积分器的研究与实现

李璐君

2017 年 10 月 8 日

目　录

1 高斯型积分公式的理论理解

1.1 为什么要计算数值积分

实际问题中常常需要计算积分，有些数值方法，如微分方程也都和积分计算想联系。
依据人们所熟知的微积分基本定理，对于积分$I=\int_a^b f(x)dx$ 只要找到被积函数$f(x)$ 的原函数$F(x)$,便有下列的牛顿-莱布尼茨公式： $\int_a^b f(x)dx=F(b)-F(a)$.
但实际使用这种积分方法往往有困难，大量的被积函数我们找不到其用初等函数表示的原函数；另外，当$\mho(x)$是由测量实验数据或者数值计算给出的一张数据表的时候，牛顿莱布尼茨公式也不能直接使用，而且随着计算机的普及和在科研领域的应用，迫切需要一种计算机可以使用的数值积分算法因此有必要研究积分的数值计算问题。

1.2 为什么高斯型积分可以逼近真实积分值

首先，积分中值定理告诉我们，在积分区间$[a,b]$内存在一点c,$\int_a^b f(x)dx=(b-a)f(c)$成立。也就是说，底为$(b-a)$高为$f(c)$的矩形面积恰等于所求曲边梯形的面积$I$，问题在于点$c$的位置一般是不知道的，因而难以准确算出$f(c)$的值。我们将$f(c)$称为区间$[a,b]$上的平均高度。这样，只要对平均高度$f(c)$提供一种算法，相应的便获得一种数值求积的方法。如果我们用两端点“高度”$f(a)$和$f(b)$的算术平均作为平均高度f(c)的近似值，这样导出的求积公式为$T=\frac{b-a}{2}[f(a)+f(b)]$ 这便是我们所熟悉的梯形公式。更一般的，我们可以在区间$[a,b]$上适当选取某些节点X_k，然后用$f(x_k)$加权平均得到平均高度$f(c)$的近似值，这样构造出的求积公式便会具有如下的形式： $\int_a^b f(x)dx\approx\sum_{k=0}^n A_kf(x_k)$
其中A_k的权只和x_k的选取有关，而不依赖于被积函数$f(x)$的具体形式。这类积分求值问题可以归结为函数值得计算，计算机可以实现且计算量较小不消耗计算资源，而且更好的避开了牛顿莱布尼茨公式的求原函数困难的问题。

1.3 为什么用高斯点去近似积分值

对于高斯求积公式，我们研究带权积分$I=\int_a^b f(x)\rho(x)dx$,这里的$\rho(x)$为权函数，它的求积公式为$\int_a^b f(x)\rho(x)dx\approx\sum_{k=0}^n A_kf(x_k)$,其中$x_k$为求积节点，定理告诉我们插值求积公式是稳定的求代数精度至少为n次，因此为了使得求积公式的代数精度达到最高代数精度 2n+1次，我们需要适当选择求积节点。而求积公式代数精度为2n+1的必要条件是求积节点为正交多项式的根。

2 代码介绍

- integrator.h (积分函数所在头文件)
- integrator.cpp (积分函数内容所在文件)
- main.cpp (主函数：包含六种积分函数)
- matrix.h (我自己编写的矩阵类的头文件)
- matrix.cpp (矩阵类以及类内函数的详细内容)由于时间原因，我的矩阵类在编写的时候已经进行了中文注释改进可能会很麻烦，望老师见谅)
- Parsing_Math_Expression (王炳辉的栈类和识别表达式的类用于精简我的程序)

3 积分方法简略介绍

积分程序中我大量运用我的矩阵类，其中我也用矩阵进行存储各种多项式的根，在此说明第一行为实部，第二行为虚部。

3.1 高斯勒让德求积方式

算法通过施密特正交化将勒让德多项式的酉矩阵正交化再利用QR算法求出勒让德多项式的根。

3.2 高斯拉盖尔求积方式

算法通过二分法求出拉盖尔多项式的根。

3.3 高斯埃尔米特求积方式

算法通过施密特正交化将勒让德多项式的酉矩阵正交化再利用QR算法求出埃尔米特多项式的根。

3.4 高斯切普雪夫求积方式

算法通过维基百科提供的递推公式求出切普雪夫多项式的根。

3.5 高斯蒙特卡洛求积方式

算法通过向平面中撒均匀点利用积分的几何意义求积分。其中求最大最小值运用暴力搜索的方式，并且函数值排序运用归并排序法，虽然这样排序做了一些无用功，但是利用自己已经编过的成熟正确的函数简化了编程难度。

3.6 黎曼和求积方式

算法通过简单函数-阶梯函数逼近的方式求矩形的面积来求积分。

4 积分程序源代码

```
#include <integrator.h>
#define pi 3.1415926


double  integrator_legend(MathFunc f,double lowerlimit,double upperlimit,unsigned n)
{
    double a = (upperlimit - lowerlimit)/2;
    double b = (upperlimit + lowerlimit)/2;
    double sum = 0;
    Matrix roots = find_root_legende(n);
    Matrix coeff = Matrix(1,n);
```

```
12
13      for(unsigned i = 0 ;i < n ;i++)
14      {
15          coeff.data[0][i] = 2.0/((1.0-pow(roots.data[0][i],2))*pow((value(
    ↪  derivation_polynomial( polynomial_legende(n),1),roots.data[0][i])),2));
16      }
17      for(unsigned i = 0 ;i < n;i++)
18      {
19          sum = sum + coeff.data[0][i]*f.eval(a*roots.data[0][i]+b);
20      }
21      return a*sum;
22  }
23
24  unsigned fac(unsigned n)
25  {
26      if(n<0) return 0;
27      if(n==0||n==1)return 1;
28      if(n>1)
29      {
30          return n*fac(n-1);
31      }
32  }
33
34  double integrator_laguerre(MathFunc f,double lowerlimit,double upperlimit,unsigned n)
35  {
36      double sum = 0;
37      double a = lowerlimit;
38      double b = upperlimit;
39      Matrix roots = find_root_laguerre(n);
40      Matrix coeff = Matrix(1,n);
41      for(unsigned i = 0 ;i < n ;i++)
42      {
43          coeff.data[0][i] =
    ↪  roots.data[0][i]/(pow(value(polynomial_laguerre(n+1),roots.data[0][i]),2)*pow(n+1,2));
44          // cout << coeff.data[0][i]<<endl;
45      }
46      cout << a << endl;
47      cout << b << endl;
48      if ((a != -DBL_MAX)&&( b == DBL_MAX))
49      {
50          for(unsigned i = 0 ;i < n;i++)
51          {
52              sum = sum + coeff.data[0][i]*f.eval(roots.data[0][i]+a)*exp(roots.data[0][i]);
53          }
54      }
55      else if((a == -DBL_MAX)&&( b != DBL_MAX))
56      {
57          for(unsigned i = 0 ;i < n;i++)
58          {
59              sum = sum + coeff.data[0][i]*f.eval(b-roots.data[0][i])*exp(roots.data[0][i]);
```

```
60          }
61      }
62      else if ((a == -DBL_MAX)&&( b == DBL_MAX))
63      {
64          for(unsigned i = 0 ;i < n;i++)
65          {
66              sum = sum + coeff.data[0][i] * f.eval(roots.data[0][i]) * exp(roots.data[0][i])
67                        + coeff.data[0][i] * f.eval(-roots.data[0][i]) * exp(roots.data[0][i]);
68          }
69      }
70      else if ((a == -DBL_MAX)&&( b ==-DBL_MAX))
71      {
72          return sum;
73      }
74      else if ((a == DBL_MAX)&&( b ==DBL_MAX))
75      {
76          return sum;
77      }
78      else
79      {
80          cout << "The gauss_laguerre method can only make the approximation of this kind of
    ↪  integration:upperlimit == infty or lower limit ==infty"<<endl;
81          exit(-1);
82      }
83      return sum;
84  }
85
86  double integrator_Hermite(MathFunc f,double lowerlimit,double upperlimit,unsigned n)
87  {
88      double a = lowerlimit;
89      double b = upperlimit;
90      double sum = 0;
91      Matrix roots = find_root_Hermite(n);
92      Matrix coeff = Matrix(1,n);
93
94      if((a == -DBL_MAX)&&(b == DBL_MAX))
95      {
96          for(unsigned i = 0 ;i < n ;i++)
97          {
98              coeff.data[0][i] =
    ↪  (pow(2,n-1)*fac(n)*sqrt(pi))/(pow(n,2))/pow(value(polynomial_Hermite(n-1),roots.data[0][i]),2);
99          }
100         for(unsigned i = 0 ;i < n;i++)
101         {
102             sum = sum +
    ↪  coeff.data[0][i]*f.eval(roots.data[0][i])*exp(roots.data[0][i]*roots.data[0][i]);
103         }
104     }
105     else
106     {
```

```
107         cout << "The integrator_Hermite can just make approximation of integration from -infty
    ↪  to infty !!"<<endl;
108         exit (-1);
109     }
110     return sum;
111 }
112 
113 double integrator_Chebyshev(MathFunc f,double lowerlimit,double upperlimit,unsigned n)
114 {
115     double a = (upperlimit - lowerlimit)/2;
116     double b = (upperlimit + lowerlimit)/2;
117     double sum = 0;
118     Matrix roots = find_root_Chebyshev(n);
119     Matrix coeff = Matrix(1,n);
120 
121     if((lowerlimit != -DBL_MAX)&&(lowerlimit!=DBL_MAX)&&(upperlimit != -DBL_MAX)&&(upperlimit
    ↪  != DBL_MAX))
122     {
123         for(unsigned i = 0 ;i < n ;i++)
124         {
125             coeff.data[0][i] = pi/n;
126         }
127         for(unsigned i = 0 ;i < n;i++)
128         {
129             sum = sum +
    ↪  coeff.data[0][i]*f.eval(a*roots.data[0][i]+b)*sqrt(1.0-roots.data[0][i]*roots.data[0][i]);
130         }
131     }
132     else
133     {
134         cout << "The integrator_Chebyshev can only make approximation for integration from a to
    ↪  b in which a and b isn't infty"<<endl;
135         exit(-1);
136     }
137     return a*sum;
138 }
139 
140 void Sort_2SortedArray(double* a, unsigned l, unsigned m, unsigned r)
141 {
142     unsigned i = l, j = m + 1;
143     int k = 0;
144     double * temp = (double*)malloc( (r-l+1) * sizeof(double) );
145 
146     while( (i <= m) && (j <= r) )
147     {
148         if(a[i] <= a[j])
149         {
150             temp[k] = a[i];
151             i++;
152             k++;
```

```
153             }
154             else
155             {
156                 temp[k] = a[j];
157                 j++;
158                 k++;
159             }
160         }
161         if(i<=m)
162         {
163             for(;i<=m;i++)
164             {
165                 temp[k]= a[i];
166                 k++;
167             }
168         }
169         if(j<=r)
170         {
171             for(;j<=r;j++)
172             {
173                 temp[k]= a[j];
174                 k++;
175 
176             }
177         }
178         for(unsigned m = l,i=0; m <= r; m++)
179         {
180             a[m] = temp[i];
181             k++;
182             i++;
183         }
184 
185     }
186 
187     void merge (double *a,int left,int right)
188     {
189         if(left==right)
190             return;
191         int middle =(right+left)/2;
192         merge(a,left,middle);
193         merge(a,middle+1,right);
194         Sort_2SortedArray(a,left,middle,right);
195     }
196 
197 
198     double violent_search_Max(MathFunc f, double lowerlimit,double upperlimit,unsigned long
        ↪  section)        //section的大小最少上万
199     {
200         double length = upperlimit - lowerlimit;
201         double step_length = length/section;
```

```
202     unsigned flag=0;
203     double * A = (double *)malloc((section+1)*sizeof(double));
204     double * B = (double *)malloc((section+1)*sizeof(double));
205     for(unsigned i = 0 ;i < section+1;i++)
206     {
207         A[i] = f.eval(lowerlimit+step_length*i);
208         B[i] =  A[i];
209     }
210     merge(A,0,section);
211     return A[section];
212 }
213
214 double violent_search_Min(MathFunc f, double lowerlimit,double upperlimit,unsigned long
    ↪  section)        //section的大小最少上万
215 {
216     double length = upperlimit - lowerlimit;
217     double step_length = length/section;
218
219     unsigned flag=0;
220     double * A = (double *)malloc((section+1)*sizeof(double));
221     double * B = (double *)malloc((section+1)*sizeof(double));
222     for(unsigned i = 0 ;i < section+1;i++)
223     {
224         A[i] = f.eval(lowerlimit+step_length*i);
225         B[i] =  A[i];
226     }
227     merge(A,0,section);
228
229     return A[0];
230 }
231
232 Matrix Scatter_function(double x_left,double x_right,double y_lower,double y_upper,unsigned
    ↪  long n) //本函数用于向平面中撒点
233 {
234         double x_length = x_right-x_left;
235         double y_length = y_upper-y_lower;
236
237         Matrix locations = Matrix(2,n);
238         {
239             srand(rand());
240             for(unsigned i = 1 ;i <= n; i++)
241             {
242                 locations.data[0][i-1]=(rand()*1.0/RAND_MAX)*x_length+x_left;
243             }
244
245             srand(rand());
246             for(unsigned i = 1 ;i <= n; i++)
247             {
248                 locations.data[1][i-1] =(rand()*1.0/RAND_MAX)*y_length+y_lower;
249             }
```

```
250             return locations;
251         }
252 
253 }
254 double integrator_Monte_Carlo(MathFunc f,double lowerlimit,double upperlimit,unsigned long n)
255 {
256 
257     double a = lowerlimit;
258     double b = upperlimit;
259     double length = b-a;
260     double min = violent_search_Min(f,a,b,100000);
261     double max = violent_search_Max(f,a,b,100000);
262     double value_length = max - min;
263     double area_length = max-min;
264     long number_p=0;
265     long number_n=0;
266 
267     if((lowerlimit != -DBL_MAX)&&(lowerlimit!=DBL_MAX)&&(upperlimit != -DBL_MAX)&&(upperlimit
    ↪  != DBL_MAX))
268     {
269         if(min*max >0)
270         {
271             if(min > 0)
272             {
273                 Matrix points = Scatter_function(a,b,min,max,n);
274                 for(unsigned i = 0 ; i < n ; i++)
275                 {
276                     if(points.data[1][i]<=f.eval(points.data[0][i]))
277                     {
278                         number_p ++;
279                     }
280                 }
281 
282                 return (length*value_length)*(number_p*1.0/n);
283             }
284             else
285             {
286                 Matrix points = Scatter_function(a,b,min,max,n);
287                 for(unsigned i = 0 ; i < n ; i++)
288                 {
289                     if(points.data[1][i]>=f.eval(points.data[0][i]))
290                     {
291                         number_n ++;
292                     }
293                 }
294                 return -(length*value_length)*(number_n*1.0/n);
295             }
296         }
297         else if(min*max <0)
298         {
```

```
299             Matrix points = Scatter_function(a,b,min,max,n);
300             for(unsigned i = 0 ; i < n ; i++)
301             {
302                 if(points.data[1][i]>=0)
303                 {
304                     if(points.data[1][i]<=f.eval(points.data[0][i]))
305                     {
306                         number_p ++;
307                     }
308 
309                 }
310                 else
311                 {
312                     if(points.data[1][i]>=f.eval(points.data[0][i]))
313                     {
314                         number_n ++;
315                     }
316                 }
317             }
318             return (value_length*length)*((number_p-number_n)*1.0/n);
319         }
320         else
321         {
322             if((min == 0)&&(max != 0))
323             {
324                 Matrix points = Scatter_function(a,b,0,max,n);
325 
326                 for(unsigned i = 0 ; i < n ; i++)
327                 {
328                     if(points.data[1][i]<=f.eval(points.data[0][i]))
329                     {
330                         number_p ++;
331                     }
332                 }
333 
334                 return (length*value_length)*(number_p*1.0/n);
335             }
336             else if((min != 0)&&(max == 0))
337             {
338                 Matrix points = Scatter_function(a,b,min,max,n);
339                 for(unsigned i = 0 ; i < n ; i++)
340                 {
341                     if(points.data[1][i]>=f.eval(points.data[0][i]))
342                     {
343                         number_n ++;
344                     }
345                 }
346                 return -(length*value_length)*(number_n*1.0/n);
347             }
348             else
```

```
349             {
350                 return 0;
351             }
352         }
353     }
354     else
355     {
356         cout << "The integrator_Monte_Carlo can only make approximation for integration from a
    ↪  to b in which a and b isn't infty"<<endl;
357         exit(-1);
358     }
359 }
360 
361 double integrator_Rimmen(MathFunc f,double lowerlimit,double upperlimit,unsigned n)
362 {
363     double a = lowerlimit;
364     double b = upperlimit;
365     double length = b-a;
366     double stepsize = length/n;
367     double sum=0;
368     double x = a;
369     for(unsigned i=0;i<n;i++)
370     {
371         x=a+i*stepsize;
372         sum=sum + f.eval(x) * stepsize;
373     }
374     return sum;
375 }
```