Project 4, SF2565

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

January 8, 2017

Task 1: Redesigning the Domain class

The domain class was taken directly from project 3, with the following modifications.

- Functions `xsize()`, `ysize()` and `gridValid()` was added.
- The class uses `shared_ptr` to the boundary curves.
- The `writeFile()` function was changed to `writeFile(std::string fileName)` to be able to save the different grid functions with different filenames.

The `Curvebase` class and its derived classes are almost identical to those used in project 3. We have attempted to optimize the code slightly, for example by using inlining in the classes `xLine` and `yLine`.

Task 2: The Gfctn class

The class for gridfunctions `class Gfctn` has the following data members.

- A matrix `u` to store the grid function values.
- A `shared_ptr` to a `Domain` object called `grid`.

The class has a the following constructors.

- `Gfctn(shared_ptr<Domain> grid)` which initializes the grid function with the matrix `u` being the zero matrix.
- `Gfctn(const Gfctn& U)`, a copy constructor.

Overloaded operators are provided for adding and multiplying grid functions (`+` and `*`). The following member functions are implemented.

- `void setFunction(fctnPtr f)` which sets the gridfunction values to those defined by the function `f`. This function needs to take `Point` objects as argument.
- `void print()` which prints the matrix `u`. This is useful for testing on small grids only.
- `void writeFile(std::string fileName) const` which saves the grid function values to a binary file, `fileName.bin`.

In addition to those functions listed above, the class has functions to compute the approximations to $\frac{\partial u}{\partial x}$, $\frac{\partial u}{\partial y}$, $\frac{\partial^2 u}{\partial x^2}$ and $\frac{\partial^2 u}{\partial y^2}$. Finally the class has a function for computing the Laplacian of the grid function, $\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$. These functions are

- `Gfctn D0x() const`
- `Gfctn D0y() const`

- `Gfctn DD0x() const`
- `Gfctn DD0y() const`
- `Gfctn laplace() const`

Task 3: Results

We investigate the class using the function

$$u(x, y) = \sin(x^2/10^2)\cos(x/10) + y$$

Figure 1 shows the function on the domain from project 3.

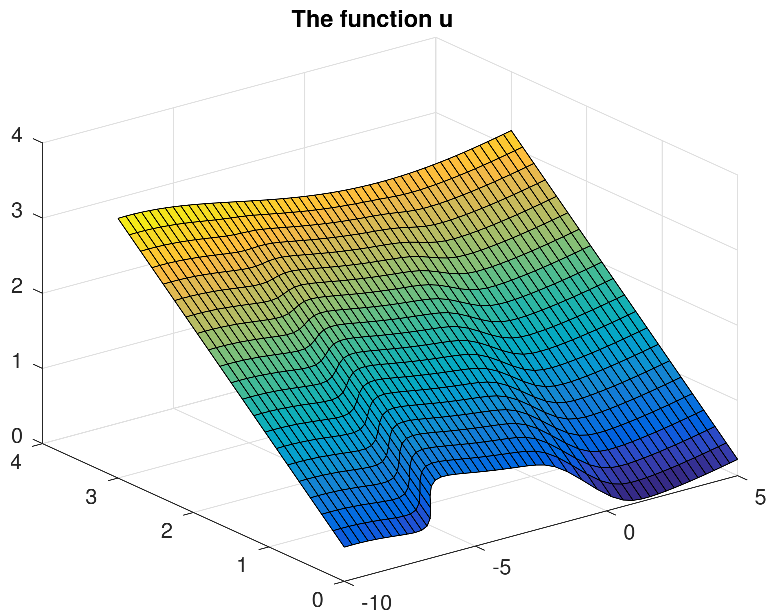

Figure 1: TODO caption

Derivative w.r.t. x

Figures 2 and 3 shows the derivative w.r.t. x and the result from the implementation. Figure 4 show the difference between the true derivative w.r.t x and the implementation.

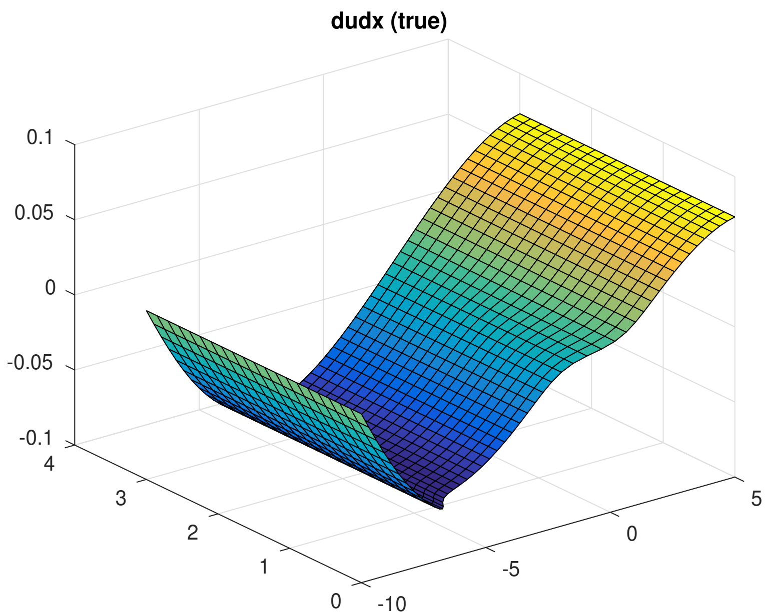

Figure 2: The true derivative $\frac{\partial u}{\partial x}$.

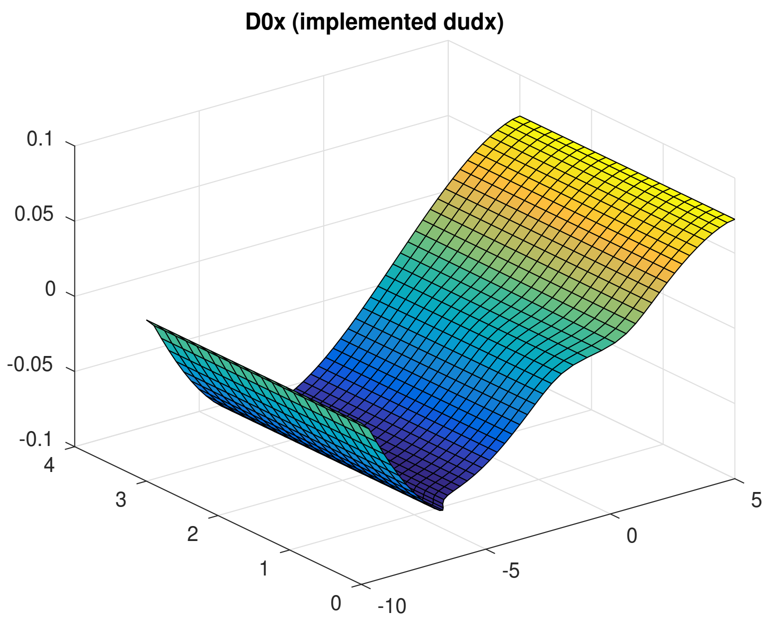

Figure 3: The result of the implementation of the derivative $\frac{\partial u}{\partial x}$.

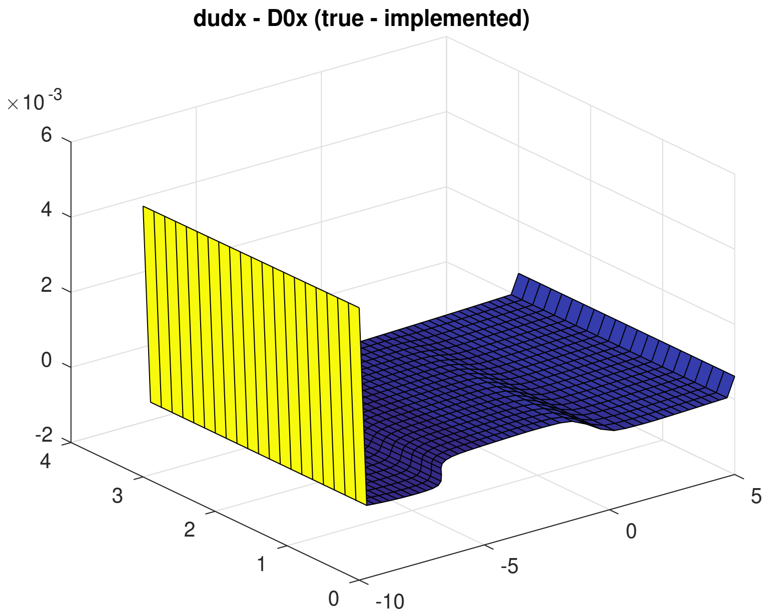

Figure 4: The difference of the true and implemented x-derivatives. Since we used one-sided differences on the boundary, the accuracy of the implementation is lower at the boundary.

Derivative w.r.t. y

Figure 5 shows the true derivative w.r.t. y while figures 6 and 7 show the implemented derivative and the difference of the true and implemented derivatives.

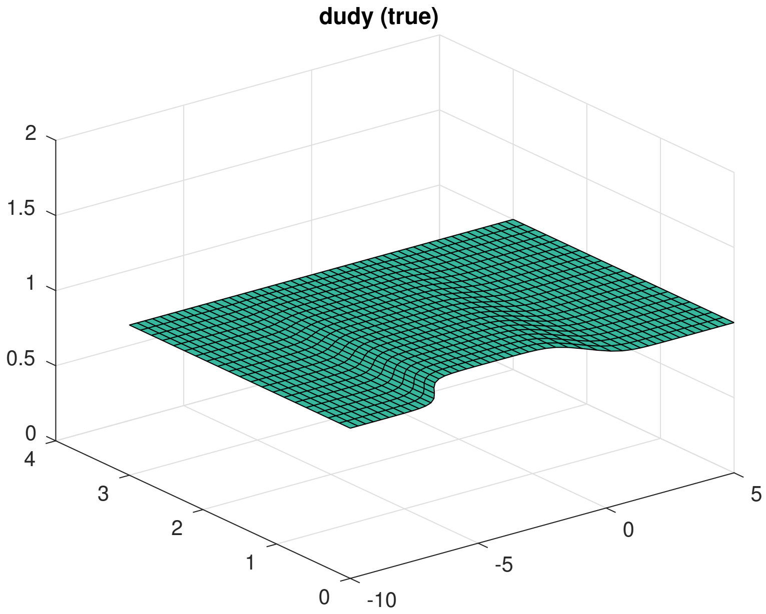

Figure 5: The true derivative $\frac{\partial u}{\partial y}$ is constant and equal to 1.

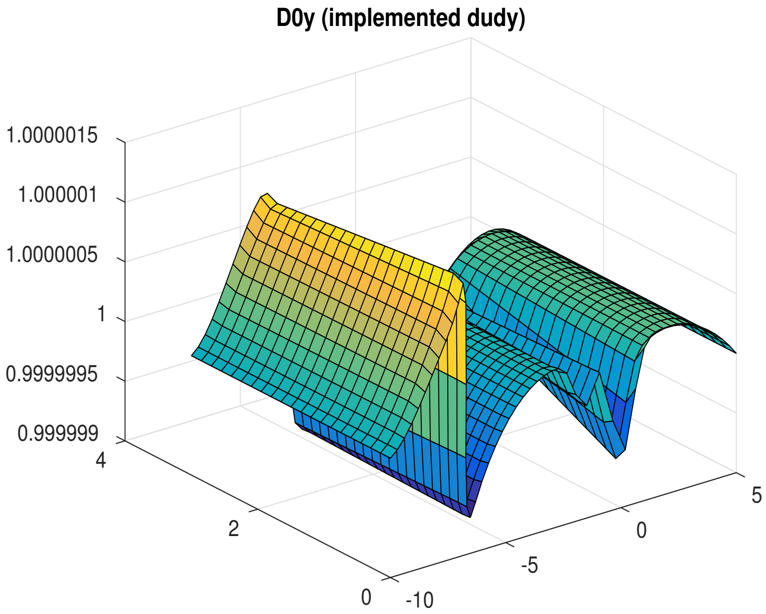

Figure 6: The implemented y-derivative. It is almost constantly equal to 1.

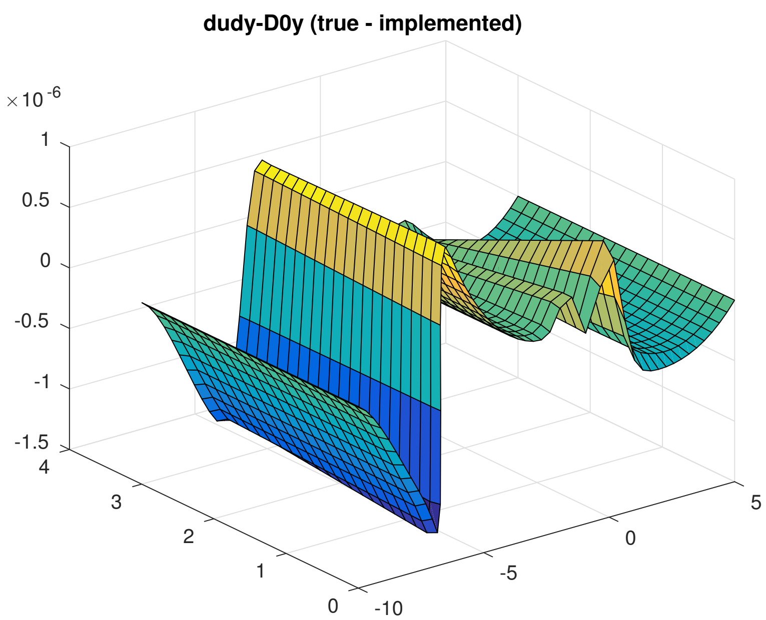

Figure 7: Difference between true and implemented derivatives.

Code

Main

```
// file: testDomain.cpp

#include <iostream>
#include <memory>
#include <cmath>                    // for sin and cos
#include "xline.hpp"
#include "yline.hpp"
#include "fxcurve.hpp"
#include "domain.hpp"
#include "gfctn.hpp"

using namespace std;

// function for testing the classes (as specified in lab instructions)
inline double f(Point p) {
  return sin((p.X()*p.X()*0.01))*cos(p.X()*0.1) + p.Y();
}

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

  /*
  shared_ptr<xQuad> a = make_shared<xQuad>(-.25,0,.25,-1,1);
  shared_ptr<yLine> b = make_shared<yLine>(0,1,1);
  shared_ptr<xLine> c = make_shared<xLine>(-1,1,1);
  shared_ptr<yLine> d = make_shared<yLine>(0,1,-1);

  */

  shared_ptr<fxCurve> a = make_shared<fxCurve>(-10.0,5.0);
  shared_ptr<yLine> b = make_shared<yLine>(0.0,3.0,5.0);
  shared_ptr<xLine> c = make_shared<xLine>(-10.0,5.0,3.0);
  shared_ptr<yLine> d = make_shared<yLine>(0.0,3.0,-10.0);




  shared_ptr<Domain> grid = make_shared<Domain>(a,b,c,d);
  grid->grid_generation(50,20);
  cout << "x-size = " << grid->xsize() << endl;
  cout << "y-size = " << grid->ysize() << endl;
  grid->writeFile("gridOut.bin");
  Gfctn U = Gfctn(grid);
  U.setFunction(&f);
  //U.print();


  Gfctn DxU = U.D0x();
  cout << "derivative x: " << endl;
  DxU.writeFile("DxOut.bin");

  Gfctn DyU = U.D0y();
  cout << "derivative y: " << endl;
  DyU.writeFile("DyOut.bin");


  Gfctn Laplace = U.laplace();
  cout << "Laplace: " << endl;
  Laplace.writeFile("laplaceOut.bin");

  return 0;      // Great Success
}
```

The Curvebase Class

```
// file: curvebase.hpp

#ifndef CURVEBASE_HPP
#define CURVEBASE_HPP

#include <cmath>
#include <iostream>


class Curvebase {
```

```
private:
  double newtonsolve(double p0, double s) const;

  double i2Simpson(double a, double b) const;

  double iSimpson(double a, double b) const;

  double dL(double t) const;       // integrand for arc length


protected:
  double a;
  double b;
  double length;

  // Pure virtual, ("= 0"), derived classes must implement:
  virtual double xp(double p) const = 0;  //parametrized by user
  virtual double yp(double p) const = 0;  //parametrized by user
  virtual double dxp(double p) const = 0;  //dx(p)/dp for arc length
  virtual double dyp(double p) const = 0;  //dy(p)/dp for arc length

  double integrate(double a, double b) const;  // TODO private?

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

  // TODO from slides 6 F_Inheritance.pdf page 30:
  // the destructor of abstract base class should always be virtual
  // virtual ~Curvebase();

};

#endif            // CURVEBASE_HPP
```

```
// file: Curvebase.cpp

#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) const {
  return iSimpson(a,0.5*(a+b)) + iSimpson(0.5*(a+b),b);
}

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

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

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

  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;
    }
  }
}


/* 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) const{

  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) const{
  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) const{
  double p, p0;
  p0 = a + s*length;                             // Initial guess for Newtons meth.
  p = newtonsolve(p0,s);
  return yp(p);
}
```

The derived classes from the Curvebase Class

`yline`).

```
// file: xline.hpp

#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).
 */

#include "curvebase.hpp"

class xLine: public Curvebase{
  public:
    xLine(double x0, double x1, double y0)      // Constructor
    {
```

```
      a = x0;
      b = x1;
      yConst = y0;
      length = x1 - x0;
    }
    ~xLine() {}                               // Destructor

    // Overwrite x(s) and y(s) in normalized coordinates
    double x(double s) const { return a+s*length; }
    double y(double s) const { return yConst; }

  protected:
    double yConst;

    // user parametrizations
    double xp(double p) const { return p; }
    double yp(double p) const { return yConst; }
    double dxp(double p) const { return 1.0; }
    double dyp(double p) const { return 0.0; }

    // Arc length
    double integrate(double a, double b) const { return b-a; }
};

#endif          // XLINE_HPP
```

```
#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)
 */

#include "curvebase.hpp"

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

    // Overwrite x(s) and y(s) in normalized coordinates
    double x(double s) const { return xC; }
    double y(double s) const { return a+s*length; }

  protected:
    double xC;

    // user parametrizations
    double xp(double p) const { return xC; }
    double yp(double p) const { return p; }
    double dxp(double p) const { return 0.0; }
    double dyp(double p) const { return 1.0; }

    // Arc length
    double integrate(double a, double b) const { return b-a; }
};


#endif          // YLINE_HPP
```

```
#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) const;
    double yp(double p) const;
    double dxp(double p) const;
    double dyp(double p) const;
};


#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) const { return p; }
double fxCurve::yp(double p) const {
  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) const { return 1.0; }
double fxCurve::dyp(double p) const {
  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));
  }
}
```

The Domain Class

```
// file: domain.hpp

#ifndef DOMAIN_HPP
#define DOMAIN_HPP

#include <memory>                        // for shared_ptr (use -std=c++11)
#include "curvebase.hpp"
#include "point.hpp"

using namespace std;

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

    inline double phi1(double t) const {return t;};           // Linear interpolation functions
    inline double phi2(double t) const {return 1.0-t;};

  public:
    // CONSTRUCTOR
```

```
    //Domain(Curvebase& s1, Curvebase& s2, Curvebase& s3, Curvebase& s4);
    Domain(shared_ptr<Curvebase> s1,     // TODO Ã¤r det hÃ¤r rÃ¤tt??
        shared_ptr<Curvebase> s2,
        shared_ptr<Curvebase> s3,
        shared_ptr<Curvebase> s4);

    // TODO move constructor? do we need it?

    // DESTRUCTOR
    ~Domain();

    // COPY-ASSIGN (?) TODO (slide 8 F_Move p.6)
    // Domain& operator=(const Domain& D); //TODO ska man ha med "D" hÃ¤r?

    Point operator()(int i, int j) const;

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

    // new functions for pro4:
    inline int xsize() const {return n_;};
    inline int ysize() const {return m_;};
    bool gridValid() const;
    // TODO Point operator()(int i, int j);

};

#endif //DOMAIN_HPP
```

```
// file: domain.cpp

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

#include "domain.hpp"
//#include "curvebase.hpp"
//#include "point.hpp"

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


using namespace std;

// CONSTRUCTOR -----------------------------------------------------------
Domain::Domain(shared_ptr<Curvebase> s1,
    shared_ptr<Curvebase> s2,
    shared_ptr<Curvebase> s3,
    shared_ptr<Curvebase> s4): n_(0), m_(0), x_(nullptr), y_(nullptr) {

  sides[0] = s1;
  sides[1] = s2;
  sides[2] = s3;
  sides[3] = s4;

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

}

// 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_;
  }
}



Point Domain:: operator()(int i, int j) const
{
```

```
  if (i < 0 || i > n_ || j < 0 || j > m_) {
    cout << "invalid index ij" << endl;
    exit(1);
  }
  //cout << "ij = " << i << "," << j << endl; //TODO remove (was for testing)
  int ind = j+i*(m_+1);
  return Point(x_[ind],y_[ind]);
}




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

// 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) {
    // 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.
   * TODO comment on memory...
   */

  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;
}

// Print (for testing) the grid coordinates: Careful if n,m are large.
void Domain::print() const {
  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(std::string fileName) const{
  if (n_ < 1 || m_ < 1) {
    std::cout << "No grid available for writeFile()" << std::endl;
    return;
  }
  FILE *fp;
  fp =fopen(fileName.c_str(),"wb");
  if (fp == nullptr) {
      std::cout << "Error opening file to write to" << std::endl;
  }
  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);
}

// Function to check if the boundaries are connected (corners)
bool Domain::checkCorners() const {
  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;
}

// new functions for pro4:

bool Domain::gridValid () const
{
  if (m_ != 0 && checkCorners()) {
    //std::cout << "grid valid!" << std::endl;
    return true;
  } else {
    std::cout << "grid NOT valid!" << std::endl;
    return false;
  }
}
```

The Gfctn Class

```
// file: gfctn.hpp

#ifndef GFCTN_HPP
#define GFCTN_HPP

#include <memory>        // for shared_ptr (use -std=c++11)
#include "matrix.hpp"
#include "domain.hpp"

// from slides "Implementation of Grid Functions"

typedef double (*fctnPtr)(Point);

class Gfctn
{
  private:
    Matrix u;
    shared_ptr<Domain> grid;

  public:
    // CONSTRUCTORS
    Gfctn(shared_ptr<Domain> grid_);
    Gfctn(const Gfctn& U);

    // OPERATORS
    Gfctn& operator=(const Gfctn& U);    // copy assignment
    Gfctn& operator=(Gfctn&& U) noexcept; // move assignment

    Gfctn operator+(const Gfctn& U) const;
    Gfctn operator*(const Gfctn& U) const;

    // MEMBER FUNCTIONS
    void setFunction(const fctnPtr f);  // set grid function values
    inline void writeFile(std::string fileName) const {u.writeFile(fileName);} //write to binary file
    Gfctn D0x() const;                  // du/dx
    Gfctn D0y() const;                  // du/dy
    Gfctn laplace() const;              // d2u/dx2 + d2u/dy2


// etc
};

#endif // GFCTN_HPP
```

```
// file: gfctn.cpp

#include <iostream>
#include "gfctn.hpp"

/* Source file for Gfctn class.
 * See gfctn.hpp for declarations.
 *
 * TODO more comments
 */

// Constructors ---------------------------------------------------------------

Gfctn::Gfctn(shared_ptr<Domain> grid_)
```

```
  : u(grid_->xsize() + 1, grid_->ysize() + 1), grid(grid_) {}

Gfctn::Gfctn(const Gfctn &U)
  : u(U.u), grid(U.grid) {}

// Destructor ----------------------------------------------

/*
Gfctn::~Gfctn()
{
  // TODO implement destructor
}
*/

// Operator overloadings -----------------------------------

// Copy assignment
Gfctn &Gfctn::operator=(const Gfctn &U) {
  u = U.u;
  grid = U.grid;
  return *this;
}

// Move assignment
Gfctn &Gfctn::operator=(Gfctn &&U) noexcept {
  u = U.u;
  grid = U.grid;
  U.u = Matrix();
  U.grid = nullptr;
  return *this;
}

Gfctn Gfctn::operator+(const Gfctn &U) const {
  if (grid == U.grid) { // Defined on same grid?
    Gfctn tmp = Gfctn(grid);
    tmp.u = u + U.u;  // Matrix operator +()
    return tmp;
  } else {
    std::cout << "error: different grids" << std::endl;
    exit(1);
  }
}

Gfctn Gfctn::operator*(const Gfctn &U) const {
  if (grid == U.grid) {
    Gfctn tmp = Gfctn(grid);
    for (int j = 0; j < grid->ysize(); j++) {
      for (int i = 0; i < grid->xsize(); i++) {
        tmp.u(i, j) = u.get(i, j) * U.u.get(i, j);
      }
    }
    return tmp;
  } else {
    std::cout << "error: different grids (*)" << std::endl;
    exit(1);
  }
}

// Member functions ----------------------------------------

/* setFunction
 * Computes the value of the function f in all grid points and puts in the matrix u
 *
 */
void Gfctn::setFunction(const fctnPtr f)
{
  for (int j = 0; j <= grid->ysize(); j++) {
    for (int i = 0; i <= grid->xsize(); i++) {
      u(i, j) = f((*grid)(i, j));
      //cout << (*grid)(i,j) << endl;
    }
  }
}



/* du/dx of grid function u
 * usage: Gfctn DxU = U.D0x();
 * Implementation of derivative from p.13 in slide F_PDEs
 */
```

```
Gfctn Gfctn::D0x() const {
  Gfctn tmp(grid);
  if (grid->gridValid()) {
    double xi, xj, yi, yj, ui, uj;
    double h1 = 1.0 / grid->xsize();
    double h2 = 1.0 / grid->ysize();

    for (int j = 0; j <= grid->ysize(); j++) {
      for (int i = 0; i <= grid->xsize(); i++) { //start at i=1, end at i=n-1
        if (i == 0) {
          xi = ((*grid)(i + 1, j).X() - (*grid)(i, j).X()) / h1;
          yi = ((*grid)(i + 1, j).Y() - (*grid)(i, j).Y()) / h1;
          ui = (u.get(i + 1, j) - u.get(i, j)) / h1;
        } else if (i == grid->xsize()) {
          xi = ((*grid)(i, j).X() - (*grid)(i - 1, j).X()) / h1;
          yi = ((*grid)(i, j).Y() - (*grid)(i - 1, j).Y()) / h1;
          ui = (u.get(i, j) - u.get(i - 1, j)) / h1;
        } else {
          xi = ((*grid)(i + 1, j).X() - (*grid)(i - 1, j).X()) / (2.0 * h1);
          yi = ((*grid)(i + 1, j).Y() - (*grid)(i - 1, j).Y()) / (2.0 * h1);
          ui = (u.get(i + 1, j) - u.get(i - 1, j)) / (2.0 * h1);
        }
        if (j == 0) {
          xj = ((*grid)(i, j + 1).X() - (*grid)(i, j).X()) / h2;
          yj = ((*grid)(i, j + 1).Y() - (*grid)(i, j).Y()) / h2;
          uj = (u.get(i, j + 1) - u.get(i, j)) / h2;
        } else if (j == grid->ysize()) {
          xj = ((*grid)(i, j).X() - (*grid)(i, j - 1).X()) / h2;
          yj = ((*grid)(i, j).Y() - (*grid)(i, j - 1).Y()) / h2;
          uj = (u.get(i, j) - u.get(i, j - 1)) / h2;
        } else {
          xj = ((*grid)(i, j + 1).X() - (*grid)(i, j - 1).X()) / (2.0 * h2);
          yj = ((*grid)(i, j + 1).Y() - (*grid)(i, j - 1).Y()) / (2.0 * h2);
          uj = (u.get(i, j + 1) - u.get(i, j - 1)) / (2.0 * h2);
        }
        tmp.u(i, j) = (ui * yj - uj * yi) / (xi * yj - yi * xj);
      }
    }
  } else {
    cout << "grid invalid in D0x" << endl;

  }
  return tmp;
}


/* du/dy of grid function u
 * Analogous to above
 */
Gfctn Gfctn::D0y() const {
  Gfctn tmp(grid);
  if (grid->gridValid()) {
    double xi, xj, yi, yj, ui, uj;
    double h1 = 1.0 / grid->xsize();
    double h2 = 1.0 / grid->ysize();

    for (int j = 0; j <= grid->ysize(); j++) {
      for (int i = 0; i <= grid->xsize(); i++) { //start at i=1, end at i=n-1
        if (i == 0) {
          xi = ((*grid)(i + 1, j).X() - (*grid)(i, j).X()) / h1;
          yi = ((*grid)(i + 1, j).Y() - (*grid)(i, j).Y()) / h1;
          ui = (u.get(i + 1, j) - u.get(i, j)) / h1;
        } else if (i == grid->xsize()) {
          xi = ((*grid)(i, j).X() - (*grid)(i - 1, j).X()) / h1;
          yi = ((*grid)(i, j).Y() - (*grid)(i - 1, j).Y()) / h1;
          ui = (u.get(i, j) - u.get(i - 1, j)) / h1;
        } else {
          xi = ((*grid)(i + 1, j).X() - (*grid)(i - 1, j).X()) / (2.0 * h1);
          yi = ((*grid)(i + 1, j).Y() - (*grid)(i - 1, j).Y()) / (2.0 * h1);
          ui = (u.get(i + 1, j) - u.get(i - 1, j)) / (2.0 * h1);
        }
        if (j == 0) {
          xj = ((*grid)(i, j + 1).X() - (*grid)(i, j).X()) / h2;
          yj = ((*grid)(i, j + 1).Y() - (*grid)(i, j).Y()) / h2;
          uj = (u.get(i, j + 1) - u.get(i, j)) / h2;
        } else if (j == grid->ysize()) {
          xj = ((*grid)(i, j).X() - (*grid)(i, j - 1).X()) / h2;
          yj = ((*grid)(i, j).Y() - (*grid)(i, j - 1).Y()) / h2;
          uj = (u.get(i, j) - u.get(i, j - 1)) / h2;
        } else {
```

```
          xj = ((*grid)(i, j + 1).X() - (*grid)(i, j - 1).X()) / (2.0 * h2);
          yj = ((*grid)(i, j + 1).Y() - (*grid)(i, j - 1).Y()) / (2.0 * h2);
          uj = (u.get(i, j + 1) - u.get(i, j - 1)) / (2.0 * h2);
        }
        tmp.u(i, j) = (-ui * xj + uj * xi) / (xi * yj - yi * xj);
      }
    }
  } else {
    cout << "grid invalid in D0y" << endl;
  }
  return tmp;
}

/* Laplacian of grid function
 */
Gfctn Gfctn::laplace() const {
  Gfctn laplace = D0x().D0x() + D0y().D0y();
  return laplace;
}



// file: gfctn.cpp
```

The Matrix Class

```
// file: matrix.hpp

#ifndef MATRIX_HPP
#define MATRIX_HPP

#include <iostream>

class Matrix
{
  private:
    int m, n;    // matrix dim.
    double *a;   // matrix elements

  public:
    // Constructors and destructors
    Matrix(int n_ = 0, int m_ = 0);
    Matrix(const Matrix &M);
    ~Matrix();

    // Functions
    void fillMatrix(double b[]);
    void identity();
    void print() const;
    inline int rowSizeMatrix() const {return n;}
    inline int colSizeMatrix() const {return m;}
    void randomize();
    void writeFile(std::string fileName) const;
    inline double get(int i, int j) const {
      return a[i*m+j];                    // get element from matrix
    }

    // Operator overloadings
    Matrix &operator=(const Matrix &M);
    Matrix &operator=(Matrix &&M) noexcept;
    const Matrix &operator*=(const double d);
    const Matrix &operator+=(const Matrix &M);
    const Matrix operator+(const Matrix& M) const;
    double& operator()(int i, int j) const;
    friend std::ostream& operator<<(std::ostream& os, const Matrix& M);
};

#endif // MATRIX_HPP
```

```
// file: matrix.cpp

#include <iostream>
#include <iomanip>      // for setprecision in operator<<
#include "matrix.hpp"

/* Source file for Matrix class.
 * See matrix.hpp for declarations
 * This class implements matrix using C-style array
 * a[i+j*n] is the element on row i, col j i.e. A[i,j].
```

```
 */

using namespace std;

// Constructors ----------------------------------------------

Matrix::Matrix(int n_, int m_): m(m_), n(n_), a(nullptr)
{
  if (m*n > 0) {
    a = new double[m*n];
    fill(a,a+m*n,0.0);
  }
}

Matrix::Matrix(const Matrix &M)
{
  n = M.n;
  m = M.m;
  a = new double [m*n];
  for (int i = 0; i < n*m;  i++) {
    a[i] = M.a[i];
  }
  //cout << "matrix copy-constructor:" << this << endl;
}

// Destructor   ----------------------------------------------

Matrix::~Matrix()
{
  delete[] a;
}

// Member functions      ------------------------------------

void Matrix::fillMatrix(double b[])
{
  for (int i = 0; i < m*n; i++) {
    a[i] = b[i];
  }
}

void Matrix::identity()
{
  if (n!=m) {
    cout << "A non-square matrix can not be the identity matrix" << endl;
    return;
  }
  for (int i = 0; i < n*n; i++) {
    (i%n == i/n)? a[i] = 1: a[i] = 0;
  }
}

void Matrix::print() const
{
  cout << endl;
  if (n == 0 || m == 0) {
    cout << "[]" << endl;
    return;
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      cout << a[j + i*m] << " ";
    }
    cout << endl;
  }
}

void Matrix::randomize()
{
  if (n == 0 || m == 0) {
      cout << "empty matrix, no randomizing done" << endl;
      return;
  }
  //srand(time(0)); gives the same random number every time
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) { // TODO make single loop
      a[i*m+j] = rand()%10;
    }
  }
}
```

```
// Write the grid to an external file to enable visualization in e.g. matlab.
void Matrix::writeFile(string fileName) const{
     if (n < 1 || m < 1) {
          cout << "No matrix available for writeFile()" << endl;
          return;
     }
     FILE *fp;
     fp =fopen(fileName.c_str(),"wb");
     if (fp == nullptr) {
          cout << "Error opening file to write to" << endl;
          return;
     }
     fwrite(&n,sizeof(int),1,fp);
     fwrite(&m,sizeof(int),1,fp);
     fwrite(a,sizeof(double),n*m,fp);
     fclose(fp);
}



// Operator overloadings        -----------------------------

/* Equality operator
 * Usage: M1 = M2; where M1 and M2 are Matrix-obj.
 */
Matrix &Matrix::operator=(const Matrix &M)
{
  if (this == &M) {
    return *this;
  }
  if (n == M.n && m == M.m) {
    for (int i = 0; i < n*m; i++) {
      a[i] = M.a[i];
    }
  } else {
    if (a) {     // if initialized, delete a
      delete a;
    }
    n = M.n;
    m = M.m;
    a = new double[n*m];
    for (int i = 0; i < n*m; i++) {
      a[i] = M.a[i];
    }
  }
  return *this;
}

Matrix &Matrix::operator=(Matrix &&M) noexcept{
     if (this == &M) {
          return *this;
     }
     m = M.m;
     n = M.n;
     a = M.a;
     M.m = 0;
     M.n = 0;
     M.a = nullptr;

     return *this;
}

/* Matrix-scalar multiplication operator
 * Usage: M *= d; where M is Matrix-obj and d is double
 */
const Matrix &Matrix::operator*=(const double d)
{
  for (int i = 0; i < n*m; i++) {
    a[i] *= d;
  }
  return *this;
}

/* Matrix addition operator
 * Usage: M1 += M2
 */
const Matrix &Matrix::operator+=(const Matrix &M)
{
  if(n != M.n || m != M.m) {
```

```
    cerr << "Dimensions mismatch in sum. Exiting." << endl;
    exit(1);
  }
  for (int i = 0; i < n*m; i++) {
    a[i] += M.a[i];
  }

  return *this;
}

/* Matrix addition operator
 * Usage: A = B+C;
 */
const Matrix Matrix::operator+(const Matrix &M) const
{
  if (n != M.n || m != M.m) {
    cerr << "Dimensions mismatch in sum. Exiting" << endl;
    exit(1);
  }
  Matrix A(n,m);
  for (int i = 0; i < n*m; i++) {
    A.a[i] = a[i]+M.a[i];
  }
  return A;
}



/* Matrix element access operator
 * Usage: e = M(i,j)
 */
double& Matrix::operator()(int i, int j) const
{
  if (i < 0 || i >= n || j < 0 || j >= m) {
    cerr << "Bad index in matrix" << endl;
    exit(1);
  }
  return a[j+i*m];
}

/* Stream insertion operator
 * Usage: cout << M << endl;
 */
ostream& operator<<(ostream& os, const Matrix& M)
{
  int n = M.n;
  int m = M.m;
  os << endl;
  os << fixed << setprecision(4);
  if (n <= 0 || m <= 0) {
    os << "[]" << endl;
    return os;
  }
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (M.a[j + i*m] >= 0) {
        os << " ";
      }
      os << M.a[j + i*m] << "  ";
    }
    os << endl;
  }
  return os;
}




// matrix.cpp
```

The Point Class

```
// file: point.hpp

#ifndef POINT_HPP
#define POINT_HPP

#include <iostream>

using namespace std;
```

```
class Point
{
  private:
    double x;
    double y;

  public:

    // Constructors and destructor

    Point(double xx = 0.0, double yy = 0.0);      // constructor
    Point(const Point& Q);                        // copy constructor
    ~Point();                                     // destructor

    // Overloadings
    /*
    Point& operator=(const Point& Q);             // copy-assignment. P = Q;
    const Point operator-() const;                // P = -Q;
    const Point& operator-=(const Point& Q);      // P -= Q;
    const Point operator-(const Point& Q) const;// P-Q;

    // Other member functions
    void zero();
    void print();
    */
    double X() const; // const - obj will not change when called
    double Y() const;
    friend ostream& operator<<(ostream& os, const Point& P);
};

#endif          // POINT_HPP
```

```
// file: point.cpp


/* Source file for class Point.
 * Constructor, copy constructor, destructor.
 * Copy assignment overloading, overloadings of operators
 * for P = -Q; P -= Q; P = Q1-Q2;
 * Member access functions for cartesian coordinates.
 * Function to set point to origin.
 * Friend overloading for output stream.
 *
 * See also point.hpp for declarations
 */

#include "point.hpp"

using namespace std;


// CONSTRUCTORS AND DESTRUCTORS         ---     ---     ---     ---


// constructor using initializer list
Point::Point(double xx, double yy) :
  x(xx),
  y(yy)
{}

// Copy constructor
Point::Point(const Point& Q) :
  x(Q.x), y(Q.y)
{}

// destructor
Point::~Point()
{}


// OPERATOR OVERLOADINGS        ---     ---     ---     ---     ---

/*
// copy-assignment
Point& Point:: operator=(const Point& Q)
{
  if (this != &Q) {
```

```
    x = Q.x;
    y = Q.y;
  }
  return *this; // dereferencing
}

// negative operator i.e. P = -Q;
const Point Point:: operator-() const
{
  return Point(-x,-y);
}

// "reduce by" operator i.e. P -= Q;
const Point& Point:: operator-=(const Point& Q)
{
  x -= Q.x;
  y -= Q.y;
  return *this;
}

// minus operator i.e. P-Q
const Point Point:: operator-(const Point& Q) const
{
  return Point(x-Q.x,y-Q.y);
}

// MISC. MEMBER FUNCTIONS    ---    ---    ---    ---    ---

// set x = y = 0, i.e. point at origin.
void Point::zero()
{
  x = y = 0.0;
}

// standard member function print()
void Point::print()
{
  cout << "[" << x << "," << y << "]" << endl;
}
*/

// member access function for x
double Point::X() const
{
  return x;
}

// member access function for y
double Point::Y() const
{
  return y;
}


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

// output operator for ostream
ostream& operator<<(ostream& os, const Point& P)
{
  os << "[" << P.x << "," << P.y << "]"; //friend function
  return os;
}
```