Problem 4

Ruipeng Li, School of Physics, 1400011365

May 10, 2017

Abstract

程序的具体说明请参见 README.md 文件，本报告主要是进行一些算法的分析以及结果的报告。首先我们给出问题的描述和分析，接着我们会分析具体的算法和表现、误差，最后我们会讨论下计算结果的物理意义。代码请参见 https://github.com/Viciberty/CP.

1 Analysis

我们要求的是给定离散点，求一个数值积分，主要分为两个步骤。

- 先对离散点进行插值，并确定 a,b,c,d，主要目的是尽量减少其误差;
- 再对插值函数进行数值积分，主要目的是保证收敛性的同时提高精度。

我们的目标是尽量提高插值和积分的精度，这要求我们用更为精确的算法。对于插值，我们选取了 N 次拉格朗日插值，并调用了开源代码实现的三次样条插值进行对比。对于求根，由于我们要求的两个值基本都在拟合出来的单调区间内，故我采用了二分查找的方式求根，并通过控制结束搜索的区间长度来控制根的精度。最后对于积分，我采用了 N 次的复化牛顿柯特斯公式。

2 Algorithm

2.1 Interpolation

首先，我们对给出的 V 和 M 各 51 个均匀的离散点进行插值。我自己写了一个 N 阶拉格朗日插值，然后为了对比结论，又引用了一个三次样条插值。下图2.1给出的是二阶拉格朗日插值的结果。

2.1.1 算法选择

1. 精度

我们优先考虑较为简便快捷的多项式插值，截断误差为 $R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\omega_{n+1}(x)$，其中 $\omega_{n+1}(x) = \prod_{i=0}^{n}(x - x_i), \xi \in (a, b)$, 比较适合于被插函数较为光滑的情形。用于我们需要插值都是能量这种物理中的函数，一般性质都比较优良，所以我认为用拉格朗日插值就能达到

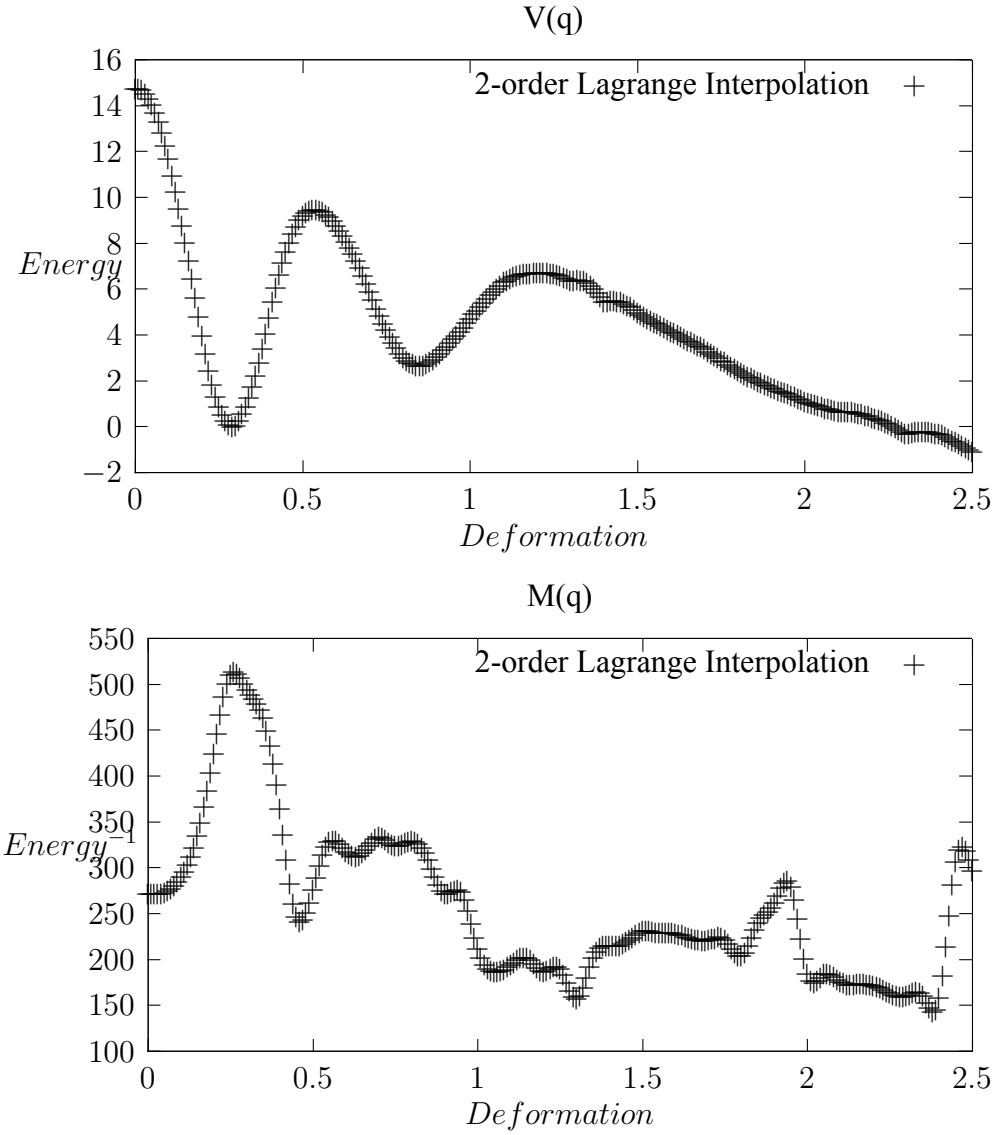

Figure 1: 原始数据插值

较好的效果。为了减少高次插值带来的龙格效应，我因此选择了分段插值，若数据点不够刚好分段，那么最后几个点则用更低阶的方式进行插值。同时我也调用了开源代码，实现了三次样条插值这一插值方式。通过对比，用三次样条插值算的的结果与 mathematica 等科学计算软件算的结果几乎是一致的。

在这里数值计算带来的舍入误差相对来说是小的，因为实验测量的精度问题，只有三位小数，而且算法稳定。

2. 速度

时间复杂度 $O(n)$, 只用先搜索到输入值的所在区间，直接计算插值即可。

3. 稳定性

算法很稳定。

2.1.2 算法表现

通过单元测试，能保证算法的正确性，即在给定的点值是一致的，在插值区间也能得到光滑的插值。不同的插值方式在这里并不能体现出优劣，但实际上，由于物理中的函数都是光滑的，我们可以相信，样条插值出的函数相对来说会更加精准。

2.2 Extract Root

首先，对于不同阶的插值，我们本质上要求解的是一个非线性方程求根的问题。在这里，由于所求的根都落在拟合的单调区间内，所以我们采用了二分法查找根。同时这种方法的另一种好处是我可以通过控制结束查找区间的大小来控制所求的根的精度，方便后期继续求解积分。

2.2.1 算法选择

1. 精度

二分法求单根，精度可以由自行通过 ϵ 控制。

2. 速度

二分法的缺点就是收敛速度较慢，是线性收敛，比起牛顿法的二次收敛要慢很多。但在我们的问题中，这样的速度尚可接受。再就是时间复杂度 $O(log_2 n)$,n 为划分的区间个数，就是精度的倒数。

3. 稳定性

算法总是收敛的，而且数值形态好。

2.2.2 算法表现

通过单元测试，我测试了一些函数的单根，是正确的。而且可以对精度进行准确控制，在这里出现的误差相对来说较小。

2.3 Integral

首先，我们本质上是要求解一个瑕积分，在函数的两端均是瑕点，我们的做法是在指定区间往里一个 ϵ 进行积分, 再对忽略的部分进行误差分析。积分我们用的是 N 次的复化牛顿柯特斯公式。（当 $n \geq 8$ 时，科特斯系数出现负值，计算会不稳定，我们通常只用 1、2、4 次，均在程序中实现了）主要计算用的是二次的 Simpson 公式，综合考量了时间与精度的选择。为了提升精度，我们还可以选用自适应的辛普森算法。

2.3.1 算法选择

1. 精度
辛普森公式有三次代数精度，还是相当高的。$R[f] = -\frac{b-a}{180}(\frac{b-a}{2})^4 f^{(4)}(\eta), \eta \in (a, b)$

2. 速度
辛普森公式的时间复杂度取决于分划区间，为 O(n), 主要时间用来求值。

3. 稳定性
科特斯算法对 $n \leq 8$ 总是收敛的，而且数值形态良好。对于辛普森公式，我们知其稳定性良好。

2.3.2 算法表现

通过单元测试，我测试了一些常见解析函数的积分求值，都是准确的。下面讨论下这个数值积分函数的表现，主要是针对分划和积分区间平移的稳定性。
1. 分划
情况 1, 对积分 T 与 W，对不同的分划，积分值如下表:

Value \ Function / Division	T	W
10^3	3.17762	50.2936
10^4	3.1178	50.2939
10^5	3.1161	50.2939
10^6	3.1161	50.2939

情况 2，对不同分划，积分值如下表:

Value \ Function / Division	T	W
10^3	9.05447	16.9009
10^4	8.69931	16.901
10^5	8.68124	16.901
10^6	8.68107	16.901

我们知道对分划而言，对情况 1，W 由于是正常积分，故在分划为 10^4 时，就已经收敛了，而 T 由于是反常积分，故在 10^5 时，才收敛，对情况 2 的 T，这个反常积分也在趋于收敛。

2. 对于瑕积分端点的选取

由于二分求值的精度所限，为了保证根号下一定大于 0，我们取得积分区间为根向内左右各缩小了 ϵ，即二分法求根的精度，保证根号下一定是大于 0 的。

函数表达式为 $\int_a^b \frac{g(x)}{\sqrt{x-a}}dx, |b-a| < \epsilon, g(x) < +\infty$, 我们不妨设 $|g(x)| < N$, 那么积分值 $< N\int_a^b \frac{1}{\sqrt{x-a}}dx = 2N\sqrt{|b-a|} < 2N\sqrt{\epsilon}$, 按理说应该是收敛的，但却是以 $\sqrt{\epsilon}$ 收敛的。我们利用如下代码输出 N 大致的值 $cout << T\_help(root\_a + error * epsilon, f[1], f[0], E\_0[0]) * sqrt(epsilon) << endl;$，N 约为 3.791508，这样误差约为 0.002398，相对误差为 0.077%, 几乎可以忽略不计。因此我们对瑕积分做这样的近似是相对合理的。

3 Results

3.1 Results display

插值采用二阶拉格朗日插值，二分法求根（精度为 0.0000001），simpson 公式求积分（分划为 100000）时.

1. $E_0 = 0.9MeV$ 时

$$a = 0.248704, b = 0.33049, c = 2.03757, \tau = 9.9522e + 22$$

2. $E_0 = 4.8MeV$ 时

$$a = 0.189898, b = 0.400845, x = 0.730046, y = 1.00563, c = 1.51872, \tau = 3.35509e - 06$$

插值采用三次样条插值，二分法求根（精度为 0.0000001），simpson 公式求积分（分划为 100000）时.

1. $E_0 = 0.9MeV$ 时

$$a = 0.248804, b = 0.332631, c = 2.03761, \tau = 7.54335e + 22$$

2. $E_0 = 4.8MeV$ 时

$$a = 0.189148, b = 0.400932, x = 0.731019, y = 1.00592, c = 1.51653, \tau = 2.74074e - 06$$

采用三次样条与 Mathematica 等数值计算软件能达到较为一致的计算结果，也说明可能这些科学计算软件对于这样的插值采取的三次样条插值。

3.2 Results Analysis

经过上面三步算法的分析，我们知道，总共的误差有截断误差、舍入误差、观测误差。截断误差主要来自于插值时对函数的拟合带来的误差，舍入误差是在牛顿法控制精度后和 double 类型决定的误差。而由于观测数据本来就有的误差，我们还需要进行所谓的误差传递的计算，但在计算物理这门课我们就不加以讨论了。通过以上讨论，我们对精度还可以有更高的提高要求，不过是在牺牲时间的情况下，比如提升找根的精度和采用更高阶的数

值积分算法，比如自适应的辛普森算法也能很好的提高精度。

而对于程序速度来说，由于数据规模较小，基本在可以接受的范围内。求值的算法为 $O(n)$, 实际上也可以二分查找为 $O(log_2 n)$, 但由于 n=51 就没有必要去优化了。二分求根的时间复杂度为 $O(log_2 n)$ 总体来说还是挺快的，主要的时间都在计算数值积分上了，辛普森的积分公式为 $O(n)$, 但由于求值带来的常数比较大，所以是主要的时间贡献项。我们可以采用 OPENMP 或者 CUDA 对积分进行优化，由于其无数据、程序上的依赖性，直接开多线程即可。

3.3 Physical significance

当能量为 0.9MeV 时，体系处于束缚态，故衰变寿命较长，当能量为 4.8MeV 时，体系处于扩展态，故寿命较短，而且衰减的非常之快，符合物理直觉。

4 Appendix: Code

```
#include <iostream>
#include <functional>
#include <math.h>
#include "spline.h"
#include <vector>

using namespace std;
#define epsilon 0.0000001  //Bisection Accuracy
#define error 1
#define division 100000
#define hbar 6.582119513e-22


/*****************************************************
 *Interpolation
 *****************************************************/
class Interpolation
{
    int n;
    std::vector<double> x;
    std::vector<double> y;
    tk::spline s;
public:
    void _Initial(const double* x_, const double* y_, int n_)//
       Initialization
    {
        n = n_; //n data
```

```
        for (int i=0;i<n;i++) {x.push_back(x_[i]); y.push_back(y_
            [i]);}
        s.set_points(x,y);
    }
/************************************************
*Calculate Function Value
*order<=0 3-degree spline interpolation
*order==1 linear interpolation
*order==2 quadratic interpolation
*order or more
************************************************/
    double value(double var, int order=0)
    {
        if ((var < x[0]) || (var > x[n-1])) {cout<<"Wrong Input\n
            "; return 0;} //In the definition domain
        double result=0; //function value
    if (order>0) //order interpolation
    {
        int index = search(var, order, 0, n-1) / order *order;//
            Fine Interval
        if (index >= (n-1) - (n-1)%order) order = (n-1)%order;//
            The last serval points use lower orders
             double l[order+1];
             for (int i=index; i< index+order+1; i++){
                 l[i-index]=1.0;
                 for (int j=index; j< index+order+1; j++)
                 {if (i!=j) l[i-index]*=(var - x[j])/(x[i]-x[j]);}
                 result+=l[i-index]*y[i];
                 }
     }
     else //3-degree spline interpolation
     {
        result = s(var);
     }
    return result;
 }


/************************************************
*Calculate Root-Bisection
************************************************/
//value_interval
    int* value_interval(double var, int* temp)
    {
        int count=0;
```

```
        for (int i=0; i<n; i++)
        {
            if ((y[i]-var)*(y[i+1]-var)<=0) {temp[count]=i; count
               ++;}
            }
        return temp;
    }
//calculate root-bisection
    double root(double begin, double end, double var)
    {
        if ((end-begin)<epsilon) return (end+begin)/2;
        if ((value(begin)-var)*(value((begin+end)/2)-var) <= 0)
           return root(begin,(begin+end)/2,var);
         else if ((value(end)-var)*(value((begin+end)/2)-var) <=
            0) return root((begin+end)/2,end,var);
         else return -1;
    }


private://search
    int search(double var, int order, int begin, int end)
    {
        for (int i=0;i<n;i++)
            if ((var >= x[i])&&(var <= x[i+1])) return i;
        return -1;
    }
};

/*********************************************************
 *Integral Function
 *order=1 trapezoid method
 *order=2 simpson method

 *share is the partition of the set
 *********************************************************/
double integral(double (*f_)(double,Interpolation,Interpolation,
   double),double begin,double end, Interpolation M,
   Interpolation V,double Energy, int order=2,int share =
   division)
{
    auto f = [&](double x){return f_(x,M,V,Energy);};
    double sum=0;
    double delta=(end-begin)*1.0/share;

    double a=begin;
```

```
    double b=begin,mid;
    for (int count= 0; count<share ; count++)
    {
        a = begin+count*delta;
        b = begin+(count+1)*delta;
        mid = (a+b)/2.0;
        switch (order)
        {
        case 1: {sum +=delta*f(mid); break;}
        case 2: {sum += (f(a)+ 4.0*f(mid)+f(b))*delta/6.0; break
           ;}
        case 4: {sum += (0.0778*f(a)+0.3556*f(a+(b-a)/4)+0.1333*f
           (a+2*(b-a)/4)+0.3556*f(a+3*(b-a)/4)+0.0778*f(b))*delta
           ; break;
            }
        }
    }
    return sum;
}


/*********************************************************
 *Auxiliary Function
 *********************************************************/

//Function for Assistance

double W_help(double q, Interpolation M, Interpolation V, double
   Energy)
{
    return sqrt(M.value(q)*(V.value(q)-Energy));
}

double T_help(double q, Interpolation M, Interpolation V, double
   Energy)
{
    return sqrt(M.value(q)/(Energy-V.value(q)));
}



/*****************************************************************

 *Main Program
```

```
 ************************************************************************

int main(int argc, const char * argv[]) {

/*********************************************************
 *Data Initialization
 *********************************************************/
    const double q[51] = {
       0,0.05,0.1,0.15,0.2,0.25,0.3,0.35,0.4,0.45,0.5,0.55,0.6,0.65,0.7,0.75
        };
    const double V[51] = {
       14.757,14.022,11.665,7.996,3.953,0.848,0,1.727,4.741,7.607,9.199,9.42
        };
    const double M[51] = {
       271.7416,272.5246,289.6114,334.4802,424.6238,510.052,493.5916,462.747
        };
    const double E_0[2] = {0.9,4.8};
/*********************************************************
*Interpolation
*********************************************************/
    Interpolation f[2];
    f[0]._Initial(q, V, 51);  //V(q)
    f[1]._Initial(q, M, 51);  //M(q)
    double root_a,root_b,root_x,root_y,root_c;
/*********************************************************
*Numerical Integration1  E_0 = 0.9
 *********************************************************/

    int temp_1[3];
    f[0].value_interval(0.9, temp_1);
    root_a = f[0].root(q[temp_1[0]], q[temp_1[0]+1], E_0[0]);
    root_b = f[0].root(q[temp_1[1]], q[temp_1[1]+1], E_0[0]);
    root_c = f[0].root(q[temp_1[2]], q[temp_1[2]+1], E_0[0]);
    cout<<"ROOT1: "<<root_a <<" "<<root_b<<" "<<root_c<<endl;

    double W = exp(-2*integral(W_help, root_b+epsilon, root_c-
       epsilon, f[1],f[0],E_0[0]));
    double T = hbar*integral(T_help, root_a+error*epsilon, root_b
      -error*epsilon, f[1],f[0],E_0[0]);
    cout<<"tau = "<<T/W<<" when E_0 = 0.9"<<endl<<endl;

/*********************************************************
*Numerical Integration2  E_0 = 4.8
 *********************************************************/
```

```
    int temp[5];
    f[0].value_interval(4.8, temp);
    root_a = f[0].root(q[temp[0]], q[temp[0]+1], E_0[1]);
    root_b = f[0].root(q[temp[1]], q[temp[1]+1], E_0[1]);
    root_x = f[0].root(q[temp[2]], q[temp[2]+1], E_0[1]);
    root_y = f[0].root(q[temp[3]], q[temp[3]+1], E_0[1]);
    root_c = f[0].root(q[temp[4]], q[temp[4]+1], E_0[1]);
    cout<<"ROOT2: "<< root_a <<" "<<root_b<<" "<<root_x<<" "<<
       root_y<<" "<<root_c<<endl;

    W = exp(-2*(integral(W_help, root_b+epsilon, root_x-epsilon,
       f[1],f[0],E_0[1])+integral(W_help, root_y+epsilon, root_c-
       epsilon, f[1],f[0],E_0[1])));
    T = hbar*(integral(T_help, root_a+error*epsilon, root_b-error
       *epsilon, f[1],f[0],E_0[1])+integral(T_help, root_x+error*
       epsilon, root_y-error*epsilon, f[1],f[0],E_0[1]));
    cout<<"tau2 = "<<T/W<<" when E_0 = 4.8 "<<endl;

    return 0;
}

    /*
 * spline.h
 *
 * simple cubic spline interpolation library without external
 * dependencies
 *
 *
    ---------------------------------------------------------------------

 * Copyright (C) 2011, 2014 Tino Kluge (ttk448 at gmail.com)
 *
 *  This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License
 *  as published by the Free Software Foundation; either version
    2
 *  of the License, or (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be
    useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty
    of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
```

```
 *  You should have received a copy of the GNU General Public
    License
 *  along with this program.  If not, see <http://www.gnu.org/
    licenses/>.
 *
    ------------------------------------------------------------------

 *
 */


#ifndef TK_SPLINE_H
#define TK_SPLINE_H

#include <cstdio>
#include <cassert>
#include <vector>
#include <algorithm>


// unnamed namespace only because the implementation is in this
// header file and we don't want to export symbols to the obj
   files
namespace
{

namespace tk
{

// band matrix solver
class band_matrix
{
private:
    std::vector< std::vector<double> > m_upper;  // upper band
    std::vector< std::vector<double> > m_lower;  // lower band
public:
    band_matrix() {};                             // constructor
    band_matrix(int dim, int n_u, int n_l);       // constructor
    ~band_matrix() {};                            // destructor
    void resize(int dim, int n_u, int n_l);      // init with dim
       ,n_u,n_l
    int dim() const;                             // matrix
       dimension
    int num_upper() const
    {
```

```
        return m_upper.size()-1;
    }
    int num_lower() const
    {
        return m_lower.size()-1;
    }
    // access operator
    double & operator () (int i, int j);            // write
    double   operator () (int i, int j) const;      // read
    // we can store an additional diogonal (in m_lower)
    double& saved_diag(int i);
    double  saved_diag(int i) const;
    void lu_decompose();
    std::vector<double> r_solve(const std::vector<double>& b)
       const;
    std::vector<double> l_solve(const std::vector<double>& b)
       const;
    std::vector<double> lu_solve(const std::vector<double>& b,
                                 bool is_lu_decomposed=false);

};


// spline interpolation
class spline
{
public:
    enum bd_type {
        first_deriv = 1,
        second_deriv = 2
    };

private:
    std::vector<double> m_x,m_y;            // x,y coordinates of
        points
    // interpolation parameters
    // f(x) = a*(x-x_i)^3 + b*(x-x_i)^2 + c*(x-x_i) + y_i
    std::vector<double> m_a,m_b,m_c;        // spline
       coefficients
    double  m_b0, m_c0;                     // for left extrapol
    bd_type m_left, m_right;
    double  m_left_value, m_right_value;
    bool    m_force_linear_extrapolation;

public:
```

```
    // set default boundary condition to be zero curvature at
       both ends
    spline(): m_left(second_deriv), m_right(second_deriv),
        m_left_value(0.0), m_right_value(0.0),
        m_force_linear_extrapolation(false)
    {
        ;
    }

    // optional, but if called it has to come be before
       set_points()
    void set_boundary(bd_type left, double left_value,
                      bd_type right, double right_value,
                      bool force_linear_extrapolation=false);
    void set_points(const std::vector<double>& x,
                    const std::vector<double>& y, bool
                       cubic_spline=true);
    double operator() (double x) const;
    double deriv(int order, double x) const;
};



//
   ------------------------------------------------------------------

// implementation part, which could be separated into a cpp file
//
   ------------------------------------------------------------------



// band_matrix implementation
// -------------------------

band_matrix::band_matrix(int dim, int n_u, int n_l)
{
    resize(dim, n_u, n_l);
}
void band_matrix::resize(int dim, int n_u, int n_l)
{
    assert(dim>0);
    assert(n_u>=0);
    assert(n_l>=0);
    m_upper.resize(n_u+1);
```

```
    m_lower.resize(n_l+1);
    for(size_t i=0; i<m_upper.size(); i++) {
        m_upper[i].resize(dim);
    }
    for(size_t i=0; i<m_lower.size(); i++) {
        m_lower[i].resize(dim);
    }
}
int band_matrix::dim() const
{
    if(m_upper.size()>0) {
        return m_upper[0].size();
    } else {
        return 0;
    }
}


// defines the new operator (), so that we can access the
   elements
// by A(i,j), index going from i=0,...,dim()-1
double & band_matrix::operator () (int i, int j)
{
    int k=j-i;        // what band is the entry
    assert( (i>=0) && (i<dim()) && (j>=0) && (j<dim()) );
    assert( (-num_lower()<=k) && (k<=num_upper()) );
    // k=0 -> diogonal, k<0 lower left part, k>0 upper right part
    if(k>=0)   return m_upper[k][i];
    else            return m_lower[-k][i];
}
double band_matrix::operator () (int i, int j) const
{
    int k=j-i;        // what band is the entry
    assert( (i>=0) && (i<dim()) && (j>=0) && (j<dim()) );
    assert( (-num_lower()<=k) && (k<=num_upper()) );
    // k=0 -> diogonal, k<0 lower left part, k>0 upper right part
    if(k>=0)   return m_upper[k][i];
    else            return m_lower[-k][i];
}
// second diag (used in LU decomposition), saved in m_lower
double band_matrix::saved_diag(int i) const
{
    assert( (i>=0) && (i<dim()) );
    return m_lower[0][i];
}
```

```
double & band_matrix::saved_diag(int i)
{
    assert( (i>=0) && (i<dim()) );
    return m_lower[0][i];
}

// LR-Decomposition of a band matrix
void band_matrix::lu_decompose()
{
    int  i_max,j_max;
    int  j_min;
    double x;

    // preconditioning
    // normalize column i so that a_ii=1
    for(int i=0; i<this->dim(); i++) {
        assert(this->operator()(i,i)!=0.0);
        this->saved_diag(i)=1.0/this->operator()(i,i);
        j_min=std::max(0,i-this->num_lower());
        j_max=std::min(this->dim()-1,i+this->num_upper());
        for(int j=j_min; j<=j_max; j++) {
            this->operator()(i,j) *= this->saved_diag(i);
        }
        this->operator()(i,i)=1.0;          // prevents rounding
           errors
    }

    // Gauss LR-Decomposition
    for(int k=0; k<this->dim(); k++) {
        i_max=std::min(this->dim()-1,k+this->num_lower());  //
           num_lower not a mistake!
        for(int i=k+1; i<=i_max; i++) {
            assert(this->operator()(k,k)!=0.0);
            x=-this->operator()(i,k)/this->operator()(k,k);
            this->operator()(i,k)=-x;                        //
               assembly part of L
            j_max=std::min(this->dim()-1,k+this->num_upper());
            for(int j=k+1; j<=j_max; j++) {
                // assembly part of R
                this->operator()(i,j)=this->operator()(i,j)+x*
                   this->operator()(k,j);
            }
        }
    }
}
```

```
// solves Ly=b
std::vector<double> band_matrix::l_solve(const std::vector<double
   >& b) const
{
    assert( this->dim()==(int)b.size() );
    std::vector<double> x(this->dim());
    int j_start;
    double sum;
    for(int i=0; i<this->dim(); i++) {
        sum=0;
        j_start=std::max(0,i-this->num_lower());
        for(int j=j_start; j<i; j++) sum += this->operator()(i,j)
           *x[j];
        x[i]=(b[i]*this->saved_diag(i)) - sum;
    }
    return x;
}
// solves Rx=y
std::vector<double> band_matrix::r_solve(const std::vector<double
   >& b) const
{
    assert( this->dim()==(int)b.size() );
    std::vector<double> x(this->dim());
    int j_stop;
    double sum;
    for(int i=this->dim()-1; i>=0; i--) {
        sum=0;
        j_stop=std::min(this->dim()-1,i+this->num_upper());
        for(int j=i+1; j<=j_stop; j++) sum += this->operator()(i,
           j)*x[j];
        x[i]=( b[i] - sum ) / this->operator()(i,i);
    }
    return x;
}

std::vector<double> band_matrix::lu_solve(const std::vector<
   double>& b,
        bool is_lu_decomposed)
{
    assert( this->dim()==(int)b.size() );
    std::vector<double>  x,y;
    if(is_lu_decomposed==false) {
        this->lu_decompose();
    }
    y=this->l_solve(b);
```

```
    x=this->r_solve(y);
    return x;
}



// spline implementation
// -----------------------

void spline::set_boundary(spline::bd_type left, double left_value
    ,
                          spline::bd_type right, double
                             right_value,
                          bool force_linear_extrapolation)
{
    assert(m_x.size()==0);          // set_points() must not have
        happened yet
    m_left=left;
    m_right=right;
    m_left_value=left_value;
    m_right_value=right_value;
    m_force_linear_extrapolation=force_linear_extrapolation;
}


void spline::set_points(const std::vector<double>& x,
                        const std::vector<double>& y, bool
                           cubic_spline)
{
    assert(x.size()==y.size());
    assert(x.size()>2);
    m_x=x;
    m_y=y;
    int   n=x.size();
    // TODO: maybe sort x and y, rather than returning an error
    for(int i=0; i<n-1; i++) {
        assert(m_x[i]<m_x[i+1]);
    }

    if(cubic_spline==true) { // cubic spline interpolation
        // setting up the matrix and right hand side of the
           equation system
        // for the parameters b[]
        band_matrix A(n,1,1);
```

```
std::vector<double>  rhs(n);
for(int i=1; i<n-1; i++) {
    A(i,i-1)=1.0/3.0*(x[i]-x[i-1]);
    A(i,i)=2.0/3.0*(x[i+1]-x[i-1]);
    A(i,i+1)=1.0/3.0*(x[i+1]-x[i]);
    rhs[i]=(y[i+1]-y[i])/(x[i+1]-x[i]) - (y[i]-y[i-1])/(x
       [i]-x[i-1]);
}
// boundary conditions
if(m_left == spline::second_deriv) {
    // 2*b[0] = f''
    A(0,0)=2.0;
    A(0,1)=0.0;
    rhs[0]=m_left_value;
} else if(m_left == spline::first_deriv) {
    // c[0] = f', needs to be re-expressed in terms of b:
    // (2b[0]+b[1])(x[1]-x[0]) = 3 ((y[1]-y[0])/(x[1]-x
       [0]) - f')
    A(0,0)=2.0*(x[1]-x[0]);
    A(0,1)=1.0*(x[1]-x[0]);
    rhs[0]=3.0*((y[1]-y[0])/(x[1]-x[0])-m_left_value);
} else {
    assert(false);
}
if(m_right == spline::second_deriv) {
    // 2*b[n-1] = f''
    A(n-1,n-1)=2.0;
    A(n-1,n-2)=0.0;
    rhs[n-1]=m_right_value;
} else if(m_right == spline::first_deriv) {
    // c[n-1] = f', needs to be re-expressed in terms of
       b:
    // (b[n-2]+2b[n-1])(x[n-1]-x[n-2])
    // = 3 (f' - (y[n-1]-y[n-2])/(x[n-1]-x[n-2]))
    A(n-1,n-1)=2.0*(x[n-1]-x[n-2]);
    A(n-1,n-2)=1.0*(x[n-1]-x[n-2]);
    rhs[n-1]=3.0*(m_right_value-(y[n-1]-y[n-2])/(x[n-1]-x
       [n-2]));
} else {
    assert(false);
}

// solve the equation system to obtain the parameters b[]
m_b=A.lu_solve(rhs);
```

```
        // calculate parameters a[] and c[] based on b[]
        m_a.resize(n);
        m_c.resize(n);
        for(int i=0; i<n-1; i++) {
            m_a[i]=1.0/3.0*(m_b[i+1]-m_b[i])/(x[i+1]-x[i]);
            m_c[i]=(y[i+1]-y[i])/(x[i+1]-x[i])
                   - 1.0/3.0*(2.0*m_b[i]+m_b[i+1])*(x[i+1]-x[i]);
        }
    } else { // linear interpolation
        m_a.resize(n);
        m_b.resize(n);
        m_c.resize(n);
        for(int i=0; i<n-1; i++) {
            m_a[i]=0.0;
            m_b[i]=0.0;
            m_c[i]=(m_y[i+1]-m_y[i])/(m_x[i+1]-m_x[i]);
        }
    }

    // for left extrapolation coefficients
    m_b0 = (m_force_linear_extrapolation==false) ? m_b[0] : 0.0;
    m_c0 = m_c[0];

    // for the right extrapolation coefficients
    // f_{n-1}(x) = b*(x-x_{n-1})^2 + c*(x-x_{n-1}) + y_{n-1}
    double h=x[n-1]-x[n-2];
    // m_b[n-1] is determined by the boundary condition
    m_a[n-1]=0.0;
    m_c[n-1]=3.0*m_a[n-2]*h*h+2.0*m_b[n-2]*h+m_c[n-2];   // = f'_
       {n-2}(x_{n-1})
    if(m_force_linear_extrapolation==true)
        m_b[n-1]=0.0;
}

double spline::operator() (double x) const
{
    size_t n=m_x.size();
    // find the closest point m_x[idx] < x, idx=0 even if x<m_x
       [0]
    std::vector<double>::const_iterator it;
    it=std::lower_bound(m_x.begin(),m_x.end(),x);
    int idx=std::max( int(it-m_x.begin())-1, 0);

    double h=x-m_x[idx];
    double interpol;
```

```
    if(x<m_x[0]) {
        // extrapolation to the left
        interpol=(m_b0*h + m_c0)*h + m_y[0];
    } else if(x>m_x[n-1]) {
        // extrapolation to the right
        interpol=(m_b[n-1]*h + m_c[n-1])*h + m_y[n-1];
    } else {
        // interpolation
        interpol=((m_a[idx]*h + m_b[idx])*h + m_c[idx])*h + m_y[
           idx];
    }
    return interpol;
}

double spline::deriv(int order, double x) const
{
    assert(order>0);

    size_t n=m_x.size();
    // find the closest point m_x[idx] < x, idx=0 even if x<m_x
       [0]
    std::vector<double>::const_iterator it;
    it=std::lower_bound(m_x.begin(),m_x.end(),x);
    int idx=std::max( int(it-m_x.begin())-1, 0);

    double h=x-m_x[idx];
    double interpol;
    if(x<m_x[0]) {
        // extrapolation to the left
        switch(order) {
        case 1:
            interpol=2.0*m_b0*h + m_c0;
            break;
        case 2:
            interpol=2.0*m_b0*h;
            break;
        default:
            interpol=0.0;
            break;
        }
    } else if(x>m_x[n-1]) {
        // extrapolation to the right
        switch(order) {
        case 1:
            interpol=2.0*m_b[n-1]*h + m_c[n-1];
```

```
            break;
        case 2:
            interpol=2.0*m_b[n-1];
            break;
        default:
            interpol=0.0;
            break;
        }
    } else {
        // interpolation
        switch(order) {
        case 1:
            interpol=(3.0*m_a[idx]*h + 2.0*m_b[idx])*h + m_c[idx
               ];
            break;
        case 2:
            interpol=6.0*m_a[idx]*h + 2.0*m_b[idx];
            break;
        case 3:
            interpol=6.0*m_a[idx];
            break;
        default:
            interpol=0.0;
            break;
        }
    }
    return interpol;
}



} // namespace tk


} // namespace

#endif /* TK_SPLINE_H */
```