Федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский политехнический университет
Петра Великого»
Институт компьютерных наук и технологий
Программная инженерия

ПОЛИТЕХ
Санкт-Петербургский
политехнический университет
Петра Великого

Отчёт по лабораторным работам

по дисциплине «Вычислительная математика»

Студент гр. 5130904/30008 Ребдев П.А
Руководитель: Скуднева Е.В

Санкт-Петербург
2025

Оглавление

Лабораторная работа №1

1. Задание

Вариант №16

Для функции f(x) = 1 - exp(-x) по узлам x_k= 0.3k (k=0,1, …10) построить полином Лагранжа L(x) 10-й степени и сплайн-функцию S(x). Вычислить значения всех трех функций в точках y_k = 0.15 +0.3k (k=0,1, …9). Результаты отобразить графически.

Используя программу QUANC8, вычислить интегралы:

$\int_{0.5}^{1} |\sin(x) - 0.6|^m \, dx$, для m = -1 и для m = -0.5

2. Решение

2.1 Ход решения

1. Построение вектора из 10 x координат по заранее заданному правилу (x[i] = (0.3d * i))

2. Вычисление вектора значений функции в точках, необходимого для построения интерполяционного полинома Лагранжа и SPLINE

3. Построение интерполяционного полинома Лагранжа с использованием вычисленного на прошлом шаге вектора значений функций

4. Построение SPLINE с использованием вычисленного на втором шаге вектора значений функций

5. Смещение изначального вектора x координат на +0.15

6. Вычисление значений функции, интерполяционного полинома Лагранжа и SPLINE в новых x

7. Вычисление погрешности интерполяционного полинома Лагранжа и SPLINE

2.2 Основная программа

main.cpp

```
#include <iostream>
#include <vector>
#include <cmath>

#include "myFuncs.hpp"
#include "paint.hpp"
#include "quanc8.hpp"

int main()
{
 const unsigned pointsNum = 10;

 std::cout <<  std::fixed << std::setprecision(8);
 std::cout << "\033[1;32m";
 std::cout << "Base points = 0,3 * k (k = 0, 1, ...10)\nf(x) = 1 - exp(x)\nPoints to calculate = 0,15 + 0,3k (k = 0, 1, ...9)\n";
 std::cout << "\033[0m\n";

 std::vector< double > x(pointsNum);

 for (size_t i = 0; i <= pointsNum; ++i)
 {
  x[i] = (0.3d * i);
 }
```

```
 std::vector< double > orig(pointsNum);
 std::vector< double > Lagrange(pointsNum);
 std::vector< double > spline(pointsNum);

 for (size_t i = 0; i < pointsNum; ++i)
 {
  double point = 0.15d + 0.3d * i;
  std::cout << "x = " << point;

  orig[i] = baseFunc(point);
  std::cout << ";\t f(x) = " << orig[i];

  Lagrange[i] = LagrangePolynomial(pointsNum, x, point);
  std::cout << ";\t L(x) = " << Lagrange[i];

  spline[i] = splineNum(pointsNum, x, point);
  std::cout << ";\t S(x) = " << spline[i] << '\n';
 }
 std::cout << '\n';
 paint(x, orig, Lagrange, spline);

 return 0;
}
```

2.3 Интерполяционный полином Лагранжа

myFuncs.cpp

```
double LagrangePolynomial(const unsigned pointsNum, const std::vector< double > & x, double point)
{
 double result = 0;
 for (size_t i = 0; i < pointsNum; ++i)
 {
  double localResult = baseFunc(x[i]);
  for (size_t j = 0; j < pointsNum; ++j)
  {
   if (j != i)
   {
    localResult *= (point - x[j]);
    localResult /= (x[i] - x[j]);
   }
  }
  result += localResult;
 }
 return result;
}
```

2.4 Программа spline

myFuncs.cpp

```
double splineNum(const unsigned pointsNum, const std::vector< double > & x, double point)
{
 std::vector< double > func(pointsNum);
 for (size_t i = 0; i < pointsNum; ++i)
 {
```

```
    func[i] = baseFunc(x[i]);
  }
  tk::spline s(x, func);
  return s(point);
}
```

2.5 Программа QUANC8

main.cpp

```
  double * result = new double;
  double * errest = new double;
  int * nofunR = new int;
  double * posnR = new double;
  int * flag = new int;

  std::cout << "\033[1;32m";
  std::cout << "Find integral from 0.5 to 1 abs(sin(x) -0.6)^m dx. With m = -1 and m = -0.5";
  std::cout << "\033[0m\n";
  quanc8(quanc1, 0.5, 1, std::pow(10, -6), std::pow(10, -7), result, errest, nofunR, posnR, flag);
  std::cout << "I(-1) = " << *result << '\n';
  quanc8(quanc2, 0.5, 1, std::pow(10, -6), std::pow(10, -7), result, errest, nofunR, posnR, flag);
  std::cout << "I(-0.5) = " << *result << "\n\n";

  delete result;
  delete errest;
  delete nofunR;
  delete posnR;
  delete flag;
```

myFuncs.cpp

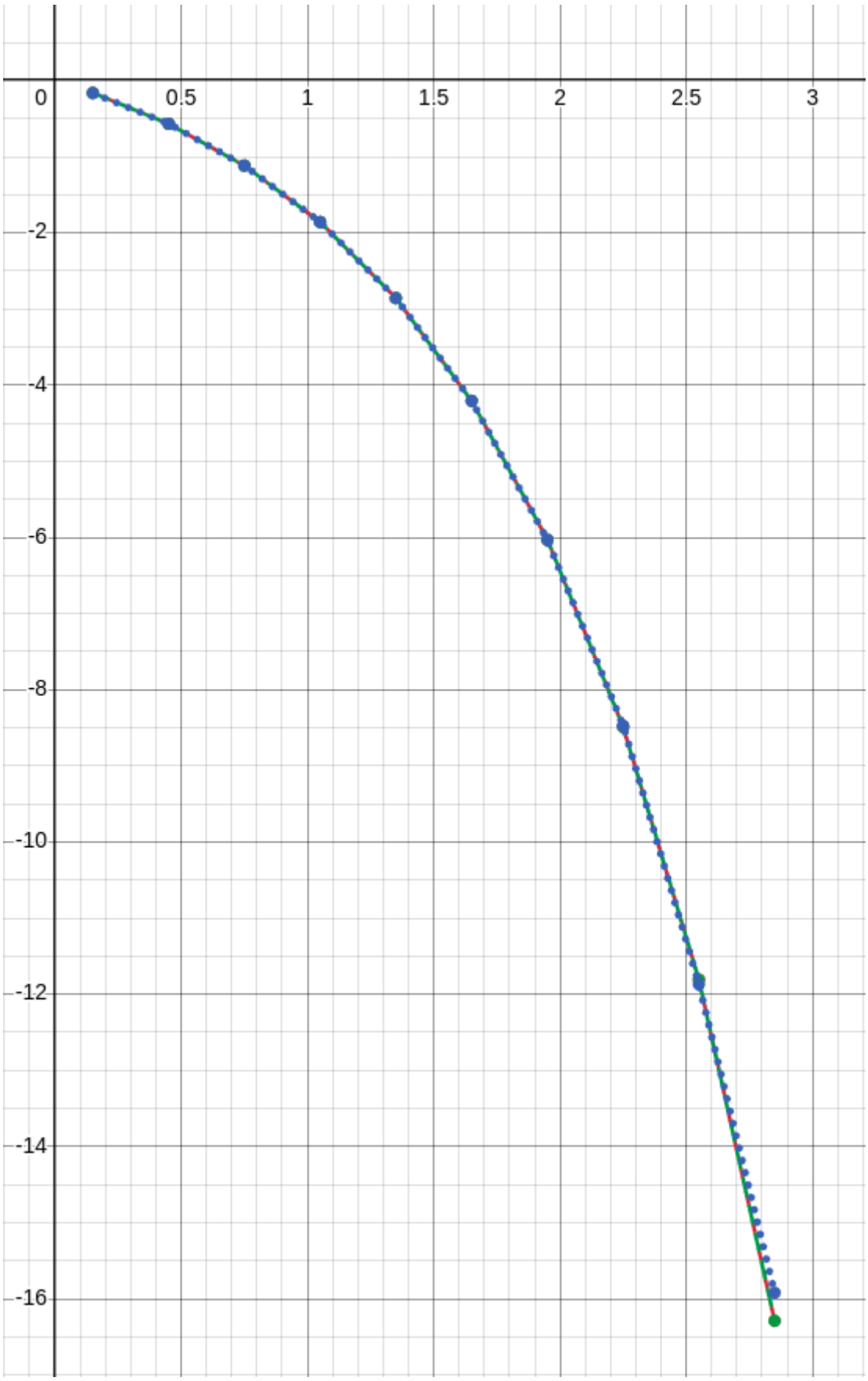
```
double quanc1(double x)
{
  return std::pow(std::abs(std::sin(x) - 0.6), -1);
}
double quanc2(double x)
{
  return std::pow(std::abs(std::sin(x) - 0.6), -0.5);
}
```

3. Результаты

3.1 Нахождение значения функции в точках

В результате вычислений была получена следующая таблица:

x	f(x)	L(x)	|L(x) - f(x)|	S(x)	|S(x) - f(x)|
0.15	-0.161834	-0.161834	0	-0.165899	0,004065
0.45	-0.568312	-0.568312	0	-0.567179	0,001133
0.75	-1.117000	-1.117000	0	-1.117272	0.000272
1.05	-1.857651	-1.857651	0	-1.857430	0,000221
1.35	-2.857426	-2.857426	0	-2.857681	0,000255
1.65	-4.206980	-4.206980	0	-4.205697	0,001283
1.95	-6.028688	-6.028688	0	-6.032910	0,004222
2.25	-8.487736	-8.487736	0	-8.471245	0,016491
2.55	-11.807104	-11.807104	0	-11.867653	0,060549
2.85	-16.287782	-16.287777	$5 * 10^{-6}$	-15.919744	0,368038



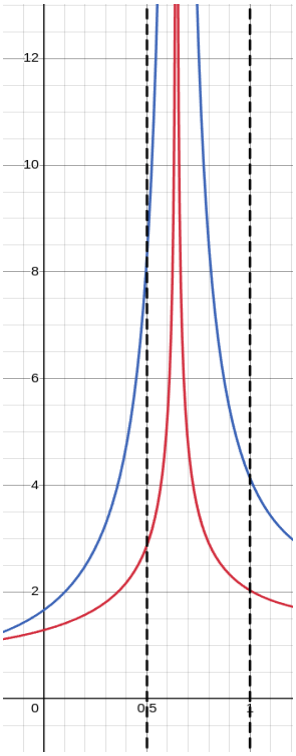
На графике красным обозначена оригинальная функция, зелёным интерполяционный полином Лагранжа, синим сплайн функция

3.2 Вычисления интеграла

При вычисления интеграла с помощью программы QUANC8 были получены следующие результаты:

m = -1; I = 60.089301

m = -0.5; I = 2.2098554



На графики чёрным пунктиром обозначены границы интегрирования, красным m = -0.5, синим m = -1

4. Выводы

В ходе выполнения лабораторной работы с помощью интерполяционного полинома Лагранжа и программы SPLINE были найдены значения функции в точках. При использование типа данных double (16 чисел после мантиссы), и округление до 6го знака после запятой были получены следующие погрешности:

	полином Лагранжа	программа SPLINE
Максимальная разница	0,000005	0,368038
	0,00003%	2,25%
Минимальная разница	0	0,000272
	0	0,0002%

По данным из таблицы можно сделать следующие выводы:

1. Полином Лагранжа отлично подходит для аппроксимации функций, погрешность в 6м разряде начинает появляться при вычисление чисел с двумя целыми разрядами, что является довольно хорошим результатом

2. Высокая погрешность программы SPLINE скорее всего свидетельствует о не качественности конкретной реализации SPLINE алгоритма.

С помощью программы QUANC8 были вычислены значения интеграла для m = -0.5 и m = -1 с абсолютной и относительной погрешностями равными 10^-7. Погрешность вычисления при m = -0.5 составила 0.00000003, а при m = 0.01433744. Можно сделать вывод что программа QUANC8 вычисляет значение интеграла с высокой точностью.

5. Дополнения

5.1 Полный код всех программ

Описание файлов:

- main.cpp отвечает за хранение данных и их вывод в консоль
- myFunc.* отвечают за реализацию интерполяционного полинома Лагранжа, вычисление значения функции и является «прослойкой» между main.cpp, spline.h и quanc8.cpp
- paint.* отвечает за визуализацию вычисленных значений функции, интерполяционного полинома Лагранжа и программы SPLINE

main.cpp

```
#include <iostream>
#include <vector>
#include <cmath>
#include <iomanip>

#include "myFuncs.hpp"
#include "paint.hpp"
#include "quanc8.hpp"

int main()
{
 const unsigned pointsNum = 10;

 std::cout <<  std::fixed << std::setprecision(8);
 std::cout << "\033[1;32m";
 std::cout << "Base points = 0,3 * k (k = 0, 1, ...10)\nf(x) = 1 - exp(x)\nPoints to calculate = 0,15 + 0,3k (k = 0, 1, ...9)\n";
 std::cout << "\033[0m\n";

 std::vector< double > x(pointsNum);

 for (size_t i = 0; i <= pointsNum; ++i)
 {
  x[i] = (0.3d * i);
 }

 std::vector< double > orig(pointsNum);
 std::vector< double > Lagrange(pointsNum);
 std::vector< double > spline(pointsNum);

 for (size_t i = 0; i < pointsNum; ++i)
 {
  double point = 0.15d + 0.3d * i;
  std::cout << "x = " << point;

  orig[i] = baseFunc(point);
```

```
    std::cout << ";\t f(x) = " << orig[i];

    Lagrange[i] = LagrangePolynomial(pointsNum, x, point);
    std::cout << ";\t L(x) = " << Lagrange[i];

    spline[i] = splineNum(pointsNum, x, point);
    std::cout << ";\t S(x) = " << spline[i] << '\n';
  }
  std::cout << '\n';
  paint(x, orig, Lagrange, spline);


  double * result = new double;
  double * errest = new double;
  int * nofunR = new int;
  double * posnR = new double;
  int * flag = new int;

  std::cout << "\033[1;32m";
  std::cout << "Find integral from 0.5 to 1 abs(sin(x) -0.6)^m dx. With m = -1 and m = -0.5";
  std::cout << "\033[0m\n";
  quanc8(quanc1, 0.5, 1, std::pow(10, -7), std::pow(10, -7), result, errest, nofunR, posnR, flag);
  std::cout << "I(-1) = " << *result << '\n';
  quanc8(quanc2, 0.5, 1, std::pow(10, -7), std::pow(10, -7), result, errest, nofunR, posnR, flag);
  std::cout << "I(-0.5) = " << *result << "\n\n";

  delete result;
  delete errest;
  delete nofunR;
  delete posnR;
  delete flag;

  return 0;
}
```

myFuncs.hpp

```
#ifndef MYFUNCS_HPP
#define MYFUNCS_HPP

#include <vector>

double baseFunc(double point);
double LagrangePolynomial(const unsigned pointsNum, const std::vector< double > & x, double point);
double splineNum(const unsigned pointsNum, const std::vector< double > & x, double point);
double quanc1(double x);
double quanc2(double x);

#endif
```

myFuncs.cpp

```
#include "myFuncs.hpp"

#include <cmath>

#include "spline.h"
```

```
double baseFunc(double point)
{
 return (1 - std::exp(point));
}
double LagrangePolynomial(const unsigned pointsNum, const std::vector< double > & x, double point)
{
 double result = 0;
 for (size_t i = 0; i < pointsNum; ++i)
 {
  double localResult = baseFunc(x[i]);
  for (size_t j = 0; j < pointsNum; ++j)
  {
   if (j != i)
   {
    localResult *= (point - x[j]);
    localResult /= (x[i] - x[j]);
   }
  }
  result += localResult;
 }
 return result;
}
double splineNum(const unsigned pointsNum, const std::vector< double > & x, double point)
{
 std::vector< double > func(pointsNum);
 for (size_t i = 0; i < pointsNum; ++i)
 {
  func[i] = baseFunc(x[i]);
 }
 tk::spline s(x, func);
 return s(point);
}
double quanc1(double x)
{
 return std::pow(std::abs(std::sin(x) - 0.6), -1);
}
double quanc2(double x)
{
 return std::pow(std::abs(std::sin(x) - 0.6), -0.5);
}
```

paint.hpp

```
#ifndef PAINT_HPP
#define PAINT_HPP

#include <vector>

using vec = std::vector< double >;

void paint(const vec & x, const vec & orig, const vec & Lagrange, const vec & spline);

#endif
```

paint.cpp

```
#include "paint.hpp"
```

```
#include <SFML/Graphics.hpp>

void paint(const vec & x, const vec & orig, const vec & Lagrange, const vec & spline)
{
 const double scale = 100;
 const double w = 1920;
 const double h = 1080;

 sf::RenderWindow window(sf::VideoMode(w, h), "UwU");

 while (window.isOpen())
 {
  sf::Event event;
  while (window.pollEvent(event))
  {
   if (event.type == sf::Event::Closed)
   {
    window.close();
   }
  }
  window.clear(sf::Color::White);

  for (size_t i = 0; i < x.size(); ++i)
  {
   sf::CircleShape circle(5);
   circle.setPosition((w / 20) - orig[i] * scale, (h / 3) + x[i] * scale);
   circle.setFillColor(sf::Color(255, 0, 0, 150));
   window.draw(circle);

   circle.setPosition((w / 20) - Lagrange[i] * scale, (h / 3) + x[i] * scale);
   circle.setFillColor(sf::Color(0, 255, 0, 150));
   window.draw(circle);

   circle.setPosition((w / 20) - spline[i] * scale, (h / 3) + x[i] * scale);
   circle.setFillColor(sf::Color(0, 0, 255, 150));
   window.draw(circle);
  }
  for (size_t i = 0; i < orig.size() - 1; ++i)
  {
   sf::Vertex line[] = {
    sf::Vertex(sf::Vector2f((w / 20) - orig[i] * scale, (h / 3) + x[i] * scale), sf::Color(255, 0, 0, 150)),
    sf::Vertex(sf::Vector2f((w / 20) - orig[i + 1] * scale, (h / 3) + x[i + 1] * scale), sf::Color(255, 0, 0, 150))
   };
   window.draw(line, 5, sf::Lines);

   line[0] = sf::Vertex(sf::Vector2f((w / 20) - Lagrange[i] * scale, (h / 3) + x[i] * scale), sf::Color(0, 255, 0, 150));
   line[1] = sf::Vertex(sf::Vector2f((w / 20) - Lagrange[i + 1] * scale, (h / 3) + x[i + 1] * scale), sf::Color(0, 255, 0, 150));
   window.draw(line, 5, sf::Lines);

   line[0] = sf::Vertex(sf::Vector2f((w / 20) - spline[i] * scale, (h / 3) + x[i] * scale), sf::Color(0, 0, 255, 150));
   line[1] = sf::Vertex(sf::Vector2f((w / 20) - spline[i + 1] * scale, (h / 3) + x[i + 1] * scale), sf::Color(0, 0, 255, 150));
   window.draw(line, 5, sf::Lines);
  }
  window.display();
 }
}
```

quanc8.hpp

```
#pragma once

int quanc8(double(*fun)(double x), double a, double b,
           double abserr, double relerr,
           double *resultR, double *errestR,
           int *nofunR,
           double *posnR, int *flag);
```

quanc8.cpp

```
#include <math.h>


/*
 fun       : The name of the user defined function f(x).
 a        : The lower limit of integration.
 b        : The upper limit of integration.
 abserr    : The absolute error tolerance. Should be positive.
 relerr    : The relative error tolerance. Should be positive.

 result    : An approximation to the integral I.
 errest    : An estimate of the absolute error in I.
 nofun      : The number of function evaluations used in the
                              calculation of result.
 posn      : If flag < 0, then posn is the point reached when
                              the limit on nfe was approached.
 flag     : Error indicator.
                              = 0, normal return.
                              = 1, invalid user input, relerr < 0, abserr < 0
                              < 0, If flag = -n then n subintervals did not
                                        converge.  A small number (say 10) of
                                        unconverged subintervals may be acceptable.
                                        Check posn for further information.
*/


int quanc8(double(*fun)(double x), double a, double b,
           double abserr, double relerr,
           double *resultR, double *errestR,
           int *nofunR,
           double *posnR, int *flag)

{  /* begin function quanc8 */

  /* this code has been translated from fortran, hence use
  elements 1 .. n */
           double w0, w1, w2, w3, w4, area, x0, f0, stone, step, cor11;
           double qprev, qnow, qdiff, qleft, esterr, tolerr;
           static double qright[32], f[17], x[17], fsave[9][31];
           static double xsave[9][31];
           double posn, result, errest;
           double temp, temp1;
           int nofun;
           int levmin, levmax, levout, nomax, nofin, lev, nim, i, j, ii;
```

```
/* check user input */
if (abserr < 0.0 || relerr < 0.0)
{
        *flag = 1;
        *resultR = 0.0;
        *errestR = 0.0;
        *posnR = 0.0;
        *nofunR = 0;
        return (0);
}

/*  ***  stage 1 ***  general initialization
set constants.  */
levmin = 1;
levmax = 30;
levout = 6;
nomax = 5000;

{
        ii = 1;
        for (i = 0; i <= levout; i++) ii *= 2;
} /* ---> ii = 2 ** (levout+1) */
nofin = nomax - 8 * (levmax - levout + ii);
/* note that there will be trouble when nofun reaches nofin */

temp = 14175.0;
w0 = 3956.0 / temp;
w1 = 23552.0 / temp;
w2 = -3712.0 / temp;
w3 = 41984.0 / temp;
w4 = -18160.0 / temp;

/* initialize running sums to zero. */
*flag = 0;
posn = 0.0;
result = 0.0;
cor11 = 0.0;
errest = 0.0;
area = 0.0;
nofun = 0;
if (a == b) goto ExitQuanc8;

/* ***  stage 2 ***  initialization for first interval */
lev = 0;
nim = 1;
x0 = a;
x[16] = b;
qprev = 0.0;

/* set up evenly spaced panels */
f0 = (*fun) (x0);
stone = (b - a) / 16.0;
x[8] = 0.5 * (x0 + x[16]);
x[4] = 0.5 * (x0 + x[8]);
x[12] = 0.5 * (x[8] + x[16]);
x[2] = 0.5 * (x0 + x[4]);
```

```
        x[6] = 0.5 * (x[4] + x[8]);
        x[10] = 0.5 * (x[8] + x[12]);
        x[14] = 0.5 * (x[12] + x[16]);
        for (j = 2; j <= 16; j = j + 2)  f[j] = (*fun) (x[j]);
        nofun = 9;

        /* ***  stage 3 ***  central calculation
        requires qprev,x0,x2,x4,...,x16,f0,f2,f4,...,f16.
        calculates x1,x3,...x15, f1,f3,...f15,qleft,qright,qnow,qdiff,area.
        */
Stage3:
        x[1] = 0.5 * (x0 + x[2]);
        f[1] = (*fun) (x[1]);
        for (j = 3; j <= 15; j = j + 2)
        {
                x[j] = 0.5 * (x[j - 1] + x[j + 1]);
                f[j] = (*fun) (x[j]);
        }
        nofun += 8;
        step = (x[16] - x0) / 16.0;
        qleft = (w0 * (f0 + f[8]) + w1 * (f[1] + f[7]) + w2 * (f[2] + f[6])
                + w3 * (f[3] + f[5]) + w4 * f[4]) * step;
        qright[lev + 1] = (w0 * (f[8] + f[16]) + w1 * (f[9] + f[15]) + w2 * (f[10] + f[14])
                + w3 * (f[11] + f[13]) + w4 * f[12]) * step;
        qnow = qleft + qright[lev + 1];
        qdiff = qnow - qprev;
        area += qdiff;

        /* ***  stage 4 *** interval convergence test */
        esterr = fabs(qdiff) / 1023.0;
        {
                tolerr = abserr;
                temp = relerr * fabs(area);
                if (temp > tolerr) tolerr = temp;
        }  /* ---> tolerr = max(abserr, relerr * abs(area)) */
        tolerr *= (step / stone);

        if (lev < levmin)     goto Stage5;  /* keep subdividing              */
        if (lev >= levmax)    goto Stage6B; /* too many nested levels        */
        if (nofun > nofin)    goto Stage6;  /* close to limit on fn calls    */
        if (esterr <= tolerr) goto Stage7;  /* this interval has converged   */

                                                                        /* ***  stage 5  ***  no
convergence
                                                                        locate next interval.          */
Stage5:
        nim *= 2;
        ++lev;

        /* store right hand elements for future use. */
        for (i = 1; i <= 8; i++)
        {
                fsave[i][lev] = f[i + 8];
                xsave[i][lev] = x[i + 8];
        }
```

```
        /* assemble left hand elements for immediate use. */
        qprev = qleft;
        for (i = 1; i <= 8; i++)
        {
                j = (-i);
                f[2 * j + 18] = f[j + 9];
                x[2 * j + 18] = x[j + 9];
        }

        goto Stage3;

        /* ***  stage 6  ***  trouble section
        number of function values is about to exceed limit.  */
Stage6:
        nofin *= 2;
        levmax = levout;
        posn = x0;   /* this is our trouble spot */
        goto Stage7;

        /* current level is levmax. */
Stage6B:
        --(*flag);   /* another interval has not converged */

                                /* ***  stage 7  ***  interval converged
                                add contributions into running sums.  */
Stage7:
        result += qnow;
        errest += esterr;
        cor11 += qdiff / 1023.0;

        /* locate next interval. */
        while (nim != (2 * (nim / 2)))
        {
                nim /= 2;
                --lev;
        }
        ++nim;
        if (lev <= 0) goto Stage8;

        /* assemble elements required for the next interval. */
        qprev = qright[lev];
        x0 = x[16];
        f0 = f[16];
        for (i = 1; i <= 8; i++)
        {
                f[2 * i] = fsave[i][lev];
                x[2 * i] = xsave[i][lev];
        }
        goto Stage3;

        /* ***  stage 8  ***  finalize and return */
Stage8:
        result += cor11;

        /* make sure errest not less than roundoff level. */
        if (errest == 0.0) goto ExitQuanc8;
```

```
        temp1 = fabs(result);
        temp = temp1 + errest;
        while (temp == temp1)
        {
                errest *= 2.0;
                temp = temp1 + errest;
        }

ExitQuanc8:
        /* --- return variables --- */
        *resultR = result;
        *errestR = errest;
        *posnR = posn;
        *nofunR = nofun;
        return (0);
}  /* end of quanc8() */
```

spline.h

```
/*
 * spline.h
 *
 * simple cubic spline interpolation library without external
 * dependencies
 *
 * ---------------------------------------------------------------------
 * Copyright (C) 2011, 2014, 2016, 2021 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 <cmath>
#include <vector>
#include <algorithm>
#ifdef HAVE_SSTREAM
#include <sstream>
#include <string>
#endif // HAVE_SSTREAM
```

```
// not ideal but disable unused-function warnings
// (we get them because we have implementations in the header file,
// and this is because we want to be able to quickly separate them
// into a cpp file if necessary)
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-function"

// 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
{

// spline interpolation
class spline
{
public:
   // spline types
   enum spline_type {
      linear = 10,          // linear interpolation
      cspline = 30,         // cubic splines (classical C^2)
      cspline_hermite = 31    // cubic hermite splines (local, only C^1)
   };

   // boundary condition type for the spline end-points
   enum bd_type {
      first_deriv = 1,
      second_deriv = 2
   };

protected:
   std::vector<double> m_x,m_y;          // x,y coordinates of points
   // interpolation parameters
   // f(x) = a_i + b_i*(x-x_i) + c_i*(x-x_i)^2 + d_i*(x-x_i)^3
   // where a_i = y_i, or else it won't go through grid points
   std::vector<double> m_b,m_c,m_d;       // spline coefficients
   double m_c0;                          // for left extrapolation
   spline_type m_type;
   bd_type m_left, m_right;
   double  m_left_value, m_right_value;
   bool m_made_monotonic;
   void set_coeffs_from_b();             // calculate c_i, d_i from b_i
   size_t find_closest(double x) const;    // closest idx so that m_x[idx]<=x

public:
   // default constructor: set boundary condition to be zero curvature
   // at both ends, i.e. natural splines
   spline(): m_type(cspline),
      m_left(second_deriv), m_right(second_deriv),
      m_left_value(0.0), m_right_value(0.0), m_made_monotonic(false)
   {
      ;
   }
```

```
  spline(const std::vector<double>& X, const std::vector<double>& Y,
         spline_type type = cspline,
         bool make_monotonic = false,
         bd_type left  = second_deriv, double left_value  = 0.0,
         bd_type right = second_deriv, double right_value = 0.0
        ):
      m_type(type),
      m_left(left), m_right(right),
      m_left_value(left_value), m_right_value(right_value),
      m_made_monotonic(false) // false correct here: make_monotonic() sets it
  {
      this->set_points(X,Y,m_type);
      if(make_monotonic) {
          this->make_monotonic();
      }
  }


  // modify boundary conditions: if called it must be before set_points()
  void set_boundary(bd_type left, double left_value,
                    bd_type right, double right_value);

  // set all data points (cubic_spline=false means linear interpolation)
  void set_points(const std::vector<double>& x,
                  const std::vector<double>& y,
                  spline_type type=cspline);

  // adjust coefficients so that the spline becomes piecewise monotonic
  // where possible
  //   this is done by adjusting slopes at grid points by a non-negative
  //   factor and this will break C^2
  //   this can also break boundary conditions if adjustments need to
  //   be made at the boundary points
  // returns false if no adjustments have been made, true otherwise
  bool make_monotonic();

  // evaluates the spline at point x
  double operator() (double x) const;
  double deriv(int order, double x) const;

  // returns the input data points
  std::vector<double> get_x() const { return m_x; }
  std::vector<double> get_y() const { return m_y; }
  double get_x_min() const { assert(!m_x.empty()); return m_x.front(); }
  double get_x_max() const { assert(!m_x.empty()); return m_x.back(); }

#ifdef HAVE_SSTREAM
  // spline info string, i.e. spline type, boundary conditions etc.
  std::string info() const;
#endif // HAVE_SSTREAM

};



namespace internal
```

```
{

// 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 (int)m_upper.size()-1;
   }
   int num_lower() const
   {
      return (int)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 diagonal (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);

};

} // namespace internal




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

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

void spline::set_boundary(spline::bd_type left, double left_value,
                          spline::bd_type right, double right_value)
{
   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;
```

```
}


void spline::set_coeffs_from_b()
{
   assert(m_x.size()==m_y.size());
   assert(m_x.size()==m_b.size());
   assert(m_x.size()>2);
   size_t n=m_b.size();
   if(m_c.size()!=n)
      m_c.resize(n);
   if(m_d.size()!=n)
      m_d.resize(n);

   for(size_t i=0; i<n-1; i++) {
      const double h  = m_x[i+1]-m_x[i];
      // from continuity and differentiability condition
      m_c[i] = ( 3.0*(m_y[i+1]-m_y[i])/h - (2.0*m_b[i]+m_b[i+1]) ) / h;
      // from differentiability condition
      m_d[i] = ( (m_b[i+1]-m_b[i])/(3.0*h) - 2.0/3.0*m_c[i] ) / h;
   }

   // for left extrapolation coefficients
   m_c0 = (m_left==first_deriv) ? 0.0 : m_c[0];
}

void spline::set_points(const std::vector<double>& x,
                  const std::vector<double>& y,
                  spline_type type)
{
   assert(x.size()==y.size());
   assert(x.size()>2);
   m_type=type;
   m_made_monotonic=false;
   m_x=x;
   m_y=y;
   int n = (int) x.size();
   // check strict monotonicity of input vector x
   for(int i=0; i<n-1; i++) {
      assert(m_x[i]<m_x[i+1]);
   }


   if(type==linear) {
      // linear interpolation
      m_d.resize(n);
      m_c.resize(n);
      m_b.resize(n);
      for(int i=0; i<n-1; i++) {
         m_d[i]=0.0;
         m_c[i]=0.0;
         m_b[i]=(m_y[i+1]-m_y[i])/(m_x[i+1]-m_x[i]);
      }
      // ignore boundary conditions, set slope equal to the last segment
      m_b[n-1]=m_b[n-2];
      m_c[n-1]=0.0;
```

```
    m_d[n-1]=0.0;
  } else if(type==cspline) {
    // classical cubic splines which are C^2 (twice cont differentiable)
    // this requires solving an equation system

    // setting up the matrix and right hand side of the equation system
    // for the parameters b[]
    internal::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*c[0] = f''
      A(0,0)=2.0;
      A(0,1)=0.0;
      rhs[0]=m_left_value;
    } else if(m_left == spline::first_deriv) {
      // b[0] = f', needs to be re-expressed in terms of c:
      // (2c[0]+c[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*c[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) {
      // b[n-1] = f', needs to be re-expressed in terms of c:
      // (c[n-2]+2c[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 c[]
    m_c=A.lu_solve(rhs);

    // calculate parameters b[] and d[] based on c[]
    m_d.resize(n);
    m_b.resize(n);
    for(int i=0; i<n-1; i++) {
      m_d[i]=1.0/3.0*(m_c[i+1]-m_c[i])/(x[i+1]-x[i]);
      m_b[i]=(y[i+1]-y[i])/(x[i+1]-x[i])
          - 1.0/3.0*(2.0*m_c[i]+m_c[i+1])*(x[i+1]-x[i]);
```

```
    }
    // for the right extrapolation coefficients (zero cubic term)
    // f_{n-1}(x) = y_{n-1} + b*(x-x_{n-1}) + c*(x-x_{n-1})^2
    double h=x[n-1]-x[n-2];
    // m_c[n-1] is determined by the boundary condition
    m_d[n-1]=0.0;
    m_b[n-1]=3.0*m_d[n-2]*h*h+2.0*m_c[n-2]*h+m_b[n-2];   // = f'_{n-2}(x_{n-1})
    if(m_right==first_deriv)
      m_c[n-1]=0.0;   // force linear extrapolation

  } else if(type==cspline_hermite) {
    // hermite cubic splines which are C^1 (cont. differentiable)
    // and derivatives are specified on each grid point
    // (here we use 3-point finite differences)
    m_b.resize(n);
    m_c.resize(n);
    m_d.resize(n);
    // set b to match 1st order derivative finite difference
    for(int i=1; i<n-1; i++) {
      const double h  = m_x[i+1]-m_x[i];
      const double hl = m_x[i]-m_x[i-1];
      m_b[i] = -h/(hl*(hl+h))*m_y[i-1] + (h-hl)/(hl*h)*m_y[i]
             +  hl/(h*(hl+h))*m_y[i+1];
    }
    // boundary conditions determine b[0] and b[n-1]
    if(m_left==first_deriv) {
      m_b[0]=m_left_value;
    } else if(m_left==second_deriv) {
      const double h = m_x[1]-m_x[0];
      m_b[0]=0.5*(-m_b[1]-0.5*m_left_value*h+3.0*(m_y[1]-m_y[0])/h);
    } else {
      assert(false);
    }
    if(m_right==first_deriv) {
      m_b[n-1]=m_right_value;
      m_c[n-1]=0.0;
    } else if(m_right==second_deriv) {
      const double h = m_x[n-1]-m_x[n-2];
      m_b[n-1]=0.5*(-m_b[n-2]+0.5*m_right_value*h+3.0*(m_y[n-1]-m_y[n-2])/h);
      m_c[n-1]=0.5*m_right_value;
    } else {
      assert(false);
    }
    m_d[n-1]=0.0;

    // parameters c and d are determined by continuity and differentiability
    set_coeffs_from_b();

  } else {
    assert(false);
  }

  // for left extrapolation coefficients
  m_c0 = (m_left==first_deriv) ? 0.0 : m_c[0];
}
```

```
bool spline::make_monotonic()
{
   assert(m_x.size()==m_y.size());
   assert(m_x.size()==m_b.size());
   assert(m_x.size()>2);
   bool modified = false;
   const int n=(int)m_x.size();
   // make sure: input data monotonic increasing --> b_i>=0
   //           input data monotonic decreasing --> b_i<=0
   for(int i=0; i<n; i++) {
      int im1 = std::max(i-1, 0);
      int ip1 = std::min(i+1, n-1);
      if( ((m_y[im1]<=m_y[i]) && (m_y[i]<=m_y[ip1]) && m_b[i]<0.0) ||
         ((m_y[im1]>=m_y[i]) && (m_y[i]>=m_y[ip1]) && m_b[i]>0.0) ) {
         modified=true;
         m_b[i]=0.0;
      }
   }
   // if input data is monotonic (b[i], b[i+1], avg have all the same sign)
   // ensure a sufficient criteria for monotonicity is satisfied:
   //    sqrt(b[i]^2+b[i+1]^2) <= 3 |avg|, with avg=(y[i+1]-y[i])/h,
   for(int i=0; i<n-1; i++) {
      double h = m_x[i+1]-m_x[i];
      double avg = (m_y[i+1]-m_y[i])/h;
      if( avg==0.0 && (m_b[i]!=0.0 || m_b[i+1]!=0.0) ) {
         modified=true;
         m_b[i]=0.0;
         m_b[i+1]=0.0;
      } else if( (m_b[i]>=0.0 && m_b[i+1]>=0.0 && avg>0.0) ||
               (m_b[i]<=0.0 && m_b[i+1]<=0.0 && avg<0.0) ) {
         // input data is monotonic
         double r = sqrt(m_b[i]*m_b[i]+m_b[i+1]*m_b[i+1])/std::fabs(avg);
         if(r>3.0) {
            // sufficient criteria for monotonicity: r<=3
            // adjust b[i] and b[i+1]
            modified=true;
            m_b[i]   *= (3.0/r);
            m_b[i+1] *= (3.0/r);
         }
      }
   }

   if(modified==true) {
      set_coeffs_from_b();
      m_made_monotonic=true;
   }

   return modified;
}

// return the closest idx so that m_x[idx] <= x (return 0 if x<m_x[0])
size_t spline::find_closest(double x) const
{
   std::vector<double>::const_iterator it;
   it=std::upper_bound(m_x.begin(),m_x.end(),x);      // *it > x
   size_t idx = std::max( int(it-m_x.begin())-1, 0);   // m_x[idx] <= x
```

```
  return idx;
}

double spline::operator() (double x) const
{
  // polynomial evaluation using Horner's scheme
  // TODO: consider more numerically accurate algorithms, e.g.:
  //   - Clenshaw
  //   - Even-Odd method by A.C.R. Newbery
  //   - Compensated Horner Scheme
  size_t n=m_x.size();
  size_t idx=find_closest(x);

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

double spline::deriv(int order, double x) const
{
  assert(order>0);
  size_t n=m_x.size();
  size_t idx = find_closest(x);

  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_c0*h + m_b[0];
      break;
    case 2:
      interpol=2.0*m_c0;
      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_c[n-1]*h + m_b[n-1];
      break;
    case 2:
```

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

#ifdef HAVE_SSTREAM
std::string spline::info() const
{
    std::stringstream ss;
    ss << "type " << m_type << ", left boundary deriv " << m_left << " = ";
    ss << m_left_value << ", right boundary deriv " << m_right << " = ";
    ss << m_right_value << std::endl;
    if(m_made_monotonic) {
      ss << "(spline has been adjusted for piece-wise monotonicity)";
    }
    return ss.str();
}
#endif // HAVE_SSTREAM


namespace internal
{

// 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 -> diagonal, 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 -> diagonal, 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;
}

} // namespace internal


} // namespace tk


} // namespace

#pragma GCC diagnostic pop

#endif /* TK_SPLINE_H */
```