][0] = 1;    A[1][1] = 2;    A[1][2] = 4;
    A[2][0] = 1;    A[2][1] = 3;    A[2][2] = 9;
    A[3][0] = 1;    A[3][1] = 3;    A[3][2] = 9;

    Matrix<double> B(M,M);
    B[0][0] = 1;    B[0][1] = 0;    B[0][2] = 0;    B[0][3] = 0;
    B[1][0] = 0;    B[1][1] = 1;    B[1][2] = 0;    B[1][3] = 0;
    B[2][0] = 0;    B[2][1] = 0;    B[2][2] = 1;    B[2][3] = 0;
    B[3][0] = 0;    B[3][1] = 0;    B[3][2] = 0;    B[3][3] = 1;

    Vector<double> b(M);
    b[0]= 1;    b[1] = 0;   b[2] = 1, b[3] = 2;

    Matrix<double> Q, R;
```

```
    QRD<double> qr;
    qr.dec(A);
    qr.getQ(Q);
    qr.getR(R);

    cout << "The original matrix A : " << A << endl;
    cout << "The orthogonal matrix Q  : " << Q << endl;
    cout << "The upper triangular matrix R : " << R << endl;

    Vector<double> x = qr.solve(b);
    cout << "The constant vector b : " << b << endl;
    cout << "The least squares solution of A * x = b : " << x << endl;

    Matrix<double> X = qr.solve(B);
    cout << "The constant matrix B : " << B << endl;
    cout << "The least squares solution of A * X = B : " << X << endl;

    return 0;
}
```

运行结果：

```
The original matrix A : size: 4 by 3
1 0 0
1 2 4
1 3 9
1 3 9

The orthogonal matrix Q  : size: 4 by 3
-0.5 0.816497 -0.288675
-0.5 0 0.866025
-0.5 -0.408248 -0.288675
-0.5 -0.408248 -0.288675

The upper triangular matrix R : size: 3 by 3
-2 -4 -11
0 -2.44949 -7.34847
0 0 -1.73205

The constant vector b : size: 4 by 1
1
0
1
2

```

```
The least squares solution of A * x = b : size: 3 by 1
1
-1.83333
0.666667

The constant matrix B : size: 4 by 4
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

The least squares solution of A * X = B : size: 3 by 4
1 4.44089e-016 -1.11022e-016 -1.11022e-016
-0.833333 1.5 -0.333333 -0.333333
0.166667 -0.5 0.166667 0.166667
```

3.5 SVD 分解

对于一个m行n列的矩阵A，QR分解将分主一个m阶的正交矩阵U和一个m行n列的对角矩阵S和一个m阶的正交矩阵V，满足A=U*S*V’，矩阵的SVD分解总是存在的。利用矩阵的SVD分解可以求矩阵的秩，矩阵的 2 范数和条件数等等，具体见表 3.8。

表 3.8 SVD 分解

Operation	Effect
SVD<Real> sv	建立 SVD 类
sv.~SVD<Real>()	SVD 析构函数
sv.dec(A)	对矩阵 A 进行奇异值分解
sv.getU()	获取左奇异向量
sv.getV()	获取右奇异向量 U
sv.getS()	获取奇异值
sv.norm2()	计算矩阵的 2 范数（最大奇异值）
sv.cond(b)	计算矩阵的条件数
sv.rank(B)	计算矩阵的秩

SVD<Real>类的测试代码：

```
/*****************************************************************************
 *                              svd_test.cpp
 *
 * SVD class testing.
 *
 * Zhang Ming, 2010-01
```

```
7.    ************************************************************************/
8.
9.
10.  #define BOUNDS_CHECK
11.
12.  #include <iostream>
13.  #include <svd.h>
14.
15.
16.  using namespace std;
17.  using namespace itlab;
18.
19.
20.  const int M = 4;
21.  const int N = 2;
22.
23.
24.  int main()
25.  {
26.      Matrix<double> A(M,N);
27.      A(1,1) = 1;     A(1,2) = 2;
28.      A(2,1) = 3;     A(2,2) = 4;
29.      A(3,1) = 5;     A(3,2) = 6;
30.      A(4,1) = 7;     A(4,2) = 8;
31.
32.      SVD<double> svd;
33.      svd.dec(A);
34.
35.      Matrix<double> U = svd.getU();
36.      Matrix<double> V = svd.getV();
37.      Vector<double> s = svd.getS();
38.
39.      Matrix<double> S(N,N);
40.      S = 0;
41.      for( int i=0; i<N; ++i )
42.          S[i][i] = s[i];
43.
44.      cout << "The original matrix A : " << A << endl;
45.      cout << "The left singular vectors U : " << U << endl;
46.      cout << "The right singular vectors V : " << V << endl;
47.      cout << "The singular values S : " << S << endl;
48.      Matrix<double> A1 = prod( U, prod(S,transpose(V)) );
49.      cout << "The norm of error matrix between A and U * S * V' : "
50.           << norm( A - A1 ) << endl << endl;
51.
```

```
52.       cout << "The rank of A : " << svd.rank() << endl << endl;
53.       cout << "The condition number of A : " << svd.cond() << endl << endl;
54.
55.       return 0;
56.   }
```

运行结果：

```
1.    The original matrix A : size: 4 by 2
2.    1 2
3.    3 4
4.    5 6
5.    7 8
6.
7.    The left singular vectors U : size: 4 by 2
8.    0.152483 0.822647
9.    0.349918 0.421375
10.   0.547354 0.0201031
11.   0.744789 -0.381169
12.
13.   The right singular vectors V : size: 2 by 2
14.   0.641423 -0.767187
15.   0.767187 0.641423
16.
17.   The singular values S : size: 2 by 2
18.   14.2691 0
19.   0 0.626828
20.
21.   The norm of error matrix between A and U * S * V' : 3.38208e-015
22.
23.   The rank of A : 2
24.
25.   The condition number of A : 22.764
```

3.6 EVD 分解

对于一个矩阵A，存在正交矩阵V和对角矩阵D，满足AV=V*D，其中V的列向量是A的特征向量，D对角线上的元素为对应特征向量的特征值（有可能是复数）更多的说明见”evd.h”。EVD<Type>类的一些常用函数见表 3.9。

表 3.9 EVD 分解

Operation	Effect

EVD<Real> ev	建立 EVD 类
ev.~SVD<Real>()	EVD 析构函数
ev.dec(A)	对矩阵 A 进行特征分解
ev.getV()	获取特征向量 V
ev.getD()	获取特征值 D
ev. getRealEigenvalues()	获取特征值的实部
ev. getImagEigenvalues()	获取特征值的虚部

EVD<Real>类的测试代码：

```
/*****************************************************************************
 *                               evd_test.cpp
 *
 * EVD class testing.
 *
 * Zhang Ming, 2010-01
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <evd.h>


using namespace std;
using namespace itlab;


const int N = 4;


int main()
{
    Matrix<double> A(N,N);

    A[0][0] = 3.0;  A[0][1] = -2.0; A[0][2] = -0.9;     A[0][3] = 0.0;
    A[1][0] = -2.0; A[1][1] = 4.0;  A[1][2] = 1.0;      A[1][3] = 0.0;
    A[2][0] = 0.0;  A[2][1] = 0.0;  A[2][2] = -1.0;     A[2][3] = 0.0;
    A[3][0] = -0.5; A[3][1] = -0.5; A[3][2] = 0.1;      A[3][3] = 1.0;

    EVD<double> eig;
    eig.dec(A);

    Matrix<double> D = eig.getD();
```

```
36.      Matrix<double> V = eig.getV();
37.
38.      cout << "The original matrix A : " << A << endl;
39.      cout << "The eigenvalue matrix D : " << D << endl;
40.      cout << "The eigenvectors V : " << V << endl;
41.      cout << "THe real part of the eigenvalues Dr: "
42.           << eig.getRealEigenvalues() << endl;
43.      cout << "The imaginary part of the eigenvalues Di: "
44.           << eig.getImagEigenvalues() << endl;
45.      cout << "The norm of error matrix between A * V and V * D : "
46.           << norm( prod(A,V)-prod(V,D) ) << endl << endl;
47.
48.      return 0;
49.  }
```

运行结果：

```
1.   The original matrix A : size: 4 by 4
2.   3 -2 -0.9 0
3.   -2 4 1 0
4.   0 0 -1 0
5.   -0.5 -0.5 0.1 1
6.
7.   The eigenvalue matrix D : size: 4 by 4
8.   5.56155 0 0 0
9.   0 1.43845 0 0
10.  0 0 1 0
11.  0 0 0 -1
12.
13.  The eigenvectors V : size: 4 by 4
14.  -0.615302 0.417676 -8.88178e-016 0.15625
15.  0.788064 0.326112 -4.44089e-016 -0.1375
16.  1.36742e-017 5.8871e-017 -1.19514e-016 1
17.  -0.0189368 -0.848207 1.88746 -0.0453125
18.
19.  THe real part of the eigenvalues Dr: size: 4 by 1
20.  5.56155
21.  1.43845
22.  1
23.  -1
24.
25.  The imaginary part of the eigenvalues Di: size: 4 by 1
26.  0
27.  0
28.  0
```

```
0

The norm of error matrix between A * V and V * D : 1.50939e-015
```

4 Fourier 分析

4.1 FFTW 的 C++接口

FFTW 是 MIT 学者用 C 语言编写的一个计算任意长度一维或多维的离散 Fourier 变换程序库，该库的计算效率非常高，得到了广泛的应用，是信号处理中必不可少的一个 C 语言库函数。

SP++对FFTW最新版本（3.2.2）中一维FFT提供了C++接口，如表 4.1所示。该表中的函数均是模板函数，对支持float和double两种数据类型，对于实信号，其频域长度为时域长度的一半取整再加 1。

表 4.1 FFTW 的 C++接口

Operation	Effect
fft(rxn,Xk)	实信号的 Fourier 变换
fft(cxn,Xk)	复信号的 Fourier 变换
ifft(Xk,rxn)	实信号的逆 Fourier 变换
ifft(Xk,cxn)	复信号的逆 Fourier 变换

测试代码：

```
/*****************************************************************************
 *                              fft_test.cpp
 *
 * FFT interface testing.
 *
 * Zhang Ming, 2010-01
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <complex>
#include <fft.h>
#include <constants.h>


using namespace std;
```

```
19.  using namespace itlab;
20.
21.
22.  const   int     N = 7;
23.  typedef float   Type;
24.
25.
26.  int main()
27.  {
28.      // complex to complex dft test...
29.      Vector< complex<Type> > xn( N );
30.      Vector< complex<Type> > yn( N );
31.      Vector< complex<Type> > Xk( N );
32.
33.      for( int i=0; i<N; ++i )
34.      {
35.          Type theta = Type( 2*PI * i / N );
36.          xn[i] = complex<Type>( cos(theta), sin(theta) );
37.      }
38.      cout << "xn:   " << xn << endl << endl;
39.
40.      fft( xn, Xk );
41.      cout << "Xk=fft(xn):   " << Xk << endl << endl;
42.      ifft( Xk, yn );
43.      cout << "yn=ifft(Xk):   " << yn << endl << endl;
44.
45.      // real to complex and complex to real dft test...
46.      Vector<Type> sn(N);
47.      Vector< complex<Type> > Sk( N/2+1 );
48.      for( int i=0; i<N; ++i )
49.      {
50.          Type theta = Type( 2*PI * i / N );
51.          sn[i] = sin(theta);
52.      }
53.      cout << "sn:   " << sn << endl;
54.
55.      fft( sn, Sk );
56.      cout << "Sk=fft(sn):   " << Sk << endl << endl;
57.      ifft( Sk, sn );
58.      cout << "sn=ifft(Sk):   " << sn << endl;
59.
60.      return 0;
61.  }
```

运行结果：

```
The original signal is:

        0.000000        1.000000        2.000000
        1.000000        2.000000        3.000000
        2.000000        3.000000        4.000000
        3.000000        4.000000        5.000000
        4.000000        5.000000        6.000000
        5.000000        6.000000        7.000000
        6.000000        7.000000        8.000000
        7.000000        8.000000        9.000000

The Fourier transform of original signal is:

        (108.000000,0.000000)   (-52.827446,1.358153)   (-0.613126,-23.640092)
        (13.310056,1.647944)    (-4.000000,9.656854)    (-8.902567,-4.353284)
        (3.082392,-7.681941)    (5.303682,2.676653)     (-4.000000,4.000000)
        (-5.314558,-4.659704)   (0.917608,-8.368233)    (0.220781,11.701395)
        (-4.000000,1.656854)    (-0.343999,-3.954231)   (4.613126,-0.326383)
        (0.554051,4.364941)     (-4.000000,-0.000000)   (0.554051,-4.364941)
        (4.613126,0.326383)     (-0.343999,3.954231)    (-4.000000,-1.656854)
        (0.220781,-11.701395)   (0.917608,8.368233)     (-5.314558,4.659704)

The inverse Fourier thansform is

        0.000000        1.000000        2.000000
        1.000000        2.000000        3.000000
        2.000000        3.000000        4.000000
        3.000000        4.000000        5.000000
        4.000000        5.000000        6.000000
        5.000000        6.000000        7.000000
        6.000000        7.000000        8.000000
        7.000000        8.000000        9.000000

The original signal is:

        (0.000000,0.000000)     (0.000000,1.000000)     (0.000000,2.000000)
        (1.000000,0.000000)     (1.000000,1.000000)     (1.000000,2.000000)
        (2.000000,0.000000)     (2.000000,1.000000)     (2.000000,2.000000)
        (3.000000,0.000000)     (3.000000,1.000000)     (3.000000,2.000000)
        (4.000000,0.000000)     (4.000000,1.000000)     (4.000000,2.000000)
        (5.000000,0.000000)     (5.000000,1.000000)     (5.000000,2.000000)
        (6.000000,0.000000)     (6.000000,1.000000)     (6.000000,2.000000)
        (7.000000,0.000000)     (7.000000,1.000000)     (7.000000,2.000000)
```

```
44.
45.  The Fourier transform of original signal is:
46.
47.          (84.000000,24.000000)   (-42.542528,1.020494)   (5.034128,-18.695144)
48.          (13.685581,5.361763)    (-4.000000,9.656854)    (-6.244002,-4.826961)
49.          (6.192124,-5.705118)    (6.023013,5.306175)     (-4.000000,4.000000)
50.          (-1.277237,-6.299534)   (13.722858,-4.086928)   (-8.776842,-2.726593)
51.          (-4.000000,1.656854)    (-1.716734,-2.445797)   (2.729646,0.149298)
52.          (-0.485914,4.057658)    (-4.000000,-0.000000)   (0.861334,-3.324976)
53.          (4.137445,2.209863)     (-1.852433,5.326967)    (-4.000000,-1.656854)
54.          (14.648770,-2.703773)   (-3.363697,-4.437017)   (-3.674728,0.622383)
55.
56.  The inverse Fourier thansform is
57.
58.          (0.000000,0.000000)     (0.000000,1.000000)     (0.000000,2.000000)
59.          (1.000000,-0.000000)    (1.000000,1.000000)     (1.000000,2.000000)
60.          (2.000000,0.000000)     (2.000000,1.000000)     (2.000000,2.000000)
61.          (3.000000,-0.000000)    (3.000000,1.000000)     (3.000000,2.000000)
62.          (4.000000,-0.000000)    (4.000000,1.000000)     (4.000000,2.000000)
63.          (5.000000,-0.000000)    (5.000000,1.000000)     (5.000000,2.000000)
64.          (6.000000,-0.000000)    (6.000000,1.000000)     (6.000000,2.000000)
65.          (7.000000,0.000000)     (7.000000,1.000000)     (7.000000,2.000000)
```

4.2　2 的整次幂 FFT 算法

SP++中实现了长度为 2 的整次幂的FFT算法，FFTR2<Type>类采用了基 8、基 4 和基 2 的混合基算法，并采用经济存储模式，具体函数见表 4.2。长度为 2 的整次幂信号的FFT计算效率非常高，故实际中很多应用都采取了该方式。如果某些用户无法使用FFTW，可用FFRT2<Type>类计算离散Fourier变换，对于本库中其它使用到FFTW的地方，只需对程序稍加修改即可。

表 4.2　长度为 2 的幂次的 FFT 算法

Operation	Effect
FFTR2<Type> ft	创建 FFTR2 类
ft.~FFTR2<Type>()	析构 FFTR2 类
ft.fft(cxn)	复信号的 Fourier 变换
ft.fft(cxn,Xk)	实信号的 Fourier 变换
ft.ifft(cXk)	复信号的逆 Fourier 变换
ft.ifft(Xk,rxn)	实信号的逆 Fourier 变换

测试代码：

```
/*****************************************************************************
 *                              fftr2_test.cpp
 *
 * Radix 2 based FFT testing.
 *
 * Zhang Ming, 2010-04
 *****************************************************************************/


#include <iostream>
#include <iomanip>
#include <fftr2.h>


using namespace std;
using namespace itlab;


int LENGTH = 32;


int main()
{
    int     i, j, index, rows = LENGTH/4;

    Vector<double> xn(LENGTH);
    Vector< complex<double> > yn(LENGTH),
                              Xk(LENGTH);
    FFTR2<double> Fourier;

    cout << "The original signal is: " << endl;
    for( i=0; i<rows; i++ )
    {
        cout << endl;
        for( j=0; j<3; j++ )
        {
            index = 3*i+j;
            xn[index] = i+j;
            cout << setiosflags(ios::fixed) << setprecision(6);
            cout << "\t" << xn[index];
        }
    }
    cout << endl << endl;

    Fourier.fft( xn, Xk );
```

```
46.
47.     cout << "The Fourier transform of original signal is:" << endl;
48.     for( i=0; i<rows; i++ )
49.     {
50.         cout << endl;
51.         for( j=0; j<3; j++ )
52.         {
53.             index = 3*i+j;
54.             cout << setiosflags(ios::fixed) << setprecision(6);
55.             cout << "\t" << Xk[index];
56.         }
57.     }
58.     cout << endl << endl;
59.
60.     Fourier.ifft( Xk, xn );
61.     cout << "The inverse Fourier thansform is" << endl;
62.     for( i=0; i<rows; i++ )
63.     {
64.         cout << endl;
65.         for( j=0; j<3; j++ )
66.         {
67.             index = 3*i+j;
68.             cout << setiosflags(ios::fixed) << setprecision(6);
69.             cout << "\t" << xn[index];
70.         }
71.     }
72.     cout << endl << endl;
73.
74.     cout << "The original signal is: " << endl;
75.     for( i=0; i<rows; i++ )
76.     {
77.         cout << endl;
78.         for( j=0; j<3; j++ )
79.         {
80.             index = 3*i+j;
81.             yn[index] = complex<double>(i,j);
82.             cout << setiosflags(ios::fixed) << setprecision(6);
83.             cout << "\t" << yn[index];
84.         }
85.     }
86.     cout << endl << endl;
87.
88.     Fourier.fft( yn );
89.     cout << "The Fourier transform of original signal is:" << endl;
90.     for( i=0; i<rows; i++ )
```

```
91.      {
92.          cout << endl;
93.          for( j=0; j<3; j++ )
94.          {
95.              index = 3*i+j;
96.              cout << setiosflags(ios::fixed) << setprecision(6);
97.              cout << "\t" << yn[index];
98.          }
99.      }
100.     cout << endl << endl;
101.
102.     Fourier.ifft( yn );
103.     cout << "The inverse Fourier thansform is" << endl;
104.     for( i=0; i<rows; i++ )
105.     {
106.         cout << endl;
107.         for( j=0; j<3; j++ )
108.         {
109.             index = 3*i+j;
110.             cout << setiosflags(ios::fixed) << setprecision(6);
111.             cout << "\t" << yn[index];
112.         }
113.     }
114.
115.     cout << endl << endl;
116.     return 0;
117. }
```

运行结果：

```
1.   xn:   size: 7 by 1
2.   (1,0)
3.   (0.62349,0.781832)
4.   (-0.222521,0.974928)
5.   (-0.900969,0.433884)
6.   (-0.900969,-0.433884)
7.   (-0.222521,-0.974928)
8.   (0.62349,-0.781832)
9.
10.
11.  Xk=fft(xn):   size: 7 by 1
12.  (-1.78814e-007,-5.96046e-008)
13.  (7,-2.7934e-007)
14.  (2.13795e-007,-2.66695e-008)
15.  (1.84928e-007,8.90184e-008)
```

```
(-3.25876e-008,9.92628e-008)
(4.93861e-008,1.54048e-007)
(-1.20608e-007,2.32841e-008)


yn=ifft(Xk):   size: 7 by 1
(1,1.77636e-015)
(0.62349,0.781831)
(-0.222521,0.974928)
(-0.900969,0.433884)
(-0.900969,-0.433884)
(-0.222521,-0.974928)
(0.62349,-0.781832)


sn:   size: 7 by 1
0
0.781832
0.974928
0.433884
-0.433884
-0.974928
-0.781832

Sk=fft(sn):   size: 4 by 1
(-5.96046e-008,0)
(-1.28028e-007,-3.5)
(6.36894e-008,-8.22043e-008)
(9.41406e-008,-1.08758e-007)


sn=ifft(Sk):   size: 7 by 1
-2.03012e-015
0.781832
0.974928
0.433884
-0.433884
-0.974928
-0.781832
```

4.3 卷积与相关快速算法

卷积与相关是信号处理中经常使用的运算，根据卷积定理和卷积与相关的关系，可以将时域定义的卷积与相关运算转化到频域进行计算，利用FFT算法可以将时间复杂度从N^2降低为$Nlog_2N$，极大提高了计算效率。SP++提供了基于FFT的卷积、自（互）相关的快速算法，见表 4.3。

表 4.3 卷积与相关的快速算法

Operation	Effect
fastConv(xn,yn)	计算 xn 与 yn 的卷积
fastAcor(xn)	计算 xn 的自卷积
fastCcor(xn,yn)	计算 xn 与 yn 的互相关

测试代码：

```
/*****************************************************************************
 *                                fastconv_test.cpp
 *
 * Fast convolution testing.
 *
 * Zhang Ming, 2010-01
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <fastconv.h>


using namespace std;
using namespace itlab;


const   int M = 3;
const   int N = 5;
typedef double  Type;


int main()
{
    Vector<Type> xn( M ),
                 yn( N );
    Vector<Type> zn;
```

```
30.
31.     for( int i=0; i<M; ++i )
32.         xn[i] = i;
33.     for( int i=0; i<N; ++i )
34.         yn[i] = i-N/2;
35.
36.     // covolution
37.     zn = fastConv( xn, yn );
38.     cout << "xn:  " << xn << endl << endl
39.          << "yn:  " << yn << endl << endl;
40.     cout << "convolution:   " << zn << endl << endl;
41.
42.     // autor and cross correlation functions
43.     zn = fastAcor( xn );
44.     cout << "auto-correlation:   " << zn << endl << endl;
45.     zn = fastCcor( xn, yn );
46.     cout << "cross-correlation:   " << zn << endl;
47.
48.     return 0;
49. }
```

运行结果：

```
1.  xn:  size: 3 by 1
2.  0
3.  1
4.  2
5.
6.  yn:  size: 5 by 1
7.  -2
8.  -1
9.  0
10. 1
11. 2
12.
13. convolution:   size: 7 by 1
14. 8.72318e-017
15. -2
16. -5
17. -2
18. 1
19. 4
20. 4
21.
22. auto-correlation:   size: 5 by 1
```

```
-4.44089e-017
2
5
2
-5.67255e-017

cross-correlation:   size: 7 by 1
-4
-4
-1
2
5
2
7.56959e-016
```

5 数字滤波器设计

5.1 常用窗函数

表 5.1列出了常用窗函数的调用形式，返回类型均为double型，其结果与Matlab中窗函数的结果相同。

表 5.1 常用的窗函数

Operation	Effect
rectangle(n)	矩形窗
bartlett(n)	Bartlett 窗
hanning(n)	Hanning 窗
hamming(n)	Hamming 窗
blackman(n)	Blackman 窗
kaiser(n,alpha)	Kaiser 窗
gauss(n,alpha)	Gauss 窗

测试代码：

```
1.  /*****************************************************************************
2.   *                               window_test.cpp
3.   *
4.   * Windows function testing.
5.   *
6.   * Zhang Ming, 2010-01
7.   *****************************************************************************/
8.
9.
10.  #define BOUNDS_CHECK
11.
12.  #include <iostream>
13.  #include <window.h>
14.
15.
16.  using namespace std;
17.  using namespace itlab;
18.
19.
```

```
int main()
{
    int N = 5;

    cout << "Rectangle window : " << rectangle(N) << endl << endl;
    cout << "Bartlett window : " << bartlett(N) << endl << endl;
    cout << "Hanning window : " << hanning(N) << endl << endl;
    cout << "Hamming window : " << hamming(N) << endl << endl;
    cout << "Blackman window : " << blackman(N) << endl << endl;
    cout << "Kaiser window : " << kaiser(8/PI,N) << endl << endl;
    cout << "Gauss window : " << gauss(N) << endl;

    return 0;
}
```

运行结果：

```
Rectangle window : size: 5 by 1
1
1
1
1
1

Bartlett window : size: 5 by 1
0
0.5
1
0.5
0

Hanning window : size: 5 by 1
0
0.5
1
0.5
0

Hamming window : size: 5 by 1
0.08
0.54
1
0.54
0.08

```

```
Blackman window : size: 5 by 1
-1.38778e-017
0.34
1
0.34
-1.38778e-017

Kaiser window : size: 2 by 1
0.0367109
0.0367109

Gauss window : size: 5 by 1
0.0439369
0.457833
1
0.457833
0.0439369
```

5.2 滤波器基类设计

所有滤波器都有其共性，如频率选择特性，通带频率，截止频率，增益等待，因此可以将这些共性抽取出来作为滤波器设计的一个基类，如表 5.2所示。

表 5.2 数字滤波器设计基类

Operation	Effect
DFD f(select)	创建 DFD 类
f.~ DFD <Type>()	析构 DFD 类
f. setParams(fs,f1,a1,f2,a2)	根据给定参数设计滤波器
f. setParams(fs,f1,a1,f2,f3,a2,f4,a3)	显示滤波器设计结果

5.3 FIR 数字滤波器设计

有限脉冲响应滤波器（FIR）具有线性相位特性，因此在一些对相位比较敏感的场合中得到了广泛应用。并且 FIR 没有反馈，所以稳定性比较好，当然其选择性也比较差。本库中实现了基于窗函数法的 FIR 设计方法，具体见**错误！未找到引用源。**。

表 5.3 FIR 数字滤波器设计

Operation	Effect
FIR<Type> f(select,win)	创建 FIR 类
FIR<Type> f(select,win,a)	创建 FIR 类(针对 Kaiser 与 Gauss 窗)

f.~FIR<Type>()	创建 FIR 类
f.design()	根据给定参数设计滤波器
f.dispInfo()	显示滤波器设计结果
f.getCoefs()	获取滤波器系数

测试代码：

```
/*****************************************************************************
 *                              fir_test.cpp
 *
 * FIR class testing.
 *
 * Zhang Ming, 2010-03
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <fir.h>


using namespace std;
using namespace itlab;


int main()
{
    string  wType = "Hamming";

//    string  fType = "lowpass";
//    double  fs = 1000,
//            fpass = 200,
//            apass = -3,
//            fstop = 300,
//            astop = -20;
//    FIR fir( fType, wType );
//    fir.setParams( fs, fpass, apass, fstop, astop );

//    string  fType = "highpass";
//    double  fs = 1000,
//            fstop = 200,
//            astop = -20,
//            fpass = 300,
//            apass = -3;
```

```
39.  //    FIR fir( fType, wType );
40.  //    fir.setParams( fs, fstop, astop, fpass, apass );
41.
42.  //    string  fType = "bandpass";
43.  //    double  fs = 1000,
44.  //            fstop1 = 100,
45.  //            astop1 = -20,
46.  //            fpass1 = 200,
47.  //            fpass2 = 300,
48.  //            apass1 = -3,
49.  //            fstop2 = 400,
50.  //            astop2 = -20;
51.  //    FIR fir( fType, wType );
52.  //    fir.setParams( fs, fstop1, astop1, fpass1, fpass2, apass1, fstop2, astop2 );
53.
54.      string  fType = "bandstop";
55.      double  fs = 1000,
56.              fpass1 = 100,
57.              apass1 = -3,
58.              fstop1 = 200,
59.              fstop2 = 300,
60.              astop1 = -20,
61.              fpass2 = 400,
62.              apass2 = -3;
63.      FIR fir( fType, wType );
64.      fir.setParams( fs, fpass1, apass1, fstop1, fstop2, astop1, fpass2, apass2 );
65.
66.      fir.design();
67.      fir.dispInfo();
68.
69.      cout << endl;
70.      return 0;
71.  }
```

运行结果：

```
27.                     Filter selectivity      :  bandstop
28.                     Window type             :  Hamming
29.                     Sampling Frequency (Hz) :  1000
30.                     Lower passband frequency (Hz) :  100
31.                     Lower passband gain      (dB) :  -3
32.                     Lower stopband frequency (Hz) :  200
33.                     Upper stopband frequency (Hz) :  300
34.                     Stopband gain            (dB) :  -20
35.                     Upper passband frequency (Hz) :  400
```

```
                    Upper passband gain      (dB) :  -3


                                 Filter Coefficients

     N    [     N + 0            N + 1            N + 2            N + 3     ]
    ===   ================================================================
     0    -3.85192764e-003  8.42472981e-003  3.11153775e-003 -2.03549729e-003
     4    -3.55185234e-002  2.30171390e-002 -1.41331145e-001  2.61755439e-001
     8     2.88881794e-002  3.56158158e-001  3.56158158e-001  2.88881794e-002
    12     2.61755439e-001 -1.41331145e-001  2.30171390e-002 -3.55185234e-002
    16    -2.03549729e-003  3.11153775e-003  8.42472981e-003 -3.85192764e-003


          ==================== Edge Frequency Response ====================
              Mag(fp1) = -0.85449456(dB)       Mag(fp2) = -0.80635855(dB)
              Mag(fs1) = -21.347821(dB)       Mag(fs2) = -21.392825(dB)

```

5.4 IIR 数字滤波器设计

无限脉冲响应（IIR）滤波器采用了反馈，因此具有很高的频率选择性，但同时也损失了线性相位特性，并且稳定性也不如FIR好。IIR可以借助模拟滤波器进行设计，而模拟滤波器有很成熟的设计方法，这给IIR设计带来了很大好处。本库实现的基于双线性变换的IIR设计方法，详见表 5.4。

表 5.4 IIR 数字滤波器设计

Operation	Effect
IIR<Type> f(select,method)	创建 IIR 类
f.~IIR<Type>()	析构 IIR 类
f.design()	根据给定参数设计滤波器
f.dispInfo()	显示滤波器设计结果
f.getNumCoefs()	获取滤波器的分子系数
f.getDenCoefs()	获取滤波器的分母系数

测试代码：

```
/*************************************************************************
 *                              iir_test.cpp
 *
 * IIR class testing.
 *
```

```
6.    * Zhang Ming, 2010-03
7.    ***************************************************************************/
8.
9.
10.  #define BOUNDS_CHECK
11.
12.  #include <iostream>
13.  #include <iir.h>
14.
15.
16.  using namespace std;
17.  using namespace itlab;
18.
19.
20.  int main()
21.  {
22.  //    string  fType = "lowpass";
23.  //    double  fs = 1000,
24.  //            fpass = 200,
25.  //            apass = -3,
26.  //            fstop = 300,
27.  //            astop = -20;
28.  //    string  method = "Butterworth";
29.  //    string  method = "Chebyshev";
30.  //    string  method = "InvChebyshev";
31.  //    string  method = "Elliptic";
32.  //    IIR iir( fType, method );
33.  //    iir.setParams( fs, fpass, apass, fstop, astop );
34.
35.  //    string  fType = "highpass";
36.  //    double  fs = 1000,
37.  //            fstop = 200,
38.  //            astop = -20,
39.  //            fpass = 300,
40.  //            apass = -3;
41.  //    string  method = "Butterworth";
42.  //    string  method = "Chebyshev";
43.  //    string  method = "InvChebyshev";
44.  //    string  method = "Elliptic";
45.  //    IIR iir( fType, method );
46.  //    iir.setParams( fs, fstop, astop, fpass, apass );
47.
48.  //    string  fType = "bandpass";
49.  //    double  fs = 1000,
50.  //            fstop1 = 100,
```

```
51.  //            astop1 = -20,
52.  //            fpass1 = 200,
53.  //            fpass2 = 300,
54.  //            apass1 = -3,
55.  //            fstop2 = 400,
56.  //            astop2 = -20;
57.  //    string  method = "Butterworth";
58.  //    string  method = "Chebyshev";
59.  //    string  method = "InvChebyshev";
60.  //    string  method = "Elliptic";
61.  //    IIR iir( fType, method );
62.  //    iir.setParams( fs, fstop1, astop1, fpass1, fpass2, apass1, fstop2, astop2 );
63.
64.      string  fType = "bandstop";
65.      double  fs = 1000,
66.              fpass1 = 100,
67.              apass1 = -3,
68.              fstop1 = 200,
69.              fstop2 = 300,
70.              astop1 = -20,
71.              fpass2 = 400,
72.              apass2 = -3;
73.  //    string  method = "Butterworth";
74.  //    string  method = "Chebyshev";
75.  //    string  method = "InvChebyshev";
76.      string  method = "Elliptic";
77.      IIR iir( fType, method );
78.      iir.setParams( fs, fpass1, apass1, fstop1, fstop2, astop1, fpass2, apass2 );
79.
80.      iir.design();
81.      iir.dispInfo();
82.
83.      cout << endl;
84.      return 0;
85.  }
```

运行结果：

```
1.                      Filter selectivity      :  bandstop
2.                      Approximation method    :  Elliptic
3.                      Filter order            :  4
4.                      Overall gain            :  0.153272
5.                      Sampling Frequency (Hz) :  1000
6.                      Lower passband frequency (Hz) :  100
7.                      Lower passband gain     (dB) :  -3
```

```
                    Lower stopband frequency (Hz) :  200
                    Upper stopband frequency (Hz) :  300
                    Stopband gain            (dB) :  -20
                    Upper passband frequency (Hz) :  400
                    Upper passband gain      (dB) :  -3


            Numerator Coefficients               Denominator Coefficients

    N   [  1       z^-1           z^-2  ]     [  1       z^-1           z^-2  ]
   ===  ================================     ================================
    1    1.0   -0.45087426    1.00000000       1.0   -1.44482415    0.70572930
    2    1.0    0.45087426    1.00000000       1.0    1.44482415    0.70572930


          ==================== Edge Frequency Response ====================
              Mag(fp1) = -3.00000000(dB)       Mag(fp2) = -3.00000000(dB)
              Mag(fs1) = -36.87418446(dB)      Mag(fs2) = -36.87418446(dB)
```

6 时频分析

6.1 加窗 Fourier 变换

加窗 Fourier 变换（WFT）或短时 Fourier 变换（STFT）由于其思想直观，实现简单，成为了最常用的时频分析方法之一。WFT 通过对信号与时频原子作内积，将信号从时域信号变换到时频域，实现的时频局部化的分析特性，克服了 Fourier 变换全局性的缺点。

时频原子是通过对基本窗函数进行时移和调制所得，所以窗函数的选择对时频变换结果影响非常大，通常选用时频聚集性好的窗函数，如Gauss窗或Hamming等等。WFT的正反变换调用格式见表 6.1。

表 6.1 加窗 Fourier 变换

Operation	Effect
wft(sn,gn,mod)	计算 sn 的加窗 Fourier 变换系数
iwft(coefs,gn)	计算 coefs 的逆加窗 Fourier 变换

测试代码：

```
/*****************************************************************************
 *                                wft_test.cpp
 *
 * Windowed Fourier transform testing.
 *
 * Zhang Ming, 2010-03
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <wft.h>
#include <mathfunc.h>
#include <timing.h>


using namespace std;
using namespace itlab;
```

```
20.
21.
22.  const   int Lg = 128;
23.  const   int Ls = 1000;
24.  const   int Fs = 1000;
25.  typedef     float   Type;
26.
27.
28.  int main()
29.  {
30.      /****************************** [ signal ] ******************************/
31.      Type a = 0;
32.      Type b = Ls-1;
33.      Vector<Type> t = linspace(a,b,Ls) / Type(Fs);
34.      Vector<Type> s = vsin( Type(400*PI) * vpow(t,Type(2.0)) );
35.
36.      /****************************** [ widow ] ******************************/
37.      a = 0;
38.      b = Type(Lg-1);
39.      Type u = (Lg-1)/Type(2);
40.      Type r = Lg/Type(8);
41.      t = linspace(a,b,Lg);
42.      Vector<Type> g = gauss(t,u,r);
43.      g = g/norm(g);
44.
45.      /******************************** [ WFT ] ********************************/
46.      Type runtime = 0;
47.      Timing cnt;
48.      cout << "Taking windowed Fourier transform." << endl;
49.      cnt.start();
50.      Matrix< complex<Type> > coefs = wft( s, g );
51.      cnt.stop();
52.      runtime = cnt.read();
53.      cout << "The running time = " << runtime << " (ms)" << endl << endl;
54.
55.      /******************************* [ IWFT ] *******************************/
56.      cout << "Taking inverse windowed Fourier transform." << endl;
57.      cnt.start();
58.      Vector<Type> x = iwft( coefs, g );
59.      cnt.stop();
60.      runtime = cnt.read();
61.      cout << "The running time = " << runtime << " (ms)" << endl << endl;
62.
63.      cout << "The relative error is : " << "norm(s-x) / norm(s) = "
64.           << norm(s-x)/norm(s) << endl << endl;
```

```
65.
66.      return 0;
67.  }
```

运行结果：

```
1.   Taking windowed Fourier transform.
2.   The running time = 0.011 (ms)
3.
4.   Taking inverse windowed Fourier transform.
5.   The running time = 0.009 (ms)
6.
7.   The relative error is : norm(s-x) / norm(s) = 4.24083e-008
```

6.2 离散 Gabor 变换

离散 Gabor 变换（DGT）是用 Gabor 分析标架与综合标架对信号进行时频变换与逆变换，可以看作是 WFT 在时频域的一种采样。其冗余度可能通过过采样率进行调整，并且同样可以借助 FFT 算法实现，因此计算效率相对 WFT 得到很大的提高。

DGT是一种标架运算，故其分析窗函数与综合窗函数一般是不相同的，而是要通过对偶窗函数进行计算。”dgt.h”头文件中提供了对偶函数以及离散Gabor正反变换的计算，具体见表 6.2。

表 6.2 离散 Gabor 变换

Operation	Effect
dual(gn,N,dM)	计算 gn 的对偶窗函数
dgt(sn,gn,N,dM,mode)	计算 sn 的离散 Gabor 变换
idgt(coefs,gn,N,dM)	计算 coefs 的离散 Gabor 逆变换

测试代码：

```
1.   /**************************************************************************
2.    *                              dgt_test.cpp
3.    *
4.    * Discrete Gabor transform testing.
5.    *
6.    * Zhang Ming, 2010-03
7.    **************************************************************************/
8.
9.
10.  #define BOUNDS_CHECK
11.
```

```
#include <iostream>
#include <dgt.h>
#include <mathfunc.h>
#include <timing.h>


using namespace std;
using namespace itlab;


const   int Fs = 1000;
const   int Ls = 1000;
const   int Lg = 80;
const   int N  = 40;
const   int dM = 10;      // over sampling ratio is N/dM
typedef     double   Type;


int main()
{

    /****************************** [ signal ] ******************************/
    Type a = 0;
    Type b = Ls-1;
    Vector<Type> t = linspace( a, b, Ls ) / Type(Fs);
    Vector<Type> st = vcos( Type(400*PI) * vpow(t,Type(2.0)) );

    /****************************** [ widow ] ******************************/
    a = 0;
    b = Type( Lg-1 );
    Type r = sqrt( dM*N / Type(2*PI) );
    Type u = (Lg-1) / Type(2.0);
    t = linspace( a, b, Lg );
    Vector<Type> h = gauss( t, u, r );
    h = h / norm(h);

    /****************************** [ daul function ] ******************************/
    Type runtime = 0.0;
    Timing cnt;
    cout << "Compute daul function." << endl;
    cnt.start();
    Vector<Type> g = daul( h, N, dM );
    cnt.stop();
    runtime = cnt.read();
    cout << "The running time = " << runtime << " (ms)" << endl << endl;
```

```
57.
58.     /****************************** [ DGT ] ******************************/
59.     cout << "Taking discrete Gabor transform." << endl;
60.     cnt.start();
61.     Matrix< complex<Type> > C = dgt( st, g, N, dM, "sym" );
62.     cnt.stop();
63.     runtime = cnt.read();
64.     cout << "The running time = " << runtime << " (ms)" << endl << endl;
65.
66.     /****************************** [ IDGT ] ******************************/
67.     cout << "Taking inverse discrete Gabor transform." << endl;
68.     cnt.start();
69.     Vector<Type> xt = idgt( C, h, N, dM );
70.     cnt.stop();
71.     runtime = cnt.read();
72.     cout << "The running time = " << runtime << " (ms)" << endl << endl;
73.
74.     cout <<"The relative erroris : norm(s-x) / norm(s) = "
75.          << norm(st-xt)/norm(st) << endl << endl;
76.
77.     return 0;
78. }
```

运行结果：

```
1.  Compute daul function.
2.  The running time = 0.001 (ms)
3.
4.  Taking discrete Gabor transform.
5.  The running time = 0.007 (ms)
6.
7.  Taking inverse discrete Gabor transform.
8.  The running time = 0.004 (ms)
9.
10. The relative erroris : norm(s-x) / norm(s) = 3.80507e-013
11.
```

7 小波变换

7.1 连续小波变换

小波分析自上世纪 80 年代以来得到了飞速发展，并且在地震信号处理，生物医学信号处理，图像压缩与编码以及机械故障诊断等领域中取得了非常成功的应用。现已成为了最常用的一种信号分析手段。

连续小波变换（CWT）具有很高的冗余度，但其所含信息丰富，经常用于信号的属性提取；同时CWT具有很好的稳定性，在去噪中也得到了广泛使用。SP++中提供了实连续小波变换（以Mexico帽小波为例）和复连续小波变换（以Morlet小波为例）的正反变换程序，调用格式见表 7.1。其中逆变换的重构精度可以通过尺度参数进行调整，精度越高，计算量就越大，默认参数的重构误差在 10^{-4} 至 10^{-8} 之间。

表 7.1 连续小波变换

Operation	Effect
CWT<Type> wt(wname)	创建 CWT 类
wt.~CWT<Type>()	析构 CWT 类
setScales(fs,fmin,fmax,dj)	设置尺度参数
cwrR(sn)	计算 sn 的实小波变换
icwrR(coefs)	计算 coefs 的实小波逆变换
cwrC(sn)	计算 sn 的复小波变换
icwrC(coefs)	计算 coefs 的复小波逆变换

测试代码：

```
1.  /**************************************************************************
2.   *                               cwt_test.cpp
3.   *
4.   * Continuous wavelet transform testing.
5.   *
6.   * Zhang Ming, 2010-03
7.   **************************************************************************/
8.
9.
10.  #define BOUNDS_CHECK
11.
12.  #include <iostream>
13.  #include <cwt.h>
```

```
#include <mathfunc.h>
#include <utilities.h>
#include <statistics.h>
#include <timing.h>


using namespace std;
using namespace itlab;


const int Ls = 1000;
const double fs = 1000.0;


int main()
{

    /****************************** [ signal ] ******************************/
    Vector<double> t = linspace( 0.0, (Ls-1)/fs, Ls );
    Vector<double> st = vsin( 200*PI*vpow(t,2.0) );
    st = st-mean(st);

    /****************************** [ CWT ] ******************************/
    Matrix< complex<double> > coefs;
    CWT<double> wavelet("morlet");
    wavelet.setScales( fs, fs/Ls, fs/2 );
    Timing cnt;
    double runtime = 0.0;
    cout << "Taking contineous wavelet transform(Morlet)." << endl;
    cnt.start();
    coefs = wavelet.cwtC(st);
    cnt.stop();
    runtime = cnt.read();
    cout << "The running time = " << runtime << " (ms)" << endl << endl;

    /****************************** [ ICWT ] ******************************/
    cout << "Taking inverse contineous wavelet transform." << endl;
    cnt.start();
    Vector<double> xt = wavelet.icwtC(coefs);
    cnt.stop();
    runtime = cnt.read();
    cout << "The running time = " << runtime << " (ms)" << endl << endl;

    cout << "The relative error is : " << endl;
    cout << "norm(st-xt) / norm(st) = " << norm(st-xt)/norm(st) << endl;
```

```
59.       cout << endl << endl;
60.
61.
62.       /****************************** [ signal ] ******************************/
63.       Vector<float> tf = linspace( float(0.0), (Ls-1)/float(fs), Ls );
64.       Vector<float> stf = vsin( float(200*PI) * vpow(tf,float(2.0) ) );
65.       stf = stf-mean(stf);
66.
67.       /****************************** [ CWT ] ******************************/
68.       CWT<float> waveletf("mexiHat");
69.       waveletf.setScales( fs, fs/Ls, fs/2, float(0.25) );
70.       runtime = 0.0;
71.       cout << "Taking contineous wavelet transform(Mexican Hat)." << endl;
72.       cnt.start();
73.       Matrix<float> coefsf = waveletf.cwtR(stf);
74.       cnt.stop();
75.       runtime = cnt.read();
76.       cout << "The running time = " << runtime << " (ms)" << endl << endl;
77.
78.       /****************************** [ ICWT ] ******************************/
79.       cout << "Taking inverse contineous wavelet transform." << endl;
80.       cnt.start();
81.       Vector<float> xtf = waveletf.icwtR(coefsf);
82.       cnt.stop();
83.       runtime = cnt.read();
84.       cout << "The running time = " << runtime << " (ms)" << endl << endl;
85.
86.       cout << "The relative error is : " << endl;
87.       cout << "norm(st-xt) / norm(st) = " << norm(stf-xtf)/norm(stf) << endl << endl;
88.
89.       return 0;
90.   }
```

运行结果：

```
1.    Taking contineous wavelet transform(Morlet).
2.    The running time = 0.02 (ms)
3.
4.    Taking inverse contineous wavelet transform.
5.    The running time = 0.001 (ms)
6.
7.    The relative error is :
8.    norm(st-xt) / norm(st) = 0.000604041
9.
10.
```

```
Taking contineous wavelet transform(Mexican Hat).
The running time = 0.024 (ms)

Taking inverse contineous wavelet transform.
The running time = 0.001 (ms)

The relative error is :
norm(st-xt) / norm(st) = 0.00156796
```

7.2 二进小波变换

连续小波变换的尺度与平移参数都是连续取值的，所以冗余度很高，如果对尺度参数进行离散化而保持平移参数连续，就得到二进小波变换。二进小波变换即减少了冗余，又能够保证小波变换的平移不变性，因此得到了广泛的应用。

SP++中提供了基于二次样条的二进小波变换程序，如表 7.2所示。对信号进行J级的二进小波变换后可得到 1 级到J级的细节系数和J级的平滑系数，如果想得到j级的平滑系数，则可对小波变换系数进行J-j级的重构即可。

表 7.2 二进小波变换

Operation	Effect
bwt(sn,J)	对 sn 进行 J 级的二进小波变换
ibwt(coefs,j)	对 coefs 进行 j 级的二进小波重构

测试代码：

```
/*****************************************************************************
 *                              bwt_test.cpp
 *
 * Dyadic wavelet transform testing.
 *
 * Zhang Ming, 2010-03
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <bwt.h>
#include <timing.h>


using namespace std;
```

```
18.  using namespace itlab;
19.
20.
21.  const int Ls = 100;
22.
23.
24.  int main()
25.  {
26.
27.      /************************* [ signal ] ************************/
28.      Vector<double> s(Ls);
29.      for(int i=0; i<Ls; i++)
30.      {
31.          if(i<Ls/4)
32.              s[i] = 0.0;
33.          else if(i<2*Ls/4)
34.              s[i] = 1.0;
35.          else if(i<3*Ls/4)
36.              s[i] = 3.0;
37.          else
38.              s[i] = 0.0;
39.      }
40.
41.      /************************** [ BWT ] **************************/
42.      int level = 3;
43.      Timing cnt;
44.      double runtime = 0.0;
45.      cout << "Taking dyadic wavelet transform." << endl;
46.      cnt.start();
47.      Vector< Vector<double> > coefs = bwt( s, level );
48.      cnt.stop();
49.      runtime = cnt.read();
50.      cout << "The running time = " << runtime << " (s)" << endl << endl;
51.
52.      /************************** [ IBWT ] **************************/
53.      cout << "Taking inverse dyadic wavelet transform." << endl;
54.      cnt.start();
55.      Vector<double> x = ibwt( coefs, level );
56.      cnt.stop();
57.      runtime = cnt.read();
58.      cout << "The running time = " << runtime << " (s)" << endl << endl;
59.
60.      cout << "The relative error is : norm(s-x) / norm(s) = "
61.           << norm(s-x)/norm(s) << endl << endl;
62.
```

```
63.      return 0;
64.  }
```

运行结果：

```
1.   Taking dyadic wavelet transform.
2.   The running time = 0 (s)
3.
4.   Taking inverse dyadic wavelet transform.
5.   The running time = 0 (s)
6.
7.   The relative error is : norm(s-x) / norm(s) = 3.76025e-016
```

7.3 离散小波变换

如果对连续小波变换的尺度参数与平移参数均进行二进离散化，即为离散小波变换。离散小波变换实际上是信号在正交小波基上的投影，故没有冗余，并且在 Mallat 塔式快速算法，计算效率非常高。所以离散小波变换在图像压缩与编码中得到了非常成功的应用。

SP++中提供了离散小波变换的一般算法，默认的滤波器组为”db4”小波，用户根据自己的需要进行修改。为了节约存储空间，小波变换系数存放于一个一维数组中，可以通过一个指标数组访问不同的细节系数和逼近系数，但是这些实现细节已封装在类的内部，用户不必关心。具体的常用操作见表 7.3。

表 7.3 离散小波变换

Operation	Effect
DWT<Type> wt(wname)	创建 DWT 类
wt.~DWT<Type>()	析构 DWT 类
wt.dwt(sn,J)	对 sn 进行 J 级的离散小波变换
wt.idwt(coefs,j)	对 coefs 进行 j 级的离散小波重构
wt.getApprox(ceofs)	获取离散小波变换的逼近系数
wt.getApprox(ceofs,j)	获取离散小波变换的 j 级细节系数
wt.setApprox(a,ceofs)	设置离散小波变换的逼近系数
wt.setApprox(d,ceofs,j)	设置离散小波变换的 j 级细节系数

测试代码：

```
1.   /*************************************************************************
2.    *                              dwt_test.cpp
3.    *
4.    * Discrete wavelet transform testing.
5.    *
```

```
6.    * Zhang Ming, 2010-03
7.    ***************************************************************************/
8.
9.
10.  #define BOUNDS_CHECK
11.
12.  #include <iostream>
13.  #include <dwt.h>
14.  #include <timing.h>
15.
16.
17.  using namespace std;
18.  using namespace itlab;
19.
20.
21.  const int Ls = 1000;
22.
23.
24.  int main()
25.  {
26.
27.      /****************************** [ signal ] ******************************/
28.      Vector<float> s(Ls);
29.      for(int i=0; i<Ls; i++)
30.      {
31.          if(i<Ls/4)
32.              s[i] = 0.0;
33.          else if(i<2*Ls/4)
34.              s[i] = 1.0;
35.          else if(i<3*Ls/4)
36.              s[i] = 3.0;
37.          else
38.              s[i] = 0.0;
39.      }
40.
41.      /****************************** [ DWT ] ******************************/
42.      int level = 3;
43.      DWT<float> discreteWT("db4");
44.      Timing cnt;
45.      float runtime = 0.0;
46.      cout << "Taking discrete wavelet transform." << endl;
47.      cnt.start();
48.      Vector<float> coefs = discreteWT.dwt( s, level );
49.      cnt.stop();
50.      runtime = cnt.read();
```

```
51.       cout << "The running time = " << runtime << " (s)" << endl << endl;
52.
53.       /****************************** [ IDWT ] ******************************/
54.       level = 0;
55.       cout << "Taking inverse discrete wavelet transform." << endl;
56.       cnt.start();
57.       Vector<float> x = discreteWT.idwt( coefs, level );
58.       cnt.stop();
59.       runtime = cnt.read();
60.       cout << "The running time = " << runtime << " (s)" << endl << endl;
61.
62.       cout << "The relative error is : norm(s-x) / norm(s) = "
63.            << norm(s-x)/norm(s) << endl << endl;
64.
65.       return 0;
66.   }
```

运行结果：

```
1.    Taking discrete wavelet transform.
2.    first
3.    The running time = 0.001 (s)
4.
5.    Taking inverse discrete wavelet transform.
6.    The running time = 0.001 (s)
7.
8.    The relative error is : norm(s-x) / norm(s) = 9.26354e-008
```

8 优化算法

8.1 一维线搜索

所有的基于下降方向的优化方法都需要通过一维搜索来确定步长，而下降步长直接影响到优化方法的收敛速率，因此一维搜索是优化算法中一个十分关键的步骤。SP++中提供了非精确一维搜索算法类LineSearch<DType,Ftype>，同时可以获得确定步长时目标函数的计算次数。具体函数见表 8.1。

表 8.1 最速下降法

Operation	Effect
LineSearch<DType,Ftype> ols	创建一维搜索类
ols.~ LineSearch <DType,Ftype>()	析构一维搜索类
ols. getStep (func, x0, dk, maxItr)	求取一维搜索步长
ols. getFuncNum ()	获取目标函数的计算次数
ols. isSuccess ()	判断一维搜索是否成功

8.2 最速下降法

最速下降法（即梯度法）是一种最古老和最简单的优化算法，其优点是稳定性比较高，缺点是收敛速率非常慢。因此对收敛速率要求不是很高的问题和高维稳定性比较差的问题，该方法仍然是一种非常实用的算法。

SP++中最速下降类SteepDesc<DType,Ftype>提供了求最优值、最小函数数、梯度模值和目标函数计算次数等函数，详见表 8.2。

表 8.2 最速下降法

Operation	Effect
SteepDesc<DType,Ftype> fmin	创建最速下降法类
fmin.~SteepDesc<DType,Ftype>()	析构最速下降法类
fmin. optimize(func, x0, tol, maxItr)	求指定参数的函数最小值
fmin.getOptValue()	获取自变量的最优值
fmin.getGradNorm()	获取迭代过程中梯度向量的模值
fmin.getFuncMin()	获取函数的最小值
fmin.getItrNum()	获取迭代次数

最速下降法的测试代码：

```
/*****************************************************************************
 *                              steepdesc_test.cpp
 *
 * Steepest descent method testing.
 *
 * Zhang Ming, 2010-03
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <function.h>
#include <steepdesc.h>


using namespace std;
using namespace itlab;


typedef     float  Type;


int main()
{
    Type a = 1.0,
         b = -1.0,
         c = -1.0;
    ObjFunc<Type> f( a, b, c );
    Vector<Type> x0(2);
    x0(1) = Type(0.0);
    x0(2) = Type(0.0);

    Type tolErr = 1.0e-3;
    SteepDesc< Type, ObjFunc<Type> > steep;
    steep.optimize( f, x0, tolErr );
    if( steep.isSuccess() )
    {
        Vector<Type> xmin = steep.getOptValue();
        int N = steep.getItrNum();
        cout << "The iterative number is:   " << N << endl << endl;
        cout << "The number of function calculation is:   " << steep.getFuncNum() << endl << endl;
        cout << "The optimal value of x is:   " << xmin << endl;
```

```
44.          cout << "The minimum value of f(x) is:   " << f(xmin) << endl << endl;
45.          cout << "The gradient's norm at x is:   " << steep.getGradNorm()[N] << endl << endl;
46.      }
47.      else
48.          cout << "The optimal solution  cann't be found!" << endl;
49.
50.      return 0;
51.  }
```

运行结果：

```
1.   The iterative number is:   14
2.
3.   The number of function calculation is:   43
4.
5.   The optimal value of x is:   size: 2 by 1
6.   -0.706978
7.   0
8.
9.   The minimum value of f(x) is:   -0.428882
10.
11.  The gradient's norm at x is:   0.000599994
```

8.3 共轭梯度法

共轭梯度法通过生成共轭方向来确定每次迭代的搜索方向，具有二次终止性和较高的收敛速率，并且所需的存储空间也很少，因此对于一些高维的优化问题非常适用。但当目标函数不是二次函数（实际应用中目标函数往往不是二次的）时，不能通过 n 步迭代求得最优值，所以需要再开始技术重新确定下降方向。并且当目标函数不能用二次函数很好的逼近时，其收敛速率相对最速下降法并不具有明显的优势。

SP++中共轭梯度类ConjGrad <DType,Ftype>所提供的函数与最速下降法类相似，详见表 8.3，其中默认的再开始次数为目标函数的维数n。

表 8.3 共轭梯度法

Operation	Effect
ConjGrad <DType,Ftype> fmin	创建共轭梯度法类
fmin.~ ConjGrad <DType,Ftype>()	析构共轭梯度法类
fmin. optimize(func, x0, tol, maxItr)	求指定参数的函数最小值
fmin.getOptValue()	获取自变量的最优值
fmin.getGradNorm()	获取迭代过程中梯度向量的模值
fmin.getFuncMin()	获取函数的最小值
fmin.getItrNum()	获取迭代次数

共轭梯度法的测试代码：

```
/*************************************************************************
 *                              conjgrad_test.cpp
 *
 * ConjGrad descent method testing.
 *
 * Zhang Ming, 2010-03
 *************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <function.h>
#include <conjgrad.h>


using namespace std;
using namespace itlab;


typedef     double  Type;


int main()
{
    Type a = 1.0,
         b = -1.0,
         c = -1.0;
    ObjFunc<Type> f( a, b, c );
    Vector<Type> x0(2);
    x0(1) = 0.5;
    x0(2) = 0.5;

    Type tolErr = 1.0e-6;
    ConjGrad< Type, ObjFunc<Type> > prp;
    prp.optimize( f, x0, x0.dim(), tolErr );
    if( prp.isSuccess() )
    {
        Vector<Type> xmin = prp.getOptValue();
        int N = prp.getItrNum();
        cout << "The iterative number is:   " << N << endl << endl;
        cout << "The number of function calculation is:   " << prp.getFuncNum() << endl << endl;
```

```
43.          cout << "The optimal value of x is:   " << xmin << endl;
44.          cout << "The minimum value of f(x) is:   " << f(xmin) << endl << endl;
45.          cout << "The gradient's norm at x is:   " << prp.getGradNorm()[N] << endl << endl;
46.      }
47.      else
48.          cout << "The optimal solution  cann't be found!" << endl;
49.
50.      return 0;
51.  }
```

运行结果：

```
1.   The iterative number is:   29
2.
3.   The number of function calculation is:   207
4.
5.   The optimal value of x is:   size: 2 by 1
6.   -0.707107
7.   -2.25938e-019
8.
9.   The minimum value of f(x) is:   -0.428882
10.
11.  The gradient's norm at x is:   6.52428e-007
```

8.4 拟 Newton 法

拟 Newton 法通过梯度向量和校正公式来近似目标函数的 Hess 矩阵，同样具有二次终止性，并且相对最速下降法和共轭梯度法有很高的收敛速率。比较常用的两类拟 Newton 算法是 DFP 和 FBGS，是目前公认的求解非线性无约束优化问题的最好算法。但对于大规模的优化问题，拟 Newton 需要较大的存储空间，并且矩阵运算所耗费时间的弊端也突显出来。

SP++中提供了BFGS拟Newton算法，其调用格式与最速下降法以及共轭梯度法相同，详见表表 8.4。

表 8.4 拟 Newton 法

Operation	Effect
BFGS <DType,Ftype> fmin	创建拟 Newton 法类
fmin.~ BFGS <DType,Ftype>()	析构拟 Newton 法类
fmin. optimize(func, x0, tol, maxItr)	求指定参数的函数最小值
fmin.getOptValue()	获取自变量的最优值
fmin.getGradNorm()	获取迭代过程中梯度向量的模值
fmin.getFuncMin()	获取函数的最小值

fmin.getItrNum()	获取迭代次数

BFGS 的测试代码：

```
/*************************************************************************
 *                              bfgs_test.cpp
 *
 * BFGS method testing.
 *
 * Zhang Ming, 2010-03
 *************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <function.h>
#include <bfgs.h>


using namespace std;
using namespace itlab;


typedef     double  Type;


int main()
{
    Type a = 1.0,
         b = -1.0,
         c = -1.0;
    ObjFunc<Type> f( a, b, c );
    Vector<Type> x0(2);
    x0(1) = Type(0.0);
    x0(2) = Type(0.0);

    BFGS< Type, ObjFunc<Type> > bfgs;
    bfgs.optimize( f, x0 );
    if( bfgs.isSuccess() )
    {
        Vector<Type> xmin = bfgs.getOptValue();
        int N = bfgs.getItrNum();
        cout << "The iterative number is:   " << N << endl << endl;
        cout << "The number of function calculation is:   " << bfgs.getFuncNum() << endl << endl;
```

```
42.          cout << "The optimal value of x is:   " << xmin << endl;
43.          cout << "The minimum value of f(x) is:   " << f(xmin) << endl << endl;
44.          cout << "The gradient's norm at x is:   " << bfgs.getGradNorm()[N] << endl << endl;
45.      }
46.      else
47.          cout << "The optimal solution  cann't be found!" << endl;
48.
49.      return 0;
50.  }
```

运行结果：

```
1.   The iterative number is:   7
2.
3.   The number of function calculation is:   16
4.
5.   The optimal value of x is:   size: 2 by 1
6.   -0.707107
7.   0
8.
9.   The minimum value of f(x) is:   -0.428882
10.
11.  The gradient's norm at x is:   2.06179e-009
```

9 插值与拟合

9.1 Newton 插值

对于给定的 N+1 个点，可以通过 Newtow 插值法求取经过这些点的 N 次多项式。但该插值法主要针对底阶的插值多项式，因为高阶多项式一用会出现剧烈的振荡，所以在插值点之间会引入非常大的误差。

SP++中Newton插值法类NewtonInterp<Type>的使用方法见表 9.1，并且与三次样条和最小二乘拟合相同，都继承了一般的插值类模板Interpolation<Type>，有关该类的声明可参见”interpolation.h”。

表 9.1 Newton 插值法

Operation	Effect
NewtonInterp<Type> intp(xi,yi)	创建 Newton 插值类
intp.~ NewtonInterp<Type>	析构 Newton 插值类
intp.calcCoefs()	计算插值多项式的系数
intp.evaluate(x)	计算给定坐标的函数值
intp.getCoefs()	获取插值多项式的系数

Newton 插值法的测试代码：

```
/*****************************************************************************
 *                              newtoninterp_test.cpp
 *
 * Newton interpolation testing.
 *
 * Zhang Ming, 2010-04
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <constants.h>
#include <newtoninterp.h>


using namespace std;
```

```
using namespace itlab;


int main()
{
    Vector<double> x(5),
                   y(5);
    x[0] = 0; x[1] = 30;  x[2] = 45;        x[3] = 60;        x[4] = 90;
    y[0] = 0; y[1] = 0.5; y[2] = sqrt(2)/2; y[3] = sqrt(3)/2; y[4] = 1;

    NewtonInterp<double> poly(x,y);
    poly.calcCoefs();

    cout << "Coefficients of Newton interpolated polynomial:"
         << poly.getCoefs() << endl;

    cout << "The true and interpolated values:" << endl;
    cout << sin(10*D2R) << "   " << poly.evaluate(10) << endl
         << sin(20*D2R) << "   " << poly.evaluate(20) << endl
         << sin(50*D2R) << "   " << poly.evaluate(50) << endl << endl;

    return 0;
}
```

运行结果：

```
Coefficients of Newton interpolated polynomial:size: 5 by 1
0
0.0166667
-6.35455e-005
-7.25655e-007
2.67214e-009

The true and interpolated values:
0.173648   0.173361
0.34202   0.34188
0.766044   0.766026
```

9.2 三次样条插值

三次样条插值法对每个插值区间用三次多项式进行近似，并且保证区间单点插值多项式与其一阶导数均连续。因此所得的插值多项式有很好的性质，并且对于多点插

值有很高的逼近程度。SP++中三次样条的使用方法与Newton相同，见表 9.2。

表 9.2　三次样条插值法

Operation	Effect
Spline3Interp<Type> intp(xi,yi,d2l,dwr)	创建三次样条插值类
intp.~ Spline3Interp<Type>	析构三次样条插值类
intp.calcCoefs()	计算插值多项式的系数
intp.evaluate(x)	计算给定坐标的函数值
intp.getCoefs()	获取插值多项式的系数

三次样条插值法的测试代码：

```
/*****************************************************************************
 *                              spline3interp_test.cpp
 *
 * Spline3 interpolation testing.
 *
 * Zhang Ming, 2010-04
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <constants.h>
#include <spline3interp.h>


using namespace std;
using namespace itlab;


int main()
{
    // f(x) = 1 / (1+25*x^2)   -1 <= x <= 1
    float xi[11] = { -1.0,   -0.8,   -0.6, -0.4, -0.2, 0.0, 0.2, 0.4, 0.6, 0.8,   1.0 };
    float yi[11] = { 0.0385, 0.0588, 0.1,  0.2,  0.5,  1.0, 0.5, 0.2, 0.1, 0.0588, 0.0385 };
    float Ml = 0.2105, Mr = 0.2105;
    Vector<float> x( 11, xi ), y( 11, yi );

    Spline3Interp<float> poly( x, y, Ml, Mr );
    poly.calcCoefs();

    cout << "Coefficients of cubic spline interpolated polynomial:" << endl
         << poly.getCoefs() << endl << endl;
```

```
34.
35.       cout << "The true and interpolated values:" << endl;
36.       cout << "0.0755" << "    " << poly.evaluate(-0.7) << endl
37.            << "0.3077" << "    " << poly.evaluate(0.3) << endl
38.            << "0.0471" << "    " << poly.evaluate(0.9) << endl << endl;
39.
40.       return 0;
41.   }
```

运行结果：

```
1.    Coefficients of cubic spline interpolated polynomial:
2.    size: 10 by 4
3.    0.0385 0.0736753 0.10525 0.169368
4.    0.0588 0.129083 0.206871 0.888576
5.    0.1 0.31846 0.740016 0.838411
6.    0.2 0.715076 1.24306 13.4078
7.    0.5 2.82124 9.28773 -54.4695
8.    1 -1.6811e-005 -23.394 54.4704
9.    0.5 -2.82117 9.28824 -13.412
10.   0.2 -0.715311 1.24105 -0.822463
11.   0.1 -0.317587 0.74757 -0.948167
12.   0.0588 -0.132339 0.17867 -0.122367
13.
14.   The true and interpolated values:
15.   0.0755    0.0746656
16.   0.3077    0.297354
17.   0.0471    0.0472304
```

9.3 最小二乘拟合

给定观测值来求其所遵从的函数关系时需要进行曲线拟合，最常用的一类拟合方法就是最小二乘拟合，该方法以观测数据与待拟函数之间的均方误差为准则求取函数参数。实际中比较常用的是线性最小二乘拟合（许多非线性最小二乘拟合可以转化为线性最小二乘拟合），即待拟合的函数是已知的一簇函数的线性组合。

SP++中提供了线性最小二乘拟合类LSFitting<Type>，其中构造函数中”f”是已知的函数簇，其定义见”functions.h”， 具体使用方法见表 9.3。

表 9.3 最小二乘拟合

Operation	Effect
LSFitting<Type> lsf(xi,yi,f)	创建最小二乘拟合类
lsf.~ LSFitting<Type>	析构最小二乘拟合类

lsf.calcCoefs()	计算拟合函数的系数
lsf.evaluate(x)	计算给定坐标的函数值
lsf.getCoefs()	获取拟合函数的系数

最小二乘拟合的测试代码：

```
/*****************************************************************************
 *                              lsfit_test.cpp
 *
 * Least square fitting testing.
 *
 * Zhang Ming, 2010-04
 *****************************************************************************/


#define BOUNDS_CHECK

#include <iostream>
#include <lsfitting.h>
#include <utilities.h>


using namespace std;
using namespace itlab;


int main()
{
    double  A = 4.0,
            alpha = 1.0,
            beta = -2.0,
            gamma = -4.0,
            tmp = 0.0;

    int M = 100;
    Vector<double> x = linspace( 0.01, 0.5*PI, M );
    Vector<double> y(M);
    for( int i=0; i<M; ++i )
    {
        tmp = A * pow(x[i],alpha) * exp(beta*x[i]+gamma*x[i]*x[i]);
        y[i] = log(max(tmp,EPS));
    }

    Funcs<double> phi;
    LSFitting<double> lsf( x, y, phi );
```

```
40.       lsf.calcCoefs();
41.       Vector<double> parms = lsf.getCoefs();
42.       parms(1) = exp(parms(1));
43.
44.       cout << "The original parameters are:" << endl
45.            << A << endl << alpha << endl << beta << endl << gamma << endl << endl;
46.       cout << "The fitted parameters are:" << endl << parms << endl;
47.
48.       return 0;
49.   }
```

运行结果：

```
1.    The original parameters are:
2.    4
3.    1
4.    -2
5.    -4
6.
7.    The fitted parameters are:
8.    size: 4 by 1
9.    4
10.   1
11.   -2
12.   -4
```