Project 3, SF2565

Hanna Hultin, hhultin@kth.se, 931122-2468, TTMAM2
Mikael Persson, mikaepe@kth.se, 850908-0456, TTMAM2

November 30, 2016

Short review

Given a domain D enclosed by four curves $\mathbf{x}_i(p_i)$, $i = 1, 2, 3, 4$, (where p_i are any suitable parameters for the curves) we want to generate a structured grid on the domain. For this we define grid parameters $\xi_1 \in [0, 1]$ and $\xi_2 \in [0, 1]$. The unit square $R = [0, 1] \times [0, 1]$ is discretized using

$$\begin{aligned} \xi_{1,i} &= ih_1, \quad i = 0, 1, \dots, n, \\ \xi_{2,j} &= jh_2, \quad j = 0, 1, \dots, m. \end{aligned}$$

where $h_1 = 1/n$ and $h_2 = 1/m$. See Figure 1. Define linear interpolation functions $\phi_1(x) = 1$ and $\phi_2(x) = 1 - x$. Assuming bijective maps $\Phi_1 : [0, 1] \mapsto \mathbf{x}_1(p_1)$ and similarly for the three other boundaries, the grid may now be generated using the algebraic grid generation formula:

$$\begin{aligned} \mathbf{x}_{ij} = {} & \phi_2(\xi_{1,i})\Phi_4(\xi_{2,j}) + \phi_1(\xi_{1,i})\Phi_2(\xi_{2,j}) && \text{[left to right interpolation]} \\ & + \phi_2(\xi_{2,j})\Phi_1(\xi_{1,i}) + \phi_1(\xi_{2,j})\Phi_3(\xi_{1,i}) && \text{[bottom to top interpolation]} \\ & - \phi_2(\xi_{1,i})\phi_2(\xi_{2,j})\Phi_1(0) - \phi_1(\xi_{1,i})\phi_2(\xi_{2,j})\Phi_1(1) && \text{[corrections, corners]} \\ & - \phi_2(\xi_{1,i})\phi_1(\xi_{2,j})\Phi_3(0) - \phi_1(\xi_{1,i})\phi_1(\xi_{2,j})\Phi_3(1). \end{aligned}$$

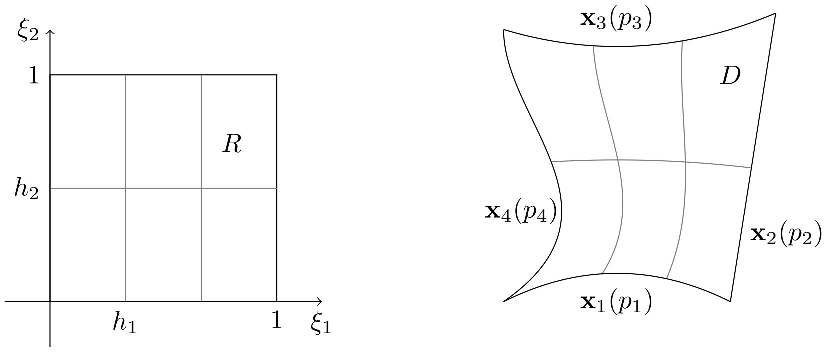

Figure 1: The domain D is enclosed by four curves parametrized by the parameter p. The grid is generated using maps from the boundary curves of $[0, 1] \times [0, 1]$ to the boundary curves of the domain D.

Task 1: The Curvebase class

In this project we use an abstract class `Curvebase` to represent curves. The class is written so that it should be easy to derive different classes from `Curvebase` to represent a wide range of different curves.

We have private virtual functions for determining the value of x and y as well as the derivatives with respect to x and y for the curve given the user parameter p. p is supposed to take values between a and b which are private members of the class. We also have a private member variable called `length` which is supposed to be the arc length of the curve.

We have public member functions for determining x and y given the parameter s which is a scaled parameter that goes from 0 to 1. To determine the corresponding x and y we use a private member function called `newtonsolve` which uses Newton's method. To compute the integral of the arc length between two values a and b `newtonsolve` calls the private function `integrate`. This function uses the private member functions `iSimpson` and `i2Simpson` to compute the integral according to Project 1.

Task 2: The derived classes

From the abstract class `Curvebase` we derived the classes `xLine`, `yLine` and `fxcurve` to be able to create the boundary curves for the desired grid.

`xLine` is a class that represents curves which has a constant y-value. Here we set a to be the initial value of x, b the final value of x and length is just set to $b - a$ since we have a straight line. We also have a private variable for the constant y-value called $yConst$.

We use x as the used parameter so the private functions for determining the values of x and y and the derivatives just becomes: $x(p) = p$, $y(p) = yConst$, $dx(p) = 1$ and $dy(p) = 0$.

We also use that we can overwrite the functions for determining x and y given the parameter s since in this special case these values can be computed easily. So we use that $x(s) = a + s * length$ and $y(s) = yConst$. We also use that the arc length integral between x_1 and x_2 is just $x_2 - x_1$.

`yLine` is a class that represents curves which has a constant x-value. For `yLine` everything looks the same as for `xLine` but with x and y interchanged everywhere.

Lastly we have the class `fxcurve`. This is a specific class used to to represent the given lower boundary curve. Here we use x as the user parameter p and then we implemented that $x(p) = p$, $y(p) = f(p)$, $dx(p) = 1$ and $dy(p) = f'(p)$ where $f(\cdot)$ is given in the project description and expression for the derivative was computed analytically.

Task 3: The Domain class

In this task we designed the class Domain which is a general class for modelling 4-sided domains and structured grids on them. This class takes references to four objects of type `Curvebase` as arguments to the constructor. The curves represented by these `Curvebase` objects are then the four sides of the `Domain`.

The class `Domain` has a public member function called `grid_generation` which generates a grid according to the algebraic grid generation formula.

Task 4: Writing the grid to a file and plotting it in Matlab

To be able to write the generated grid to a file we added the public function `writeFile` to our class `Domain`. This function opens a binary file and writes first all the x-values for the grid points and then the y-values for all the grid points before closing the file.

We then opened the binary file in Matlab and plotted the grid points. See Figure 2.

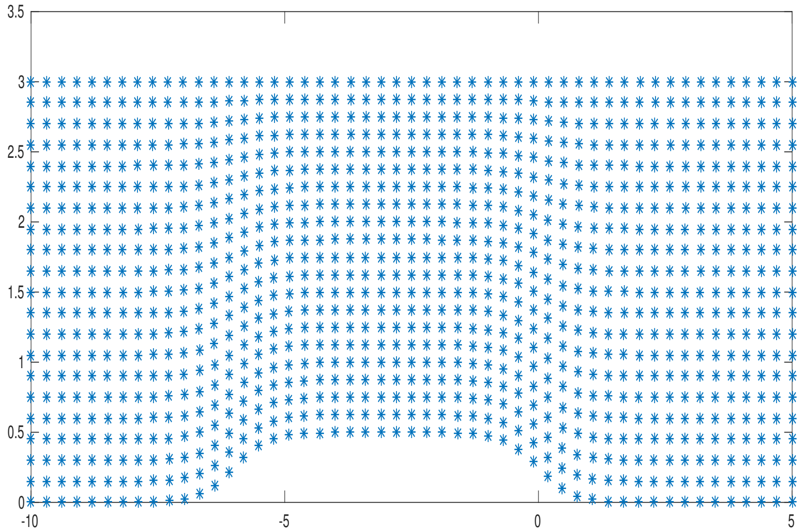

Figure 2: The generated grid using $n = 50$ and $m = 20$.

Code

Main

```
#include<iostream>

using namespace std;

#include "curvebase.hpp"
#include "xline.hpp"
#include "yline.hpp"
//#include "xquad.hpp"
#include "fxcurve.hpp"
#include "domain.hpp"


int main()
{


  fxCurve a = fxCurve(-10,5);    // f(x) from -10 to 5
  yLine b = yLine(0,3,5);        // vertical line from 0 to 3 at x=5
  xLine c = xLine(-10,5,3);      // horiz. line from -10 to 5 at y=3
  yLine d = yLine(0,3,-10);      // vertical line from 0 to 3 at x=-10


  /*
  xQuad a = xQuad(-1,-1,0,-1,0);// y=-x^2-x from x=-1 to x=0
  yLine b = yLine(0,1,0);
  xLine c = xLine(-1,0,1);
  yLine d = yLine(0,1,-1);
  */

  Domain D = Domain(a,b,c,d);    // Domain object
  D.grid_generation(50,20);      // Generate the grid on the domain

  //D.print();
  D.writeFile();                 // Write to binary file (for matlab)


  return 0;


}
```

Task 1: The Curvebase Class

```
#ifndef CURVEBASE_HPP
#define CURVEBASE_HPP

#include <cmath>
#include <iostream>



class Curvebase {

  protected:
    double a;
    double b;
    double length;


    virtual double xp(double p) = 0;     //parametrized by user
    virtual double yp(double p) = 0;     //parametrized by user
    virtual double dxp(double p) = 0;    //dx(p)/dp for arc length
    virtual double dyp(double p) = 0;    //dy(p)/dp for arc length

    double integrate(double a, double b);
    double newtonsolve(double p0, double s);

    double i2Simpson(double a, double b);
    double iSimpson(double a, double b);
    double dL(double t);                           // integrand for arc length

  public:
    Curvebase();                         //default constructor
    virtual double x(double s);          //parametrized by normalized arc length
    virtual double y(double s);          //parametrized by normalized arc length

};

#endif          // CURVEBASE_HPP
```

```
#include <cmath>
#include <iostream>
#include "curvebase.hpp"


Curvebase::Curvebase() {}; // Default constructor


/* Integrate , i2Simpson, iSimpson all taken
 * directly from project 1.
 */
inline double Curvebase::i2Simpson(double a, double b) {
  return iSimpson(a,0.5*(a+b)) + iSimpson(0.5*(a+b),b);
}

inline double Curvebase::iSimpson(double a, double b) {
  return ((b-a)/6.0)*(dL(a)+4.0*dL(0.5*(a+b)) + dL(b));
}

inline double Curvebase::dL(double p) {
  return sqrt(dxp(p)*dxp(p) + dyp(p)*dyp(p));
}

double Curvebase::integrate(double a, double b){

  double tolI = 1e-8;
  double I = 0, I1, I2, errest;
  int node = 1;

  while (true) {
    I1 = iSimpson(a,b);
    I2 = i2Simpson(a,b);
    errest = std::abs(I1-I2);
    if (errest < 15*tolI) { //if leaf
      I += I2;
      while (node % 2 != 0) { // while uneven node

        if (node == 1) {
          return I; // return if we are back at root again
```

```
        }

        node = 0.5*node;
        a = 2*a-b;
        tolI *= 2;
      }
      // First even node: go one node up - go to right child
      b = 2*b-a;
      node = node+1;
      a = 0.5*(a+b);
    } else { //if not a leaf: go to left child
      node *= 2;
      b = 0.5*(a+b);
      tolI *= 0.5;
    }
  }
  return I;
}


/* Newton solver for equation f(p) = l(p) - s*l(b)
 * input: p0 is initial guess for Newtons method.
 */
double Curvebase::newtonsolve(double p0, double s) {

  int iter = 0, maxiter = 150;
  double tolN = 1e-6;
  double err = 1.0;
  double p1,p;
  p = p0;
  while (err > tolN && iter < maxiter) {

    p1 = p - (integrate(a,p)-s*length)/dL(p);   // Newtons method
    err = fabs(p1 - p);                         // Check error
    p = p1; iter++;                             // Update
  }

  if (iter == maxiter) {                        // maxiter reached
    std::cout << "No convergence in Newton solver" << std::endl;
  }

  return p;
}


// Curve parametrized by grid coordinate
double Curvebase::x(double s){
  double p, p0;
  p0 = a + s*length;                            // Initial guess for Newtons meth.
  p = newtonsolve(p0,s);
  return xp(p);
}

// Curve parametrized by grid coordinate
double Curvebase::y(double s){
  double p, p0;
  p0 = a + s*length;                            // Initial guess for Newtons meth.
  p = newtonsolve(p0,s);
  return yp(p);
}
```

Task 2: The derived classes

```
#ifndef XLINE_HPP
#define XLINE_HPP


/* xLine: curves for lines with constant y.
 * Derived class from base class Curvebase.
 * Constructor: y0 constant y,
 *      x0, x1 interval in x: [x0, x1].
 * Overwrite integrate, xp, yp, dxp, dyp, x(s) and y(s).
 */

class xLine: public Curvebase{
  public:
    xLine(double x0, double x1, double y0);     // Constructor
    ~xLine();                                   // Destructor
    double x(double s);                         // Grid coordinate s
    double y(double s);                         // Grid coordinate s
```

```
  protected:
    double yConst;
    double xp(double p);
    double yp(double p);
    double dxp(double p);
    double dyp(double p);
    double integrate(double a, double b);        // Arc length
};

#endif            // XLINE_HPP
```

```
#include "curvebase.hpp"
#include "xline.hpp"

// Constructor
xLine::xLine(double xi, double xf, double y0) {
  a = xi;
  b = xf;
  yConst = y0;
  length = xf-xi;
}

// Destructor
xLine::~xLine() {};

// Overwrite integrate (arc length = interval length)
double xLine::integrate(double a, double b){
  return (b-a);
};

// Overwrite y(s) and x(s) in normalized coordinates
double xLine::y(double s){
    return yConst;               // Constant y for a horizontal line
};
double xLine::x(double s){
    return a+s*length;           // Simple formula for horizontal line
};

// Curve parametrized by user parameter
double xLine::xp(double p) { return p; }
double xLine::yp(double p) { return yConst; }

// Derivatives w.r.t. user parameter
double xLine::dxp(double p) { return 1; }
double xLine::dyp(double p) {return 0; }
```

```
#ifndef YLINE_HPP
#define YLINE_HPP


/* yLine: curves for lines with constant x.
 * Derived class from base class Curvebase.
 * Constructor: x0 is constant x,
 *       y0, y1 interval in y: [y0,y1].
 * Overwrite integrate, xp, yp, dxp, dyp, x(s) and y(s)
 */

class yLine: public Curvebase{
  public:
    yLine(double y0, double y1, double x0);      // Constructor
    ~yLine();                                    // Destructor
    double x(double s);                          // Grid coordinate s
    double y(double s);                          // Grid coordinate s

  protected:
    double xC;
    double xp(double p);
    double yp(double p);
    double dxp(double p);
    double dyp(double p);
    double integrate(double a, double b);        //Arc length
};

#endif            // YLINE_HPP
```

```
#include "curvebase.hpp"
#include "yline.hpp"

// Constructor
yLine::yLine(double y0, double y1, double x0) {
  a = y0;
  b = y1;
  xC = x0;
  length = y1 - y0;
}

// Destructor
yLine::~yLine() {}

// Overwrite integrate (arc length = interval length)
double yLine::integrate(double a, double b){
  return (b-a);
};

// Overwrite y(s) and x(s)
double yLine::x(double s){
  return xC;                    // Constant x for vertical line
};
double yLine::y(double s){
  return a+s*length;            // Simple formula for vertical line
};

// Curve parametrized in user coordinate
double yLine::xp(double p) { return xC; }
double yLine::yp(double p) { return p; }

// Derivatives w.r.t user parameter
double yLine::dxp(double p) { return 0; }
double yLine::dyp(double p) { return 1; }
```

```
#ifndef FXCURVE_HPP
#define FXCURVE_HPP

/* fxCurve: Derived class from base class Curvebase.
 * Constructor: interval length in x: [x0,x1].
 */

class fxCurve: public Curvebase{
  public:
    fxCurve(double xx0, double xx1);     // Constructor
    ~fxCurve();                          // Destructor

  protected:
    double xp(double p);
    double yp(double p);
    double dxp(double p);
    double dyp(double p);
};


#endif          // FXCURVE_HPP
```

```
#include <cmath>          // for exp in xp, yp, dxp, dyp

#include "curvebase.hpp"
#include "fxcurve.hpp"

// Constructor
fxCurve::fxCurve(double xx0, double xx1) {
  a = xx0;
  b = xx1;
  length = integrate(a,b);
}

// Destructor
fxCurve::~fxCurve() {}

// Curve parametrized in user parameter p
double fxCurve::xp(double p) { return p; }
double fxCurve::yp(double p) {
  if (p < -3.0) {
    return 0.5/(1.0 + exp(-3.0*(p + 6.0)));
```

```
  } else {
    return 0.5/(1.0 + exp(3.0*p));
  }
}

// Derivatives w.r.t. the user parameter p
double fxCurve::dxp(double p) { return 1.0; }
double fxCurve::dyp(double p) {
  if (p < -3.0) {
    //return 6.0*exp(-3.0*(p+6))*yp(p)*yp(p);
    return 1.5*exp(3.0*(p+6))/(1.0 + 2.0*exp(3.0*(p + 6.0)) + exp(6.0*(p+6.0)));
  } else {
    //return -6.0*exp(3.0*p)*yp(p)*yp(p);
    return -1.5*exp(3.0*p)/(1.0 + 2.0*exp(3.0*p) + exp(6.0*p));
  }
}
```

Task 3 and 4: The Domain Class

```
#ifndef DOMAIN_HPP
#define DOMAIN_HPP

#include "curvebase.hpp"

class Domain {
  private:
    Curvebase * sides[4];               // Pointers to curves of the 4 sides
    int m_, n_;                         // # of grid points in x and y
    double *x_,*y_;                     // Arrays for coordinates in grid
    bool cornersOk;

    double phi1(double t);              // Linear interpolation functions
    double phi2(double t);

  public:
    // CONSTRUCTOR
    Domain(Curvebase& s1, Curvebase& s2, Curvebase& s3, Curvebase& s4);

    // DESTRUCTOR
    ~Domain();

    // FUNCTIONS
    void grid_generation(int n, int m); // Generates the grid (x_ and y_)
    void print();                       // Print points of grid to console
    void writeFile();                   // Write points to .bin-file (use matlab to view)
    bool checkCorners();                // Check if corners are connected
};

#endif //DOMAIN_HPP
```

```
#include <cstdio>        // for writeFile()
#include <iostream>
#include <cmath>         // for fabs

#include "domain.hpp"
#include "curvebase.hpp"

/*
 * .cpp-file for class domain. See also domain.hpp.
 */


// CONSTRUCTOR ----------------------------------------------------------------
Domain::Domain(Curvebase& s1, Curvebase& s2, Curvebase& s3, Curvebase& s4) {
  sides[0] = &s1;
  sides[1] = &s2;
  sides[2] = &s3;
  sides[3] = &s4;

  cornersOk = checkCorners();             // Indicator for corners connected
  if (cornersOk == false) {
    sides[0] = sides[1] = sides[2] = sides[3] = NULL;
  }

  m_ = n_ = 0;                            // Number of grid points
  x_ = y_ = NULL;                         // Arrays for grid coordinates
}

// DESTRUCTOR -----------------------------------------------------------------
```

```
Domain::~Domain() {
  if (m_ > 0) {          // Could as well check if n_>0, since both
    delete [] x_;        // need to be positive to generate the grid
    delete [] y_;
  }
}



// MEMBER FUNCTIONS ------------------------------------------------------------

// Linear interpolation functions
double Domain::phi1(double t) {
  return t;                              // phi1(0) = 0, phi1(1) = 1
}
double Domain::phi2(double t) {
  return 1.0-t;                          // phi2(0) = 1, phi2(1) = 0
}

// Generates the grid and sets it to
void Domain::grid_generation(int n, int m) {
  if ((n < 1) || (m < 1)) {
    // Need n and m > 0 to generate any grid. Else:
    std::cout << "Warning: Non-positive grid size." << std::endl;
    std::cout << "No grid generated" << std::endl;
    return;                              // No grid is generated
  } else if (cornersOk == false) {
    // Dont generate grid if corners are disconnected
    std::cout << "No grid generated (corner disconnected)" << std::endl;
    return;                              // No grid is generated
  }

  if (n != 0) {                          // Reset the arrays
    delete[] x_;
    delete[] y_;
  }

  n_ = n;
  m_ = m;

  /* The sides' coordinates are computed once only, i.e. there is
   * 4*(n+1)+4*(m+1) calls to x(s) and y(s). If instead, one would
   * call x(s) and y(s) for each of the grid points there would be
   * 16*(n+1)*(m+1) calls. Consider memory if n,m are large.
   */

  double *xLo,*xRi,*xTo,*xLe,*yLo,*yRi,*yTo,*yLe;

  xLo = new double[n_+1];        // Lower boundary x-coords
  xRi = new double[m_+1];        // Right boundary
  xTo = new double[n_+1];        // Top boundary
  xLe = new double[m_+1];        // Left boundary

  yLo = new double[n_+1];        // same for the y-coords
  yRi = new double[m_+1];
  yTo = new double[n_+1];
  yLe = new double[m_+1];

  double h1= 1.0/n; double h2 = 1.0/m;   // Step sizes

  for (int i=0; i <= n_; i++) {          // Loop the normalized coordinate for x
    xLo[i] = sides[0]->x(i*h1);
    xTo[i] = sides[2]->x(i*h1);

    yLo[i] = sides[0]->y(i*h1);
    yTo[i] = sides[2]->y(i*h1);

  }
  for (int j=0; j <= m_; j++) {          // Loop the normalized coordinate for y
    xRi[j] = sides[1]->x(j*h2);
    xLe[j] = sides[3]->x(j*h2);

    yRi[j] = sides[1]->y(j*h2);
    yLe[j] = sides[3]->y(j*h2);
  }

  x_ = new double[(n_+1)*(m_+1)];        // x-coordinates for the entire grid
  y_ = new double[(n_+1)*(m_+1)];        // y-coordinates for the same

  for (int i = 0; i <= n_; i++) {
```

```
    for (int j = 0; j <= m_; j++) {

      x_[j+i*(m_+1)] =
        phi2(i*h1)*xLe[j]                  // left side
        + phi1(i*h1)*xRi[j]                // right side
        + phi2(j*h2)*xLo[i]                // bottom side
        + phi1(j*h2)*xTo[i]                // top side
        - phi2(i*h1)*phi2(j*h2)*xLo[0]     // lower left
        - phi1(i*h1)*phi2(j*h2)*xLo[n_]    // lower right
        - phi2(i*h1)*phi1(j*h2)*xTo[0]     // top left
        - phi1(i*h1)*phi1(j*h2)*xTo[n_];   // top right

      y_[j+i*(m_+1)] =
        phi2(i*h1)*yLe[j]                  // equivalent to x above
        + phi1(i*h1)*yRi[j]
        + phi2(j*h2)*yLo[i]
        + phi1(j*h2)*yTo[i]
        - phi2(i*h1)*phi2(j*h2)*yLo[0]
        - phi1(i*h1)*phi2(j*h2)*yLo[n_]
        - phi2(i*h1)*phi1(j*h2)*yTo[0]
        - phi1(i*h1)*phi1(j*h2)*yTo[n_];
    }
  }

  delete[] xLo;
  delete[] xRi;
  delete[] xTo;
  delete[] xLe;

  delete[] yLo;
  delete[] yRi;
  delete[] yTo;
  delete[] yLe;
}


// Function to check if the boundaries are connected (corners)
bool Domain::checkCorners() {
  if (fabs(sides[0]->x(1) - sides[1]->x(0)) > 1e-4 ||
      fabs(sides[0]->y(1) - sides[1]->y(0)) > 1e-4) {
    std::cout << "Low-Right corner disconnected" << std::endl;
    return false;
  }
  if (fabs(sides[1]->x(1) - sides[2]->x(1)) > 1e-4 ||
      fabs(sides[1]->y(1) - sides[2]->y(1)) > 1e-4) {
    std::cout << "Top-Right corner disconnected" << std::endl;
    return false;
  }
  if (fabs(sides[2]->x(0) - sides[3]->x(1) > 1e-4) ||
      fabs(sides[2]->y(0) - sides[3]->y(1) > 1e-4)) {
    std::cout << "Top-Left corner disconnected" << std::endl;
    return false;
  }
  if (fabs(sides[3]->x(0) - sides[0]->x(0)) > 1e-4 ||
      fabs(sides[3]->y(0) - sides[0]->y(0)) > 1e-4) {
    std::cout << "Low-Left corner disconnected" << std::endl;
    return false;
  }
  return true;
}



// Print (for testing) the grid coordinates: Careful if n,m are large.
void Domain::print() {
  if (n_ < 1 || m_ < 1) {
    std::cout << "No grid to print" << std::endl;
    return;
  }
  for (int i = 0; i < (n_+1)*(m_+1); i++) {
    std::cout << "[" << x_[i] << "," << y_[i] << "]" << std::endl;
  }
}

// Write the grid to an external file to enable visualization in e.g. matlab.
void Domain::writeFile(){
  if (n_ < 1 || m_ < 1) {
    std::cout << "No grid available for writeFile()" << std::endl;
    return;
  }
```

```
  FILE *fp;
  fp =fopen("outfile.bin","wb");
  fwrite(&n_,sizeof(int),1,fp);
  fwrite(&m_,sizeof(int),1,fp);
  fwrite(x_,sizeof(double),(n_+1)*(m_+1),fp);
  fwrite(y_,sizeof(double),(n_+1)*(m_+1),fp);
  fclose(fp);
}
```