

SP++ User Guide

张明

2010-07

目录

0	SP++导引	1
0.1	SP++概述	1
0.2	SP++安装	1
1	向量类模板	5
1.1	基本向量类	5
1.2	常用函数的向量版本	14
1.3	Matlab 中的常用函数	19
1.4	概率统计中的常用函数	27
2	矩阵类模板	31
2.1	基本矩阵类	31
2.2	Cholesky 分解	41
2.3	LU 分解	44
2.4	QR 分解	46
2.5	SVD 分解	49
2.6	EVD 分解	51
3	线性方程组	55
3.1	常规线性方程组求解	55
3.2	超定与欠定线性方程组求解	59
3.3	病态线性方程组求解	62
4	插值与拟合	65
4.1	Newton 插值	65
4.2	三次样条插值	66
4.3	最小二乘拟合	68
5	优化算法	71
5.1	一维线搜索	71
5.2	最速下降法	71
5.3	共轭梯度法	73
5.4	拟 Newton 法	75
6	Fourier 分析	79
6.1	FFTW 的 C++接口	79
6.2	2 的整次幂 FFT 算法	82
6.3	卷积与相关快速算法	87
7	数字滤波器设计	91
7.1	常用窗函数	91
7.2	滤波器基类设计	93
7.3	FIR 数字滤波器设计	93
7.4	IIR 数字滤波器设计	96
8	自适应滤波器	101
8.1	Wiener 滤波器	101
8.2	Calman 滤波器	101

8.3	LMS 自适应滤波器.....	101
9	时频分析.....	103
9.1	加窗 Fourier 变换.....	103
9.2	离散 Gabor 变换.....	105
9.3	Wigner-Wille 分布.....	108
10	小波变换.....	109
10.1	连续小波变换.....	109
10.2	二进小波变换.....	112
10.3	离散小波变换.....	114

0 SP++导引

0.1 SP++概述

SP++(Signal Processing in C++)是一个信号处理的 C++开源库，该库提供了线性代数与信号处理领域中常用的基本算法的 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 SP++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，所以项目名取为TSPL(Template Signal Processing Library)。

0.2 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->Compiler and Debugger / Search directories / Compiler中加入D:\Program Files\SP++\include，如图 0.1所示。
- 3 同第 2 步，在Linker中加入D:\Program Files\SP++\lib，如图 0.2所示。
- 4 在 Settings->Compiler and Debugger / Link libraries 中 加入 D:\Program Files\SP++\libfftw3f.a与D:\Program Files\SP++\libfftw3.a，如图 0.3所示。

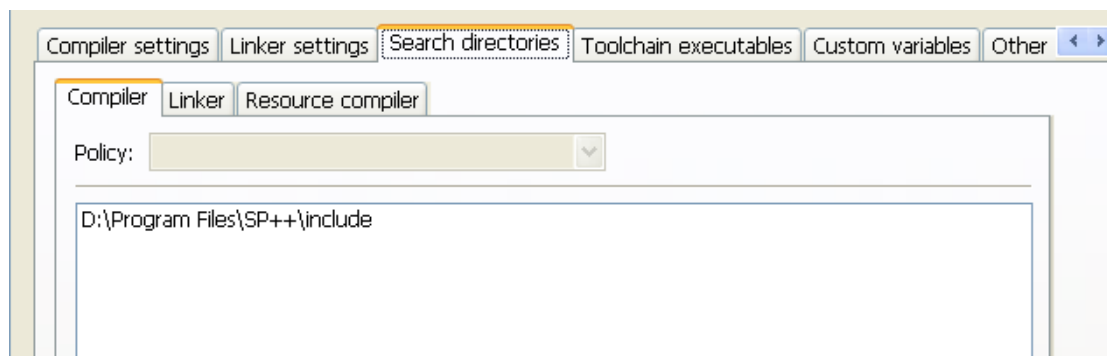


图 0.1

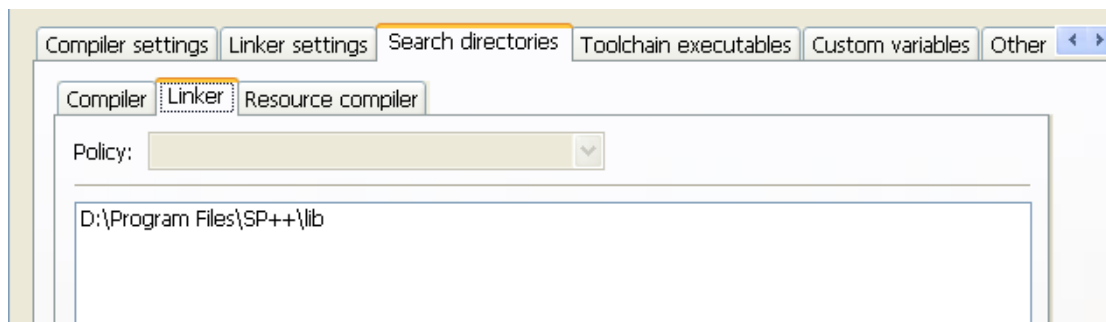


图 0.2

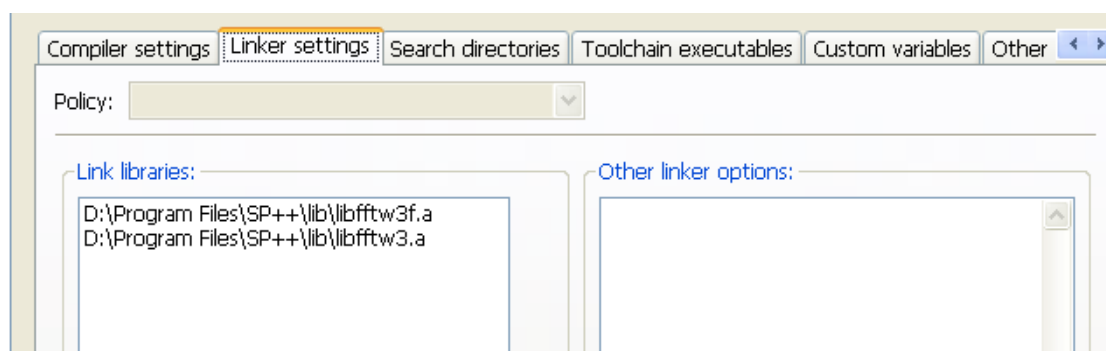


图 0.3

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, Xi'an Jiaotong University.
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.
24. // string fType = "lowpass";
25. // double fs = 1000,
26. //      fpass = 200,
27. //      apass = -3,
28. //      fstop = 300,
29. //      astop = -20;
30. // FIR fir( fType, wType );
31. // fir.setParams( fs, fpass, apass, fstop, astop );
32.
33. // string fType = "highpass";
34. // double fs = 1000,
35. //      fstop = 200,
36. //      astop = -20,
37. //      fpass = 300,
38. //      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)
27.

```


1 向量类模板

1.1 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 1.1；向量类的基本属性提取，见表 1.2；向量常用算法的运算符重载，见表 1.3；以及一些其它常用函数，见表 1.4。

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 1.1 向量类的构造与析构函数

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

表 1.2 向量类的属性获取

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

表 1.3 向量类中重载的运算符

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

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

表 1.4 向量类的其它函数

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

测试代码:

```

1.  /*****
2.   *                               vector_test.cpp
3.   *
4.   * Vector class testing.
5.   *
6.   * Zhang Ming, 2010-01, Xi'an Jiaotong University.
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;

```

向量类模板

```
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;
107.     v3 /= v1;
108.     cout << "v3 /= v1 : " << v3 << endl;
109.
110.     k = dotProd( v1, v2 );
111.     cout << "inner product of v1 and v2 : " << endl;
112.     cout << k << endl << endl;
113.
114.     cout << "L2 norm of v3 : ";
115.     cout << norm( v3 ) << endl << endl;
116.
117.     complex<double> z = -1.0;
118.     Vector< complex<double> > v( M );
119.     v[0] = polar( 1.0,PI/4 );
120.     v[1] = polar( 1.0,PI );
121.     v[2] = complex<double>( 1.0,-1.0 );
122.     Vector< complex<double> > u = v*z;
123.
124.     cout << "complex vector v : " << v << endl;
125.     cout << "complex vector u = -v : " << u << endl;
126.
127.     Vector< Vector<double> > v2d1( M );
128.     for( int i=0; i<M; ++i )
129.     {
130.         v2d1[i].resize( M );
131.         for( int j=0; j<M; ++j )
132.             v2d1[i][j] = double( i+j );
133.     }
134.
135.     cout << "two dimention vector v2d1 : " << endl;
136.     Vector< Vector<double> >::const_iterator itrD2 = v2d1.begin();
137.     int rowNum = 0;
138.     while( itrD2 != v2d1.end() )
139.         cout << "the " << rowNum++ << "th row : " << *itrD2++ << endl;
140.
141.     Vector< Vector<double> > v2d2 = v2d1+v2d1;
142.     cout << "two dimention vector v2d2 = v2d1 + v2d1 : " << v2d2 << endl;
143.
144.     return 0;
145. }

```

运行结果:

```

1.   vector v1 : size: 3 by 1

```

向量类模板

```
2.  1
3.  2
4.  3
5.
6.  vector v2 : size: 3 by 1
7.  1
8.  1
9.  1
10.
11. vector v3 : size: 3 by 1
12. 0
13. 0
14. 0
15.
16. vector v4 : size: 3 by 1
17. 1
18. 2
19. 3
20.
21. 1      2      3
22.
23. new vector v4 : size: 5 by 1
24. 1
25. 1
26. 1
27. 1
28. 1
29.
30. new vector v4 : size: 3 by 1
31. 1
32. 2
33. 3
34.
35. reversion of vector v4 : size: 3 by 1
36. 3
37. 2
38. 1
39.
40. v3 = k + v1 : size: 3 by 1
41. 3
42. 4
43. 5
44.
45. v3 += k : size: 3 by 1
```

```
46. 5
47. 6
48. 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.
```

向量类模板

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. (-1,1.22461e-016)
138. (1,-1)
139.
140. complex vector u = -v : size: 3 by 1
141. (-0.707107,-0.707107)
142. (1,-1.22461e-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
176.

```

1.2 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 1.5所示。

表 1.5 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

测试代码：

```
1.  /*****
2.  *                               mathfunc_test.cpp
3.  *
4.  * Math functions for vector and matrix testing.
5.  *
6.  * Zhang Ming, 2010-01
7.  *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <utilities.h>
14. #include <mathfunc.h>
15.
16.
17. using namespace std;
18. using namespace itlab;
19.
```

```

20.
21.  int main()
22.  {
23.
24.      int    N = 11;
25.      float  a = 0,
26.            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

```

向量类模板

```
7. 3.14159
8. 3.76991
9. 4.39823
10. 5.02655
11. 5.65487
12. 6.28319
13.
14. Sine function of x : size: 11 by 1
15. 0
16. 0.587785
17. 0.951057
18. 0.951056
19. 0.587785
20. -8.74228e-008
21. -0.587785
22. -0.951056
23. -0.951056
24. -0.587785
25. 1.74846e-007
26.
27. Cosine function of x : size: 11 by 1
28. 1
29. 0.809017
30. 0.309017
31. -0.309017
32. -0.809017
33. -1
34. -0.809017
35. -0.309017
36. 0.309017
37. 0.809017
38. 1
39.
40. Exponential function of x : size: 11 by 1
41. 1
42. 1.87446
43. 3.51359
44. 6.58606
45. 12.3453
46. 23.1407
47. 43.3762
48. 81.3068
49. 152.406
50. 285.679
51. 535.492
```

```

52.
53. Power function of x : size: 11 by 1
54. 0
55. 0.394784
56. 1.57914
57. 3.55306
58. 6.31655
59. 9.86961
60. 14.2122
61. 19.3444
62. 25.2662
63. 31.9775
64. 39.4784
65.
66. Complex vector vc : size: 11 by 1
67. (0,1)
68. (0.587785,0.809017)
69. (0.951057,0.309017)
70. (0.951056,-0.309017)
71. (0.587785,-0.809017)
72. (-8.74228e-008,-1)
73. (-0.587785,-0.809017)
74. (-0.951056,-0.309017)
75. (-0.951056,0.309017)
76. (-0.587785,0.809017)
77. (1.74846e-007,1)
78.
79. Real part of vc : size: 11 by 1
80. 0
81. 0.587785
82. 0.951057
83. 0.951056
84. 0.587785
85. -8.74228e-008
86. -0.587785
87. -0.951056
88. -0.951056
89. -0.587785
90. 1.74846e-007
91.
92. Imaginary part of vc : size: 11 by 1
93. 1
94. 0.809017
95. 0.309017
96. -0.309017

```

向量类模板

```
97. -0.809017
98. -1
99. -0.809017
100. -0.309017
101. 0.309017
102. 0.809017
103. 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
149. 0.129518
150. 0.241971
151. 0.352065
152. 0.398942
153. 0.352065
154. 0.241971
155. 0.129518
156. 0.053991
157. 0.0175283
158. 0.00443185
159. 0.000872683
160. 0.00013383
161. 1.59837e-005
162. 1.48672e-006

```

1.3 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 1.6列出了信号处理中经常使用的一些函数的C++实现。

表 1.6 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

测试代码:

```

1.  /*****
2.      *                               utilities_test.cpp
3.      *
4.      * Utilities class testing.
5.      *
6.      * Zhang Ming, 2010-01, Xi'an Jiaotong University.
7.      *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <string>
14. #include <utilities.h>
15.
16.
17. using namespace std;
18. using namespace itlab;
19.
20.
21. const int N = 5;
22.
23.
24. int main()
25. {
26.
27.     Vector<int> v1(N);
28.     for( int i=1; i<=v1.dim(); i++ )
29.         v1(i) = i;
30.     cout << "vector v1 : " << v1 << endl;
31.
32.     Vector<int> v2(N);
33.     for( int i=1; i<=v2.dim(); ++i )
34.         v2(i) = i+N;
35.     cout << "vector v2 : " << v2 << endl;
36.
37.     int N = 11;
38.     double a = 0;
39.     double b = 1.0;
40.     Vector<double> x = linspace( a, b, N );
41.     cout << N << " points linearly spaced from 0 to 1.0"
42.         << x << endl;
43.

```



```

44.     cout << "Flipping vector v1 from left to right : " << flip(v1) << endl;
45.     cout << "Dyadic upsampling of vector v1 by zeros at the even position : "
46.         << dyadUp( v1,0 ) << endl;
47.     cout << "Dyadic upsampling of vector v1 by zeros at the odd position : "
48.         << dyadUp( v1,1 ) << endl;
49.     cout << "Dyadic downsampling of vector v1 by zeros at the even position : "
50.         << 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 direction by zeros padding : "
58.         << wextend( v1,n,dire,mode ) << endl;
59.     mode = "ppd";
60.     cout << "Extending vector v1 in left direction by periodic mode : "
61.         << wextend( v1,n,dire,mode ) << endl;
62.     mode = "sym";
63.     cout << "Extending vector v1 in left direction by symetric mode : "
64.         << wextend( v1,n,dire,mode ) << endl;
65.
66.     dire = "right";
67.     mode = "zpd";
68.     cout << "Extending vector v1 in right direction by zeros padding : "
69.         << wextend( v1,n,dire,mode ) << endl;
70.     mode = "ppd";
71.     cout << "Extending vector v1 in right direction by periodic mode : "
72.         << wextend( v1,n,dire,mode ) << endl;
73.     mode = "sym";
74.     cout << "Extending vector v1 in right direction by symetric mode : "
75.         << wextend( v1,n,dire,mode ) << endl;
76.
77.     dire = "both";
78.     mode = "zpd";
79.     cout << "Extending vector v1 in both direction by zeros padding : "
80.         << wextend( v1,n,dire,mode ) << endl;
81.     mode = "ppd";
82.     cout << "Extending vector v1 in both direction by periodic mode : "
83.         << wextend( v1,n,dire,mode ) << endl;
84.     mode = "sym";
85.     cout << "Extending vector v1 in both direction 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.
28. Flipping vector v1 from left to right : size: 5 by 1
29. 5
30. 4
31. 3
32. 2
33. 1
34.
35. Dyadic upsampling of vector v1 by zeros at the even position : size: 11 by 1
36. 0
37. 1
38. 0
39. 2
40. 0
41. 3
42. 0
43. 4
44. 0
45. 5
46. 0
47.
48. Dyadic upsampling of vector v1 by zeros at the odd position : size: 9 by 1
49. 1
50. 0
51. 2
52. 0
53. 3
54. 0
55. 4
56. 0
57. 5
58.
59. Dyadic downsampling of vector v1 by zeros at the even position : size: 3 by 1
60. 1
61. 3
62. 5
```

63.

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

65. 2

66. 4

67.

68. Extending vector v1 in left dirction by zeros padding : size: 7 by 1

69. 0

70. 0

71. 1

72. 2

73. 3

74. 4

75. 5

76.

77. Extending vector v1 in left dirction by periodic mode : size: 7 by 1

78. 4

79. 5

80. 1

81. 2

82. 3

83. 4

84. 5

85.

86. Extending vector v1 in left dirction by symetric mode : size: 7 by 1

87. 2

88. 1

89. 1

90. 2

91. 3

92. 4

93. 5

94.

95. Extending vector v1 in right dirction by zeros padding : size: 7 by 1

96. 1

97. 2

98. 3

99. 4

100. 5

101. 0

102. 0

103.

104. Extending vector v1 in right dirction by periodic mode : size: 7 by 1

105. 1

106. 2

```
107. 3
108. 4
109. 5
110. 1
111. 2
112.
113. Extending vector v1 in right dirction by symetric mode : size: 7 by 1
114. 1
115. 2
116. 3
117. 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
163. 3
164.
165. Keeping the right part of vector v1 : size: 3 by 1
166. 3
167. 4
168. 5
169.
170. Keeping the first(2) to first + L(3) elements of vector v1 : size: 3 by 1
171. 2
172. 3
173. 4
174.
175. The convolution of vector v1 and v2 : size: 9 by 1
176. 6
177. 19
178. 40
179. 70
180. 110
181. 114
182. 106
183. 85
184. 50
185.
186. The modulus of 2 divided by 5 is 2.
187. The modulus of -1 divided by 5 is 4.
188.
189. The nearest integer  $\geq 10/2$  is 5.
190. The nearest integer  $\geq 10/3$  is 4.
191.
192. The numbers can be represented by the integer power of 2 from 0 to 1000 are :
193.  $0 = \log_2(1)$ 
194.  $1 = \log_2(2)$ 
```

```
195. 2 = log2(4)
196. 3 = log2(8)
197. 4 = log2(16)
198. 5 = log2(32)
199. 6 = log2(64)
200. 7 = log2(128)
201. 8 = log2(256)
202. 9 = log2(512)
203.
```

1.4 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 1.7。

表 1.7 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

测试代码：

```
1.  /*****
2.   *                               statistics_test.cpp
3.   *
4.   * Statistics routines testing.
5.   *
6.   * Zhang Ming, 2010-03, Xi'an Jiaotong University.
7.   *****/
8.
9.
10. #define BOUNDS_CHECK
11.
```

```

12. #include <iostream>
13. #include <statistics.h>
14.
15.
16. using namespace std;
17. using namespace itlab;
18.
19.
20. int main()
21. {
22.     Vector<double> xn(9);
23.     xn[0] = 1.2366;    xn[1] = -0.6313;    xn[2] = -2.3252;
24.     xn[3] = -1.2316;  xn[4] = 1.0556;    xn[5] = -0.1132;
25.     xn[6] = 0.3792;   xn[7] = 0.9442;    xn[8] = -2.1204;
26.
27.     cout << "The minnum maximum and median value of the sequence." << endl;
28.     cout << min(xn) << endl << max(xn) << endl << mid(xn) << endl << endl;
29.     cout << "The mean, variance and standard variance of the sequence." << endl;
30.     cout << mean(xn) << endl << var(xn) << endl << stdVar(xn) << endl << endl;
31.
32.     Vector<double> yn = xn - mean(xn);
33.     cout << "The skew and kurtosis of the sequence." << endl;
34.     cout << skew(yn) << endl << kurt(yn) << endl << endl;
35.
36.     cout << "The PDF of the sequence." << endl;
37.     cout << pdf(yn) << endl;
38.
39.     return 0;
40. }

```

运行结果:

```

1.  The minnum maximum and median value of the sequence.
2.  -2.3252
3.  1.2366
4.  -0.1132
5.
6.  The mean, variance and standard variance of the sequence.
7.  -0.311789
8.  1.82934
9.  1.35253
10.
11. The skew and kurtosis of the sequence.
12. -0.282956
13. -1.67782

```



```

14.
15. The PDF of the sequence.
16. size: 5 by 1
17. 0.000512486
18. 0.134665
19. 0.333867
20. 0.42622
21. 0.104735
22.

```


2 矩阵类模板

2.1 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 2.1；矩阵类的基本属性提取，见表 2.2；矩阵常用算法的运算符重载，见表 2.3；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 2.4。具体的函数声明与定义可以参见“`matrix.h`”和“`matrix-impl.h`”。

表 2.1 矩阵类的构造与析构函数

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

表 2.2 矩阵类的属性获取

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

表 2.3 矩阵类中重载的运算符

Operation	Effect
<code>A1 = A2</code>	矩阵对矩阵赋值
<code>A = x</code>	常数对矩阵赋值
<code>A[i][j]</code>	0 偏移下标访问
<code>A(i,j)</code>	1 偏移下标访问
<code>-A</code>	全部元素取反
<code>A + x</code>	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 2.4 矩阵类的其它函数

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

测试代码:

```

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

```

```

6.  * Zhang Ming, 2010-01, Xi'an Jiaotong University.
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.
36. matrix-scalar operand
37.
38. scalar x = 2
39. m1 + x : size: 3 by 3
40. 3 3 3
41. 3 3 3
42. 3 3 3
43.
44. x + m1 : size: 3 by 3
45. 3 3 3
46. 3 3 3
47. 3 3 3
48.
49. m1 += x : size: 3 by 3
50. 3 3 3
51. 3 3 3
52. 3 3 3
53.
54. m1 - x : size: 3 by 3
55. 1 1 1

```



```

56. 1 1 1
57. 1 1 1
58.
59. x - m1 : size: 3 by 3
60. -1 -1 -1
61. -1 -1 -1
62. -1 -1 -1
63.
64. m1 -= x : size: 3 by 3
65. 1 1 1
66. 1 1 1
67. 1 1 1
68.
69. m1 * x : size: 3 by 3
70. 2 2 2
71. 2 2 2
72. 2 2 2
73.
74. x * m1 : size: 3 by 3
75. 2 2 2
76. 2 2 2
77. 2 2 2
78.
79. m1 *= x : size: 3 by 3
80. 2 2 2
81. 2 2 2
82. 2 2 2
83.
84. m1 / x : size: 3 by 3
85. 1 1 1
86. 1 1 1
87. 1 1 1
88.
89. x / m1 : size: 3 by 3
90. 1 1 1
91. 1 1 1
92. 1 1 1
93.
94. m1 /= x : size: 3 by 3
95. 1 1 1
96. 1 1 1
97. 1 1 1
98.
99.

```

矩阵类模板

```
100.  
101.                                     elementwise matrix-matrix operand  
102.  
103. m1 + m2 : size: 3 by 3  
104. 3 3 3  
105. 3 3 3  
106. 3 3 3  
107.  
108. m1 += m2 : size: 3 by 3  
109. 3 3 3  
110. 3 3 3  
111. 3 3 3  
112.  
113. m1 - m2 : size: 3 by 3  
114. 1 1 1  
115. 1 1 1  
116. 1 1 1  
117.  
118. m1 -= m2 : size: 3 by 3  
119. 1 1 1  
120. 1 1 1  
121. 1 1 1  
122.  
123. m1 * m2 : size: 3 by 3  
124. 2 2 2  
125. 2 2 2  
126. 2 2 2  
127.  
128. m1 *= m2 : size: 3 by 3  
129. 2 2 2  
130. 2 2 2  
131. 2 2 2  
132.  
133. m1 / m2 : size: 3 by 3  
134. 1 1 1  
135. 1 1 1  
136. 1 1 1  
137.  
138. m1 /= m2 : size: 3 by 3  
139. 1 1 1  
140. 1 1 1  
141. 1 1 1  
142.  
143.
```

```

144.
145.                                matrix-matrix operand
146.
147. m1 * m2 : size: 3 by 3
148. 6 6 6
149. 6 6 6
150. 6 6 6
151.
152. m1' * m2 : size: 3 by 3
153. 6 6 6
154. 6 6 6
155. 6 6 6
156.
157.
158.
159.                                matrix-vector operand
160.
161. vector v1 : size: 3 by 1
162. 2
163. 2
164. 2
165.
166. m1 * v1 : size: 3 by 1
167. 6
168. 6
169. 6
170.
171. m1' * v1 : size: 3 by 1
172. 6
173. 6
174. 6
175.
176. Please input a sqare matrix mtr.
177. 2 2
178. 2 -0.5
179. -2 1
180. the inverse matrix of mtr : size: 2 by 2
181. 1 0.5
182. 2 2
183.
184. matrix m4 : size: 4 by 5
185. 1 2 3 4 5
186. 2 4 6 8 10
187. 3 6 9 12 15

```

矩阵类模板

```
188. 4 8 12 16 20
189.
190. column vectors of m4 :
191. the 1th columnsize: 4 by 1
192. 1
193. 2
194. 3
195. 4
196.
197. the 2th columnsize: 4 by 1
198. 2
199. 4
200. 6
201. 8
202.
203. the 3th columnsize: 4 by 1
204. 3
205. 6
206. 9
207. 12
208.
209. the 4th columnsize: 4 by 1
210. 4
211. 8
212. 12
213. 16
214.
215. the 5th columnsize: 4 by 1
216. 5
217. 10
218. 15
219. 20
220.
221. row vectors of m4 :
222. the 1th rowsize: 5 by 1
223. 1
224. 2
225. 3
226. 4
227. 5
228.
229. the 2th rowsize: 5 by 1
230. 2
231. 4
```

```

232. 6
233. 8
234. 10
235.
236. the 3th rowsize: 5 by 1
237. 3
238. 6
239. 9
240. 12
241. 15
242.
243. the 4th rowsize: 5 by 1
244. 4
245. 8
246. 12
247. 16
248. 20
249.

```

2.2 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 2.5。

表 2.5 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 $AX = B$

测试代码：

```

1.  /*****
2.  *                               cholesky_test.cpp
3.  *
4.  * Cholesky class testing.
5.  *

```

```

6.  * Zhang Ming, 2010-01, Xi'an Jiaotong University.
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
35.
```

2.3 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 2.6。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 2.6 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

测试代码:

```
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
35.

```

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
7. 1 0 0
8. 0.5 1 0
9. 0.5 0.333333 1
10.
11. The upper triangular matrix U is : size: 3 by 3
12. 2 5 4
13. 0 -1.5 -2
14. 0 0 -0.333333
15.
16. The constant vector b : size: 3 by 1

```

```
17. 1
18. 0
19. 1
20.
21. The solution of A * x = b : size: 3 by 1
22. -1
23. 2
24. -2
25.
26. The invse matrix of A : size: 3 by 3
27. -4 1 3
28. 4 -1 -2
29. -3 1 1
30.
31. The determinant of A :
32. 1
33.
```

2.4 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 $A=Q \cdot R$ ，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 2.7。

表 2.7 QR 分解

Operation	Effect
QRD<Real> qr	建立 LUD 类
qr.~QRD<Real>()	LUD 析构造函数
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 $AX = B$

测试代码：

```
1.  /*****
2.  *                               qrd_test.cpp
3.  *
4.  * QRD class testing.
5.  *
```

```

6.  * Zhang Ming, 2010-01, Xi'an Jiaotong University.
7.  *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <qrd.h>
14.
15.
16. using namespace std;
17. using namespace itlab;
18.
19.
20. const int M = 4;
21. const int N = 3;
22.
23.
24. int main()
25. {
26.     Matrix<double> A(M,N);
27.     A[0][0] = 1;   A[0][1] = 0;   A[0][2] = 0;
28.     A[1][0] = 1;   A[1][1] = 2;   A[1][2] = 4;
29.     A[2][0] = 1;   A[2][1] = 3;   A[2][2] = 9;
30.     A[3][0] = 1;   A[3][1] = 3;   A[3][2] = 9;
31.
32.     Matrix<double> B(M,M);
33.     B[0][0] = 1;   B[0][1] = 0;   B[0][2] = 0;   B[0][3] = 0;
34.     B[1][0] = 0;   B[1][1] = 1;   B[1][2] = 0;   B[1][3] = 0;
35.     B[2][0] = 0;   B[2][1] = 0;   B[2][2] = 1;   B[2][3] = 0;
36.     B[3][0] = 0;   B[3][1] = 0;   B[3][2] = 0;   B[3][3] = 1;
37.
38.     Vector<double> b(M);
39.     b[0] = 1;     b[1] = 0;     b[2] = 1, b[3] = 2;
40.
41.     Matrix<double> Q, R;
42.     QRD<double> qr;
43.     qr.dec(A);
44.     qr.getQ(Q);
45.     qr.getR(R);
46.
47.     cout << "The original matrix A : " << A << endl;
48.     cout << "The orthogonal matrix Q : " << Q << endl;
49.     cout << "The upper triangular matrix R : " << R << endl;

```

```

50.
51.     Vector<double> x = qr.solve(b);
52.     cout << "The constant vector b : " << b << endl;
53.     cout << "The least squares solution of A * x = b : " << x << endl;
54.
55.     Matrix<double> X = qr.solve(B);
56.     cout << "The constant matrix B : " << B << endl;
57.     cout << "The least squares solution of A * X = B : " << X << endl;
58.
59.     return 0;
60. }

```

运行结果:

```

1.  The original matrix A : size: 4 by 3
2.  1 0 0
3.  1 2 4
4.  1 3 9
5.  1 3 9
6.
7.  The orthogonal matrix Q : size: 4 by 3
8.  -0.5 0.816497 -0.288675
9.  -0.5 0 0.866025
10. -0.5 -0.408248 -0.288675
11. -0.5 -0.408248 -0.288675
12.
13. The upper triangular matrix R : size: 3 by 3
14. -2 -4 -11
15. 0 -2.44949 -7.34847
16. 0 0 -1.73205
17.
18. The constant vector b : size: 4 by 1
19. 1
20. 0
21. 1
22. 2
23.
24. The least squares solution of A * x = b : size: 3 by 1
25. 1
26. -1.83333
27. 0.666667
28.
29. The constant matrix B : size: 4 by 4
30. 1 0 0 0
31. 0 1 0 0

```

```

32. 0 0 1 0
33. 0 0 0 1
34.
35. The least squares solution of A * X = B : size: 3 by 4
36. 1 -0 1.11022e-016 -0
37. -0.833333 1.5 -0.333333 -0.333333
38. 0.166667 -0.5 0.166667 0.166667
39.

```

2.5 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 n 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2.8。

表 2.8 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

测试代码：

```

1.  /*****
2.  *                               svd_test.cpp
3.  *
4.  * SVD class testing.
5.  *
6.  * Zhang Ming, 2010-01, Xi'an Jiaotong University.
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. int main()
21. {
22.     // Matrix<double> A(4,4);
23.     // A(1,1) = 16;    A(1,2) = 2;    A(1,3) = 3;    A(1,4) = 13;
24.     // A(2,1) = 5;    A(2,2) = 11;   A(2,3) = 10;   A(2,4) = 8;
25.     // A(3,1) = 9;    A(3,2) = 7;    A(3,3) = 6;    A(3,4) = 12;
26.     // A(4,1) = 4;    A(4,2) = 14;   A(4,3) = 15;   A(4,4) = 1;
27.
28.     // Matrix<double> A(4,2);
29.     // A(1,1) = 1;    A(1,2) = 2;
30.     // A(2,1) = 3;    A(2,2) = 4;
31.     // A(3,1) = 5;    A(3,2) = 6;
32.     // A(4,1) = 7;    A(4,2) = 8;
33.
34.     Matrix<double> A(2,4);
35.     A(1,1) = 1;    A(1,2) = 3;    A(1,3) = 5;    A(1,4) = 7;
36.     A(2,1) = 2;    A(2,2) = 4;    A(2,3) = 6;    A(2,4) = 8;
37.
38.     SVD<double> svd;
39.     svd.dec(A);
40.
41.     Matrix<double> U = svd.getU();
42.     Matrix<double> V = svd.getV();
43.     Matrix<double> S = svd.getSM();
44.
45.     cout << "The original matrix A : " << A << endl;
46.     cout << "The left singular vectors U : " << U << endl;
47.     cout << "The singular values S : " << S << endl;
48.     cout << "The right singular vectors V : " << V << endl;
49.     Matrix<double> A1 = prod( U, prod(S,transpose(V)) );
50.     cout << "The norm of error matrix between A and U * S * V' : "
51.         << norm( A - A1 ) << endl << endl;
52.
53.     cout << "The rank of A : " << svd.rank() << endl << endl;
54.     cout << "The condition number of A : " << svd.cond() << endl << endl;
55.
56.     return 0;

```

```
57. }
```

运行结果:

```
1. The original matrix A : size: 2 by 4
2.  1 3 5 7
3.  2 4 6 8
4.
5. The left singular vectors U : size: 2 by 2
6.  0.641423 -0.767187
7.  0.767187 0.641423
8.
9. The singular values S : size: 2 by 2
10. 14.2691 0
11. 0 0.626828
12.
13. The right singular vectors V : size: 4 by 2
14. 0.152483 0.822647
15. 0.349918 0.421375
16. 0.547354 0.0201031
17. 0.744789 -0.381169
18.
19. The norm of error matrix between A and U * S * V' : 2.58946e-015
20.
21. The rank of A : 2
22.
23. The condition number of A : 22.764
24.
```

2.6 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 2.9。

表 2.9 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

测试代码:

```

1.  /*****
2.      *                               evd_test.cpp
3.      *
4.      * EVD class testing.
5.      *
6.      * Zhang Ming, 2010-01, Xi'an Jiaotong University.
7.      *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <evd.h>
14.
15.
16. using namespace std;
17. using namespace itlab;
18.
19.
20. const int N = 4;
21.
22.
23. int main()
24. {
25.     Matrix<double> A(N,N);
26.
27.     A[0][0] = 3.0; A[0][1] = -2.0; A[0][2] = -0.9;   A[0][3] = 0.0;
28.     A[1][0] = -2.0; A[1][1] = 4.0; A[1][2] = 1.0;   A[1][3] = 0.0;
29.     A[2][0] = 0.0; A[2][1] = 0.0; A[2][2] = -1.0;   A[2][3] = 0.0;
30.     A[3][0] = -0.5; A[3][1] = -0.5; A[3][2] = 0.1;   A[3][3] = 1.0;
31.
32.     EVD<double> eig;
33.     eig.dec(A);
34.
35.     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 0.417622 -2.35284e-016 0.152821
15. 0.788064 0.32607 -1.17642e-016 -0.134483
16. 1.06759e-017 4.59565e-017 -4.9436e-017 0.978057
17. -0.0189368 -0.848098 1 -0.0443182
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
29. 0
30.
31. The norm of error matrix between A * V and V * D : 3.82342e-015
32.

```


3 线性方程组

3.1 常规线性方程组求解

常规线性方程组有诸多解法，若系数矩阵为一般的满秩矩阵，则可以用列主元Gauss消元法或LU分解法进行求解；若系数矩阵为对称矩阵，则可以用Cholesky分解法进行求解；若系数矩阵为三对角矩阵，则可以用追赶法进行快速求解。矩阵方程其实是求解多个线性方程组，故解法一致，具体函数见表 3.1。

表 3.1 常规线性方程组求解方法

Operation	Effect
gaussSolver(A,B)	Gauss 消元法解矩阵方程
gaussSolver(A,b)	Gauss 消元法解线性方程组
luSolver(A,B)	LU 分解法解矩阵方程
luSolver(A,b)	LU 分解法解线性方程组
choleskySolver(A,B)	Cholesky 分解法解矩阵方程
choleskySolver(A,b)	Cholesky 分解法解线性方程组
febsSolver(A,b)	追赶法解三对角方程组

测试代码

```
1.  /*****
2.  *                               linearequations_test.cpp
3.  *
4.  * Deterministic Linear Equations testing.
5.  *
6.  * Zhang Ming, 2010-07, Xi'an Jiaotong University.
7.  *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <detlinequs.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.     Matrix<double> A(M,N), B(M,N);
27.     Vector<double> b(N);
28.
29.     // ordinary linear equations
30.     A[0][0] = 1;    A[0][1] = 2;    A[0][2] = 1;
31.     A[1][0] = 2;    A[1][1] = 5;    A[1][2] = 4;
32.     A[2][0] = 1;    A[2][1] = 1;    A[2][2] = 0;
33.     B = eye( 3, 1.0 );
34.     b[0] = 1;    b[1] = 0;    b[2] = 1;
35.     Matrix<double> invA( A );
36.     Matrix<double> X( B );
37.     cout << "The original matrix A is : " << A << endl;
38.     gaussSolver( invA, X );
39.     cout << "The inverse of A is ( Gauss Solver ) : " << invA << endl;
40.     cout << "The invse matrix of A is ( LUD Solver ) : " << luSolver( A, B ) << endl;
41.     cout << "The constant vector b : " << b << endl;
42.     cout << "The solution of A * x = b is ( Gauss Solver ) : " << gaussSolver( A, b ) <<
        endl;
43.     cout << "The solution of A * x = b is ( LUD Solver ) : " << luSolver( A, b ) << endl
        << endl;
44.
45.     // linear equations with symmetric coefficient matrix
46.     for( int i=1; i<N+1; ++i )
47.     {
48.         for( int j=1; j<N+1; ++j )
49.             if( i == j )
50.                 A(i,i) = i;
51.             else
52.                 if( i < j )
53.                     A(i,j) = i;
54.                 else
55.                     A(i,j) = j;
56.             b(i) = i*(i+1)/2 + i*(N-i);
57.         }
58.     cout << "The original matrix A : " << A << endl;

```

```

59.     cout << "The invse matrix of A is ( Cholesky Solver ) : " << choleskySolver( A, B )
        << endl;
60.     cout << "The constant vector b : " << b << endl;
61.     cout << "The solution of Ax = b is ( Cholesky Solver ) : " << choleskySolver( A, b )
        << endl << endl;
62.
63.     // Tridiagonal linear equations
64.     Vector<double> aa( 3, -1 ),
65.                   bb( 4, 4 ),
66.                   cc( 3, -2 ),
67.                   dd( 4 );
68.     dd(1) = 3;   dd(2) = 2;   dd(3) = 2;   dd(4) = 3;
69.     cout << "The elements below main diagonal is : " << aa << endl;
70.     cout << "The elements on main diagonal is : " << bb << endl;
71.     cout << "The elements above main diagonal is : " << cc << endl;
72.     cout << "The elements constant vector is : " << dd << endl;
73.     cout << "Teh solution is (Forward Elimination and Backward Substitution) : "
74.           << febsSolver( aa, bb, cc, dd ) << endl << endl;
75.
76.     return 0;
77. }

```

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 inverse of A is ( Gauss Solver ) : size: 3 by 3
7.  -4 1 3
8.  4 -1 -2
9.  -3 1 1
10.
11. The invse matrix of A is ( LUD Solver ) : size: 3 by 3
12. -4 1 3
13. 4 -1 -2
14. -3 1 1
15.
16. The constant vector b : size: 3 by 1
17. 1
18. 0
19. 1
20.
21. The solution of A * x = b is ( Gauss Solver ) : size: 3 by 1

```

线性方程组

```
22. -1
23. 2
24. -2
25.
26. The solution of A * x = b is ( LUD Solver ) : size: 3 by 1
27. -1
28. 2
29. -2
30.
31.
32. The original matrix A : size: 3 by 3
33. 1 1 1
34. 1 2 2
35. 1 2 3
36.
37. The invse matrix of A is ( Cholesky Solver ) : size: 3 by 3
38. 2 -1 0
39. -1 2 -1
40. 0 -1 1
41.
42. The constant vector b : size: 3 by 1
43. 3
44. 5
45. 6
46.
47. The solution of Ax = b is ( Cholesky Solver ) : size: 3 by 1
48. 1
49. 1
50. 1
51.
52.
53. The elements below main diagonal is : size: 3 by 1
54. -1
55. -1
56. -1
57.
58. The elements on main diagonal is : size: 4 by 1
59. 4
60. 4
61. 4
62. 4
63.
64. The elements above main diagonal is : size: 3 by 1
65. -2
```

```

66. -2
67. -2
68.
69. The elements constant vector is : size: 4 by 1
70. 3
71. 2
72. 2
73. 3
74.
75. The solution is (Forward Elimination and Backward Substitution) : size: 4 by 1
76. 1.56098
77. 1.62195
78. 1.46341
79. 1.11585
80.

```

3.2 超定与欠定线性方程组求解

若系数矩阵的行数大于列数，则称该方程组为超定线性方程组，这样的方程组不存在精确解，但可以求得与常数向量之间误差能量最小的解，即最小二乘解，主要有最小二乘广义逆矩阵法求解QR分解法求解。若系数矩阵的行数小于列数，则称该方程组为欠定线性方程组，这样的方程组存在无穷多组解，但是可以找到这些解中能量最小的一个，即最小范数解，主要有最小范数广义逆矩阵法求解QR分解法求解。超定与欠定线性方程组的具体函数调用形式详见表 3.2。

表 3.2 超定与欠定线性方程组求解方法

Operation	Effect
lsSolver(A,b)	最小二乘广义逆求解超定线性方程组
qrIsSolver(A,b)	QR 分解法求解超定线性方程组
lnSolver(A,b)	最小范数广义逆求解欠定线性方程组
qrlnSolver(A,b)	QR 分解法求解欠定线性方程组

测试代码

```

1.  /*****
2.  *                               linearequations_test.cpp
3.  *
4.  * Undetermined Linear Equations testing.
5.  *
6.  * Zhang Ming, 2010-07, Xi'an Jiaotong University.
7.  *****/
8.

```

```

9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <undetlinequs.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.     Matrix<double> A(M,N), B(M,N);
27.     Vector<double> b(N);
28.
29.     // overdetermined linear equations
30.     A.resize( 4, 3 );
31.     A[0][0] = 1;   A[0][1] = -1;   A[0][2] = 1;
32.     A[1][0] = 1;   A[1][1] = 2;    A[1][2] = 4;
33.     A[2][0] = 1;   A[2][1] = 3;    A[2][2] = 9;
34.     A[3][0] = 1;   A[3][1] = -4;   A[3][2] = 16;
35.     b.resize( 4 );
36.     b[0]= 1;    b[1] = 2;    b[2] = 3;    b[3] = 4;
37.     cout << "The original matrix A : " << A << endl;
38.     cout << "The constant vector b : " << b << endl;
39.     cout << "The least square solution is( using generalized inverse ) : " << lsSolver(
        A, b ) << endl;
40.     cout << "The least square solution is( using QR decomposition ) : " << qrlsSolver( A
        , b ) << endl;
41.
42.     // undetermined linear equations
43.     Matrix<double> At( transpose( A ) );
44.     b.resize( 3 );
45.     b[0]= 1;    b[1] = 2;    b[2]= 3;
46.     cout << "The original matrix A : " << At << endl;
47.     cout << "The constant vector b : " << b << endl;
48.     cout << "The least norm solution is( using generalized inverse ) : " << lnSolver( At
        , b ) << endl;

```



```

49.     cout << "The least norm solution is( using QR decomposition ) : " << qrlnSolver( At,
        b ) << endl;
50.
51.     return 0;
52. }

```

运行结果

```

1.  The original matrix A : size: 4 by 3
2.  1 -1 1
3.  1 2 4
4.  1 3 9
5.  1 -4 16
6.
7.  The constant vector b : size: 4 by 1
8.  1
9.  2
10. 3
11. 4
12.
13. The least square solution is( using generalized inverse ) : size: 3 by 1
14. 0.909091
15. 0.0787879
16. 0.212121
17.
18. The least square solution is( using QR decomposition ) : size: 3 by 1
19. 0.909091
20. 0.0787879
21. 0.212121
22.
23. The original matrix A : size: 3 by 4
24. 1 1 1 1
25. -1 2 3 -4
26. 1 4 9 16
27.
28. The constant vector b : size: 3 by 1
29. 1
30. 2
31. 3
32.
33. The least norm solution is( using generalized inverse ) : size: 4 by 1
34. 0.372727
35. 0.421212
36. 0.336364
37. -0.130303

```

```
38.
39. The least norm solution is( using QR decomposition ) : size: 4 by 1
40. 0.372727
41. 0.421212
42. 0.336364
43. -0.130303
44.
```

3.3 病态线性方程组求解

如果系数矩阵是秩亏矩阵，则称该方程组是病态方程组，这类方程组是不稳定的，故不能用常规方法求解。为了提解的高稳定性，必须牺牲解的准确性，学用的解法有截断SVD法，阻尼SVD法和Tikhonov正则化方法，其中截断SVD法求解的结果与Matlab中pinv(A)*b结果一致。具体函数调用格式见表 3.3。

表 3.3 病态线性方法组求解方法

Operation	Effect
tsvd(A,b,tol)	截断 SVD 法解秩亏线性方程组
dsvd(A,b,sigma)	阻尼 SVD 法解秩亏线性方程组
tikhonov(A,b,alpha)	Tikhonov 正则化法解秩亏线性方程组

测试代码

```
1.  /*****
2.  *                               linearequations_test.cpp
3.  *
4.  * Rank Defect Linear Equations testing.
5.  *
6.  * Zhang Ming, 2010-07, Xi'an Jiaotong University.
7.  *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <rankdeflinequs.h>
14.
15.
16. using namespace std;
17. using namespace itlab;
18.
19.
```

```

20. int main()
21. {
22.     Matrix<double> A(8,6);
23.     A[0][0]=64; A[0][1]=2; A[0][2]=3; A[0][3]=61; A[0][4]=60; A[0][5]=6;
24.     A[1][0]=9; A[1][1]=55; A[1][2]=54; A[1][3]=12; A[1][4]=13; A[1][5]=51;
25.     A[2][0]=17; A[2][1]=47; A[2][2]=46; A[2][3]=20; A[2][4]=21; A[2][5]=43;
26.     A[3][0]=40; A[3][1]=26; A[3][2]=27; A[3][3]=37; A[3][4]=36; A[3][5]=30;
27.     A[4][0]=32; A[4][1]=34; A[4][2]=35; A[4][3]=29; A[4][4]=28; A[4][5]=38;
28.     A[5][0]=41; A[5][1]=23; A[5][2]=22; A[5][3]=44; A[5][4]=45; A[5][5]=19;
29.     A[6][0]=49; A[6][1]=15; A[6][2]=14; A[6][3]=52; A[6][4]=53; A[6][5]=11;
30.     A[7][0]=8; A[7][1]=58; A[7][2]=59; A[7][3]=5; A[7][4]=4; A[7][5]=62;
31.
32.     Vector<double> b(8);
33.     b[0]=260; b[1]=260; b[2]=260; b[3]=260; b[4]=260; b[5]=260; b[6]=260; b[7]=260;
34.
35.     double alpha = 1.0e-006;
36.     double sigma = 1.0e-012;
37.
38.     cout << "The original matrix A : " << A << endl;
39.     cout << "The constant vector b : " << b << endl << endl;
40.     cout << "The solution of Truncated SVD ( is consistent with Matlab ) : "
41.         << endl << tsvd( A, b ) << endl;
42.     cout << "The solution of Damped SVD with alpha = "
43.         << sigma << " : " << endl << dsvd( A, b, sigma ) << endl;
44.     cout << "The solution of Tikhonov Regularization with alpha = "
45.         << alpha << " : " << endl << tikhonov( A, b, alpha ) << endl;
46.
47.     return 0;
48. }

```

运行结果

```

1. The original matrix A : size: 8 by 6
2. 64 2 3 61 60 6
3. 9 55 54 12 13 51
4. 17 47 46 20 21 43
5. 40 26 27 37 36 30
6. 32 34 35 29 28 38
7. 41 23 22 44 45 19
8. 49 15 14 52 53 11
9. 8 58 59 5 4 62
10.
11. The constant vector b : size: 8 by 1
12. 260
13. 260

```

线性方程组

```
14. 260
15. 260
16. 260
17. 260
18. 260
19. 260
20.
21.
22. The solution of Truncated SVD ( is consistent with Matlab ) :
23. size: 6 by 1
24. 1.15385
25. 1.46154
26. 1.38462
27. 1.38462
28. 1.46154
29. 1.15385
30.
31. The solution of Dampded SVD with alpha = 1e-012 :
32. size: 6 by 1
33. 1.1704
34. 1.46205
35. 1.42095
36. 1.42946
37. 1.40014
38. 1.117
39.
40. The solution of Tikhonov Regularization with alpha = 1e-006 :
41. size: 6 by 1
42. 1.15385
43. 1.46154
44. 1.38462
45. 1.38462
46. 1.46154
47. 1.15385
48.
```

4 插值与拟合

4.1 Newton 插值

对于给定的 $N+1$ 个点, 可以通过 Newton 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低阶的插值多项式, 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 4.1, 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 4.1 Newton 插值法

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

测试代码:

```
1.  /*****
2.   *                               newtoninterp_test.cpp
3.   *
4.   * Newton interpolation testing.
5.   *
6.   * Zhang Ming, 2010-04, Xi'an Jiaotong University.
7.   *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <constants.h>
14. #include <newtoninterp.h>
15.
16.
17. using namespace std;
```

```

18. using namespace itlab;
19.
20.
21. int main()
22. {
23.     Vector<double> x(5),
24.                 y(5);
25.     x[0] = 0; x[1] = 30; x[2] = 45;      x[3] = 60;      x[4] = 90;
26.     y[0] = 0; y[1] = 0.5; y[2] = sqrt(2)/2; y[3] = sqrt(3)/2; y[4] = 1;
27.
28.     NewtonInterp<double> poly(x,y);
29.     poly.calcCoefs();
30.
31.     cout << "Coefficients of Newton interpolated polynomial:"
32.           << poly.getCoefs() << endl;
33.
34.     cout << "The true and interpolated values:" << endl;
35.     cout << sin(10*D2R) << " " << poly.evaluate(10) << endl
36.           << sin(20*D2R) << " " << poly.evaluate(20) << endl
37.           << sin(50*D2R) << " " << poly.evaluate(50) << endl << endl;
38.
39.     return 0;
40. }

```

运行结果:

```

1. Coefficients of Newton interpolated polynomial:size: 5 by 1
2. 0
3. 0.0166667
4. -6.35455e-005
5. -7.25655e-007
6. 2.67214e-009
7.
8. The true and interpolated values:
9. 0.173648 0.173361
10. 0.34202 0.34188
11. 0.766044 0.766026
12.

```

4.2 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 4.2。

表 4.2 三次样条插值法

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

测试代码：

```

1.  /*****
2.      *                               spline3interp_test.cpp
3.      *
4.      * Spline3 interpolation testing.
5.      *
6.      * Zhang Ming, 2010-04, Xi'an Jiaotong University.
7.      *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <constants.h>
14. #include <spline3interp.h>
15.
16.
17. using namespace std;
18. using namespace itlab;
19.
20.
21. int main()
22. {
23.     // f(x) = 1 / (1+25*x^2)   -1 <= x <= 1
24.     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 };
25.     float yi[11] = { 0.0385, 0.0588, 0.1,  0.2,  0.5,  1.0, 0.5, 0.2, 0.1, 0.0588, 0.038
        5 };
26.     float Ml = 0.2105, Mr = 0.2105;
27.     Vector<float> x( 11, xi ), y( 11, yi );
28.
29.     Spline3Interp<float> poly( x, y, Ml, Mr );

```

```

30.     poly.calcCoefs();
31.
32.     cout << "Coefficients of cubic spline interpolated polynomial:" << endl
33.         << poly.getCoefs() << 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
18.

```

4.3 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 4.3。

表 4.3 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

测试代码:

```

1.  /*****
2.      *                               lsfit_test.cpp
3.      *
4.      * Least square fitting testing.
5.      *
6.      * Zhang Ming, 2010-04, Xi'an Jiaotong University.
7.      *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <lsfitting.h>
14. #include <utilities.h>
15.
16.
17. using namespace std;
18. using namespace itlab;
19.
20.
21. int main()
22. {
23.     double A = 4.0,
24.            alpha = 1.0,
25.            beta = -2.0,
26.            gamma = -4.0,
27.            tmp = 0.0;
28.
29.     int M = 100;
30.     Vector<double> x = linspace( 0.01, 0.5*PI, M );
31.     Vector<double> y(M);
32.     for( int i=0; i<M; ++i )
33.     {

```

```

34.         tmp = A * pow(x[i],alpha) * exp(beta*x[i]+gamma*x[i]*x[i]);
35.         y[i] = log(max(tmp,EPS));
36.     }
37.
38.     Funcs<double> phi;
39.     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
13.

```

5 优化算法

5.1 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 5.1。

表 5.1 一维线搜索

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

5.2 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`SteepestDesc<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 5.2。

表 5.2 最速下降法

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

测试代码：

```

1.  /*****
2.      *                      steepdesc_test.cpp
3.      *
4.      * Steepest descent method testing.
5.      *
6.      * Zhang Ming, 2010-03, Xi'an Jiaotong University.
7.      *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <function.h>
14. #include <steepdesc.h>
15.
16.
17. using namespace std;
18. using namespace itlab;
19.
20.
21. typedef    float  Type;
22.
23.
24. int main()
25. {
26.     Type a = 1.0,
27.         b = -1.0,
28.         c = -1.0;
29.     ObjFunc<Type> f( a, b, c );
30.     Vector<Type> x0(2);
31.     x0(1) = Type(0.0);
32.     x0(2) = Type(0.0);
33.
34.     Type tolErr = 1.0e-3;
35.     SteepDesc< Type, ObjFunc<Type> > steep;
36.     steep.optimize( f, x0, tolErr );
37.     if( steep.isSuccess() )
38.     {
39.         Vector<Type> xmin = steep.getOptValue();
40.         int N = steep.getItrNum();
41.         cout << "The iterative number is:  " << N << endl << endl;
42.         cout << "The number of function calculation is:  " << steep.getFuncNum() << endl
43.         1 << 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:  " << 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
12.
```

5.3 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 5.3，其中默认的再开始次数为目标函数的维数 n 。

表 5.3 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, maxIter)	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

测试代码:

```

1.  /*****
2.      *                               conjgrad_test.cpp
3.      *
4.      * ConjGrad descent method testing.
5.      *
6.      * Zhang Ming, 2010-03, Xi'an Jiaotong University.
7.      *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <function.h>
14. #include <conjgrad.h>
15.
16.
17. using namespace std;
18. using namespace itlab;
19.
20.
21. typedef    double  Type;
22.
23.
24. int main()
25. {
26.     Type a = 1.0,
27.         b = -1.0,
28.         c = -1.0;
29.     ObjFunc<Type> f( a, b, c );
30.     Vector<Type> x0(2);
31.     x0(1) = 0.5;
32.     x0(2) = 0.5;
33.
34.     Type tolErr = 1.0e-6;
35.     ConjGrad< Type, ObjFunc<Type> > prp;
36.     prp.optimize( f, x0, x0.dim(), tolErr );
37.     if( prp.isSuccess() )
38.     {

```

```

39.         Vector<Type> xmin = prp.getOptValue();
40.         int N = prp.getItrNum();
41.         cout << "The iterative number is:  " << N << endl << endl;
42.         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 can'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
12.

```

5.4 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。比较常用的两类似 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 5.4。

表 5.4 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

测试代码:

```

1.  /*****
2.   *                                     bfgs_test.cpp
3.   *
4.   * BFGS method testing.
5.   *
6.   * Zhang Ming, 2010-03, Xi'an Jiaotong University.
7.   *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <function.h>
14. #include <bfgs.h>
15.
16.
17. using namespace std;
18. using namespace itlab;
19.
20.
21. typedef    double  Type;
22.
23.
24. int main()
25. {
26.     Type a = 1.0,
27.         b = -1.0,
28.         c = -1.0;
29.     ObjFunc<Type> f( a, b, c );
30.     Vector<Type> x0(2);
31.     x0(1) = Type(0.0);
32.     x0(2) = Type(0.0);
33.

```



```

34.     BFGS< Type, ObjFunc<Type> > bfgs;
35.     bfgs.optimize( f, x0 );
36.     if( bfgs.isSuccess() )
37.     {
38.         Vector<Type> xmin = bfgs.getOptValue();
39.         int N = bfgs.getItrNum();
40.         cout << "The iterative number is:  " << N << endl << endl;
41.         cout << "The number of function calculation is:  " << bfgs.getFuncNum() << endl
42.         << 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:  " << bfgs.getGradNorm()[N] << endl << en
46.         dl;
47.     }
48.     else
49.         cout << "The optimal solution  can't be found!" << endl;
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
12.

```


6 Fourier 分析

6.1 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 6.1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 6.1 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：

```
1.  /*****
2.   *                                     fft_test.cpp
3.   *
4.   * FFT interface testing.
5.   *
6.   * Zhang Ming, 2010-01, Xi'an Jiaotong University.
7.   *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <complex>
14. #include <fft.h>
15. #include <constants.h>
16.
17.
18. 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:

```

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)
16. (-3.25876e-008,9.92628e-008)
17. (4.93861e-008,1.54048e-007)
18. (-1.20608e-007,2.32841e-008)
19.
20.
21. yn=ifft(Xk):   size: 7 by 1
22. (1,1.77636e-015)
23. (0.62349,0.781831)
24. (-0.222521,0.974928)
25. (-0.900969,0.433884)
26. (-0.900969,-0.433884)
27. (-0.222521,-0.974928)
28. (0.62349,-0.781832)
29.
30.
31. sn:   size: 7 by 1
32. 0
33. 0.781832
34. 0.974928
35. 0.433884
36. -0.433884
37. -0.974928
38. -0.781832
39.
40. Sk=fft(sn):   size: 4 by 1
41. (-5.96046e-008,0)
42. (-1.28028e-007,-3.5)
43. (6.36894e-008,-8.22043e-008)

```

```
44. (9.41406e-008,-1.08758e-007)
45.
46.
47. sn=ifft(Sk):   size: 7 by 1
48. -2.03012e-015
49. 0.781832
50. 0.974928
51. 0.433884
52. -0.433884
53. -0.974928
54. -0.781832
55.
```

6.2 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 6.2。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 6.2 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：

```
1.  /*****
2.  *                                     ffr2_test.cpp
3.  *
4.  * Radix 2 based FFT testing.
5.  *
6.  * Zhang Ming, 2010-04, Xi'an Jiaotong University.
7.  *****/
8.
9.
```

```

10. #include <iostream>
11. #include <iomanip>
12. #include <fftr2.h>
13.
14.
15. using namespace std;
16. using namespace itlab;
17.
18.
19. int LENGTH = 32;
20.
21.
22. int main()
23. {
24.     int    i, j, index, rows = LENGTH/4;
25.
26.     Vector<double> xn(LENGTH);
27.     Vector< complex<double> > yn(LENGTH),
28.                               Xk(LENGTH);
29.     FFTR2<double> Fourier;
30.
31.     cout << "The original signal is: " << endl;
32.     for( i=0; i<rows; i++ )
33.     {
34.         cout << endl;
35.         for( j=0; j<3; j++ )
36.         {
37.             index = 3*i+j;
38.             xn[index] = i+j;
39.             cout << setiosflags(ios::fixed) << setprecision(6);
40.             cout << "\t" << xn[index];
41.         }
42.     }
43.     cout << endl << endl;
44.
45.     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 transform 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 transform 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.  The original signal is:
2.
3.      0.000000      1.000000      2.000000
4.      1.000000      2.000000      3.000000
5.      2.000000      3.000000      4.000000
6.      3.000000      4.000000      5.000000
7.      4.000000      5.000000      6.000000
8.      5.000000      6.000000      7.000000
9.      6.000000      7.000000      8.000000
10.     7.000000      8.000000      9.000000
11.
12. The Fourier transform of original signal is:
13.
14.     (108.000000,0.000000)  (-52.827446,1.358153)  (-0.613126,-23.640092)
15.     (13.310056,1.647944)  (-4.000000,9.656854)  (-8.902567,-4.353284)
16.     (3.082392,-7.681941)  (5.303682,2.676653)  (-4.000000,4.000000)
17.     (-5.314558,-4.659704)  (0.917608,-8.368233)  (0.220781,11.701395)
18.     (-4.000000,1.656854)  (-0.343999,-3.954231)  (4.613126,-0.326383)
19.     (0.554051,4.364941)  (-4.000000,-0.000000)  (0.554051,-4.364941)
20.     (4.613126,0.326383)  (-0.343999,3.954231)  (-4.000000,-1.656854)
21.     (0.220781,-11.701395)  (0.917608,8.368233)  (-5.314558,4.659704)
22.

```

23.	The inverse Fourier transform is		
24.			
25.	0.000000	1.000000	2.000000
26.	1.000000	2.000000	3.000000
27.	2.000000	3.000000	4.000000
28.	3.000000	4.000000	5.000000
29.	4.000000	5.000000	6.000000
30.	5.000000	6.000000	7.000000
31.	6.000000	7.000000	8.000000
32.	7.000000	8.000000	9.000000
33.			
34.	The original signal is:		
35.			
36.	(0.000000,0.000000)	(0.000000,1.000000)	(0.000000,2.000000)
37.	(1.000000,0.000000)	(1.000000,1.000000)	(1.000000,2.000000)
38.	(2.000000,0.000000)	(2.000000,1.000000)	(2.000000,2.000000)
39.	(3.000000,0.000000)	(3.000000,1.000000)	(3.000000,2.000000)
40.	(4.000000,0.000000)	(4.000000,1.000000)	(4.000000,2.000000)
41.	(5.000000,0.000000)	(5.000000,1.000000)	(5.000000,2.000000)
42.	(6.000000,0.000000)	(6.000000,1.000000)	(6.000000,2.000000)
43.	(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 transform 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)
66.			

6.3 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 $N\log_2N$ ，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 6.3。

表 6.3 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：

```
1.  /*****
2.      *                               fastconv_test.cpp
3.      *
4.      * Fast convolution testing.
5.      *
6.      * Zhang Ming, 2010-01, Xi'an Jiaotong University.
7.      *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <fastconv.h>
14.
15.
16. using namespace std;
17. using namespace itlab;
18.
19.
20. const int M = 3;
21. const int N = 5;
22. typedef double Type;
23.
24.
25. int main()
26. {
27.     Vector<Type> xn( M ),
```

```

28.         yn( N );
29.     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.
7.  yn:  size: 5 by 1
8.  -2
9.  -1
10. 0
11. 1
12. 2
13.
14.
15. convolution:  size: 7 by 1
16. 8.72318e-017
17. -2
18. -5
19. -2
20. 1

```

```
21. 4
22. 4
23.
24.
25. auto-correlation: size: 5 by 1
26. -4.44089e-017
27. 2
28. 5
29. 2
30. -5.67255e-017
31.
32.
33. cross-correlation: size: 7 by 1
34. -4
35. -4
36. -1
37. 2
38. 5
39. 2
40. 7.56959e-016
41.
```


7 数字滤波器设计

7.1 常用窗函数

表 7.1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 7.1 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, Xi'an Jiaotong University.
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.
```

```

20. int main()
21. {
22.     int N = 5;
23.
24.     cout << "Rectangle window : " << rectangle(N) << endl << endl;
25.     cout << "Bartlett window : " << bartlett(N) << endl << endl;
26.     cout << "Hanning window : " << hanning(N) << endl << endl;
27.     cout << "Hamming window : " << hamming(N) << endl << endl;
28.     cout << "Blackman window : " << blackman(N) << endl << endl;
29.     cout << "Kaiser window : " << kaiser(8/PI,N) << endl << endl;
30.     cout << "Gauss window : " << gauss(N) << endl;
31.
32.     return 0;
33. }

```

运行结果:

```

1.  Rectangle window : size: 5 by 1
2.  1
3.  1
4.  1
5.  1
6.  1
7.
8.
9.  Bartlett window : size: 5 by 1
10. 0
11. 0.5
12. 1
13. 0.5
14. 0
15.
16.
17. Hanning window : size: 5 by 1
18. 0
19. 0.5
20. 1
21. 0.5
22. 0
23.
24.
25. Hamming window : size: 5 by 1
26. 0.08
27. 0.54
28. 1

```



```
29. 0.54
30. 0.08
31.
32.
33. Blackman window : size: 5 by 1
34. -1.38778e-017
35. 0.34
36. 1
37. 0.34
38. -1.38778e-017
39.
40.
41. Kaiser window : size: 2 by 1
42. 0.0367109
43. 0.0367109
44.
45.
46. Gauss window : size: 5 by 1
47. 0.0439369
48. 0.457833
49. 1
50. 0.457833
51. 0.0439369
52.
```

7.2 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 7.2所示。

表 7.2 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

7.3 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 7.3 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：

```
1.  /*****
2.  *                               fir_test.cpp
3.  *
4.  *  FIR class testing.
5.  *
6.  *  Zhang Ming, 2010-03, Xi'an Jiaotong University.
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.
24.     // string fType = "lowpass";
25.     // double fs = 1000,
26.     //         fpass = 200,
27.     //         apass = -3,
28.     //         fstop = 300,
29.     //         astop = -20;
```

```

30. //    FIR fir( fType, wType );
31. //    fir.setParams( fs, fpass, apass, fstop, astop );
32.
33. //    string fType = "highpass";
34. //    double fs = 1000,
35. //           fstop = 200,
36. //           astop = -20,
37. //           fpass = 300,
38. //           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：

```
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)
27.
```

7.4 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 7.4。

表 7.4 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:

```
1.  /*****
2.      *                               iir_test.cpp
3.      *
4.      * IIR class testing.
5.      *
6.      * Zhang Ming, 2010-03, Xi'an Jiaotong University.
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
8.	Lower stopband frequency (Hz)	:	200
9.	Upper stopband frequency (Hz)	:	300
10.	Stopband gain (dB)	:	-20
11.	Upper passband frequency (Hz)	:	400
12.	Upper passband gain (dB)	:	-3
13.			
14.			
15.	Numerator Coefficients		Denominator Coefficients
16.			
17.	N	[1 z ⁻¹ z ⁻²]	[1 z ⁻¹ z ⁻²]
18.	===	=====	=====
19.	1	1.0 -0.45087426 1.00000000	1.0 -1.44482415 0.70572930
20.	2	1.0 0.45087426 1.00000000	1.0 1.44482415 0.70572930
21.			
22.			
23.	===== Edge Frequency Response =====		
24.	Mag(fp1) =	-3.00000000(dB)	Mag(fp2) = -3.00000000(dB)
25.	Mag(fs1) =	-36.87418446(dB)	Mag(fs2) = -36.87418446(dB)
26.			

8 自适应滤波器

8.1 Wiener 滤波器

8.2 Kalman 滤波器

8.3 LMS 自适应滤波器

9 时频分析

9.1 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 9.1。

表 9.1 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：

```
1.  /*****
2.      *
3.      *
4.      * Windowed Fourier transform testing.
5.      *
6.      * Zhang Ming, 2010-03, Xi'an Jiaotong University.
7.      *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <wft.h>
14. #include <mathfunc.h>
15. #include <timing.h>
16.
17.
18. using namespace std;
19. 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 = 0.013 (ms)
3.
4.  Taking inverse windowed Fourier transform.
5.  The running time = 0.013 (ms)
6.
7.  The relative error is : norm(s-x) / norm(s) = 4.43299e-008
8.
```

9.2 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 9.2。

表 9.2 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, Xi'an Jiaotong University.
7.      *****/
8.
```

```

9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <dgt.h>
14. #include <mathfunc.h>
15. #include <timing.h>
16.
17.
18. using namespace std;
19. using namespace itlab;
20.
21.
22. const int Fs = 1000;
23. const int Ls = 1000;
24. const int Lg = 80;
25. const int N = 40;
26. const int dM = 10; // over sampling ratio is N/dM
27. typedef double Type;
28.
29.
30. int main()
31. {
32.
33. //***** [ signal ] *****/
34. Type a = 0;
35. Type b = Ls-1;
36. Vector<Type> t = linspace( a, b, Ls ) / Type(Fs);
37. Vector<Type> st = vcos( Type(400*PI) * vpow(t,Type(2.0)) );
38.
39. //***** [ widow ] *****/
40. a = 0;
41. b = Type( Lg-1 );
42. Type r = sqrt( dM*N / Type(2*PI) );
43. Type u = (Lg-1) / Type(2.0);
44. t = linspace( a, b, Lg );
45. Vector<Type> h = gauss( t, u, r );
46. h = h / norm(h);
47.
48. //***** [ daul function ] *****/
49. Type runtime = 0.0;
50. Timing cnt;
51. cout << "Compute daul function." << endl;
52. cnt.start();

```

```

53.   Vector<Type> g = daul( h, N, dM );
54.   cnt.stop();
55.   runtime = cnt.read();
56.   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 (ms)
3.
4.   Taking discrete Gabor transform.
5.   The running time = 0.008 (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) = 3.80504e-013
11.

```

9.3 Wigner-Wille 分布

10 小波变换

10.1 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 10.1。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 10.1 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, Xi'an Jiaotong University.
7.  *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <cwt.h>
14. #include <mathfunc.h>
```

```

15. #include <utilities.h>
16. #include <statistics.h>
17. #include <timing.h>
18.
19.
20. using namespace std;
21. using namespace itlab;
22.
23.
24. const int Ls = 1000;
25. const double fs = 1000.0;
26.
27.
28. int main()
29. {
30.
31.     /***** [ signal ] *****/
32.     Vector<double> t = linspace( 0.0, (Ls-1)/fs, Ls );
33.     Vector<double> st = vsin( 200*PI*vpow(t,2.0) );
34.     st = st-mean(st);
35.
36.     /***** [ CWT ] *****/
37.     Matrix< complex<double> > coefs;
38.     CWT<double> wavelet("morlet");
39.     wavelet.setScales( fs, fs/Ls, fs/2 );
40.     Timing cnt;
41.     double runtime = 0.0;
42.     cout << "Taking contineous wavelet transform(Morlet)." << endl;
43.     cnt.start();
44.     coefs = wavelet.cwtC(st);
45.     cnt.stop();
46.     runtime = cnt.read();
47.     cout << "The running time = " << runtime << " (ms)" << endl << endl;
48.
49.     /***** [ ICWT ] *****/
50.     cout << "Taking inverse contineous wavelet transform." << endl;
51.     cnt.start();
52.     Vector<double> xt = wavelet.icwtC(coefs);
53.     cnt.stop();
54.     runtime = cnt.read();
55.     cout << "The running time = " << runtime << " (ms)" << endl << endl;
56.
57.     cout << "The relative error is : " << endl;
58.     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> coefs = 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(coefs);
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 = 0.026 (ms)
3.
4.  Taking inverse contineous wavelet transform.
5.  The running time = 0.005 (ms)
6.
7.  The relative error is :
8.  norm(st-xt) / norm(st) = 0.00604041
9.
10.

```

```
11. Taking contineous wavelet transform(Mexican Hat).
12. The running time = 0.042 (ms)
13.
14. Taking inverse contineous wavelet transform.
15. The running time = 0.002 (ms)
16.
17. The relative error is :
18. norm(st-xt) / norm(st) = 0.00156789
19.
```

10.2 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 10.2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 10.2 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：

```
1.  /*****
2.  *                               bwt_test.cpp
3.  *
4.  * Dyadic wavelet transform testing.
5.  *
6.  * Zhang Ming, 2010-03, Xi'an Jiaotong University.
7.  *****/
8.
9.
10. #define BOUNDS_CHECK
11.
12. #include <iostream>
13. #include <bwt.h>
14. #include <timing.h>
15.
16.
```

```

17. 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
8.
```

10.3 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 10.3。

表 10.3 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(coefs)	获取离散小波变换的逼近系数
wt.getApprox(coefs,j)	获取离散小波变换的 j 级细节系数
wt.setApprox(a,coefs)	设置离散小波变换的逼近系数
wt.setApprox(d,coefs,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, Xi'an Jiaotong University.
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
9.

```