```cpp
// see:   http://llvm.org/docs/CodingStandards.html#include-style
#include "geom.h"

#include <algorithm>
#include <cassert>
#include <fstream>
#include <iostream>
#include <list>
#include <string>
#include <vector>
using namespace std;

Mesh::Mesh(int ndim, int nvert_e, int ndof_e)
    : _nelem(0), _nvert_e(nvert_e), _ndof_e(ndof_e), _nnode(0), _ndim(ndim), _ia(0), _xc(0)
{
}

Mesh::~Mesh()
{}

void Mesh::SetValues(std::vector<double> &v, const std::function<double(double, double)> &func) const
{
    int const nnode = Nnodes();            // number of vertices in mesh
    assert( nnode == static_cast<int>(v.size()) );
    for (int k = 0; k < nnode; ++k)
    {
        v[k] = func( _xc[2 * k], _xc[2 * k + 1] );
    }
}


void Mesh::Debug() const
{
    cout << "\n ############### Debug  M E S H  ###################\n";
    cout << "\n ...............    Coordinates       ...................\n";
    for (int k = 0; k < _nnode; ++k)
    {
        cout << k << " : " ;
        for (int i = 0; i < _ndof_e; ++i )
        {
            cout << _xc[2*k+i] << "  ";
        }
        cout << endl;
    }
    cout << "\n ...............    Elements         ...................\n";
    for (int k = 0; k < _nelem; ++k)
    {
        cout << k << " : ";
        for (int i = 0; i < _ndof_e; ++i )
            cout << _ia[_ndof_e * k + i] << "  ";
        cout << endl;
    }
    return;
}

void Mesh::Write_ascii_matlab(std::string const &fname, std::vector<double> const &v) const
{
    assert(Nnodes() ==  static_cast<int>(v.size()));  // fits vector length to mesh information?

    ofstream fout(fname);                             // open file ASCII mode
    if ( !fout.is_open() )
    {
        cout << "\nFile " << fname << " has not been opened.\n\n" ;
        assert( fout.is_open() && "File not opened."  );
```

```
    }

    string const DELIMETER(" ");    // define the same delimeter as in matlab/ascii_read*.m
    int const    OFFSET(1);         // convert C-indexing to matlab

    // Write data: #nodes, #space dimensions, #elements, #vertices per element
    fout << Nnodes() << DELIMETER << Ndims() << DELIMETER << Nelems() << DELIMETER << NverticesElements() << endl;

    // Write cordinates: x_k, y_k   in seperate lines
    assert( Nnodes()*Ndims() ==  static_cast<int>(_xc.size()));
    for (int k = 0, kj = 0; k < Nnodes(); ++k)
    {
        for (int j = 0; j < Ndims(); ++j, ++kj)
        {
            fout << _xc[kj] << DELIMETER;
        }
        fout << endl;
    }

    // Write connectivity: ia_k,0, ia_k,1 etc  in seperate lines
    assert( Nelems()*NverticesElements() ==  static_cast<int>(_ia.size()));
    for (int k = 0, kj = 0; k < Nelems(); ++k)
    {
        for (int j = 0; j < NverticesElements(); ++j, ++kj)
        {
            fout << _ia[kj] + OFFSET << DELIMETER;     // C to matlab
        }
        fout << endl;
    }

    // Write vector
    for (int k = 0; k < Nnodes(); ++k)
    {
        fout << v[k] << endl;
    }

    fout.close();
    return;
}


void Mesh::Visualize(std::vector<double> const &v) const
{
    // define external command
    const string exec_m("matlab -nosplash < visualize_results.m");                 // Matlab
    //const string exec_m("octave --no-window-system --no-gui visualize_results.m"); // Octave
    //const string exec_m("flatpak run org.octave.Octave visualize_results.m"); // Octave (flatpak): desktop GH

    const string fname("uv.txt");
    Write_ascii_matlab(fname, v);

    int ierror = system(exec_m.c_str());                                 // call external command

    if (ierror != 0)
    {
        cout << endl << "Check path to Matlab/octave on your system" << endl;
    }
    cout << endl;
    return;
}

```

```cpp


// ###################################################################
Mesh_2d_3_square::Mesh_2d_3_square(int nx, int ny, int myid, int procx, int procy)
    : Mesh(2, 3, 3), // two dimensions, 3 vertices, 3 dofs
      _myid(myid), _procx(procx), _procy(procy), _neigh{{-1, -1, -1, -1}}, _color(0),
_xl(0.0), _xr(1.0), _yb(0.0), _yt(1.0), _nx(nx), _ny(ny)
{
    //void IniGeom(int const myid, int const procx, int const procy, int neigh[], int &color)
    int const ky = _myid / _procx;
    int const kx = _myid % _procy;  //    MOD(myid,procx)
    // Determine the neighbors of domain/rank myid
    _neigh[0] = (ky == 0)         ?  -1 : _myid - _procx;    //   South
    _neigh[1] = (kx == _procx - 1) ?  -1 : _myid + 1;       //   East
    _neigh[2] = (ky == _procy - 1) ?  -1 : _myid + _procx;  //   North
    _neigh[3] = (kx == 0)         ?  -1 : _myid - 1;        //   West

    _color = (kx + ky) & 1 ;

    // subdomain is part of unit square
    double const hx = 1. / _procx;
    double const hy = 1. / _procy;
    _xl = kx * hx;                     //  left
    _xr = (kx + 1) * hx;               //  right
    _yb = ky * hy;                     //  bottom
    _yt = (ky + 1) * hy;               //  top

    // Calculate coordinates
    int const nnode = (_nx + 1) * (_ny + 1); // number of nodes
    Resize_Coords(nnode, 2);                 // coordinates in 2D [nnode][ndim]
    GetCoordsInRectangle(_nx, _ny, _xl, _xr, _yb, _yt, GetCoords().data());

    // Calculate element connectivity (linear triangles)
    int const nelem = 2 * _nx * _ny;         // number of elements
    Resize_Connectivity(nelem, 3);           // connectivity matrix [nelem][3]
    GetConnectivityInRectangle(_nx, _ny, GetConnectivity().data());

    return;
}


void Mesh_2d_3_square::SetU(std::vector<double> &u) const
{
    int dx    = _nx + 1;
    for (int j = 0; j <= _ny; ++j)
    {
        int k = j * dx;
        for (int i = 0; i <= _nx; ++i, ++k)
        {
            u[k] = 0.0;
        }
    }

}

void Mesh_2d_3_square::SetF(std::vector<double> &f) const
{
    int dx    = _nx + 1;
    for (int j = 0; j <= _ny; ++j)
    {
        int k = j * dx;
        for (int i = 0; i <= _nx; ++i, ++k)
        {
            f[k] = 1.0;
        }
    }
```

```
}


std::vector<int> Mesh_2d_3_square::Index_DirichletNodes() const
{
    int const dx = 1,
              dy = _nx + 1,
              bl  = 0,
              br  = _nx,
              tl  = _ny * (_nx + 1),
              tr  = (_ny + 1) * (_nx + 1) - 1;
    int const start[4] = { bl, br, tl, bl},
                         end[4] = { br, tr, tr, tl},
                                  step[4] = { dx, dy, dx, dy};

    vector<int> idx(0);
    for (int j = 0; j < 4; j++)
    {
        if (_neigh[j] < 0)
        {
            for (int i = start[j]; i <= end[j]; i += step[j])
            {
                idx.push_back(i);        // node i is Dirichlet node
            }
        }
    }
//    remove multiple elements
    sort(idx.begin(), idx.end());                            // sort
    idx.erase( unique(idx.begin(), idx.end()), idx.end() ); // remove duplicate data

    return idx;
}

void Mesh_2d_3_square::SaveVectorP(std::string const &name, vector<double> const &u) const
{
//  construct the file name for subdomain myid
    const string tmp( std::to_string(_myid / 100) + to_string((_myid % 100) / 10) + to_string(_myid % 10) );

    const string namep = name + "." + tmp;
    ofstream ff(namep.c_str());
    ff.precision(6);
    ff.setf(ios::fixed, ios::floatfield);

    // assumes tensor product grid in unit square; rowise numbered (as generated in class constructor)
    // output is provided for tensor product grid visualization ( similar to Matlab-surf() )
    auto const &xc = GetCoords();
    int k = 0;
    for (int j = 0; j <= _ny; ++j)
    {
        for (int i = 0; i <= _nx; ++i, ++k)
            ff << xc[2 * k + 0] << "   " << xc[2 * k + 1] << "   " << u[k] << endl;
        ff << endl;
    }

    ff.close();
    return;
}

void Mesh_2d_3_square::GetCoordsInRectangle(int const nx, int const ny,
        double const xl, double const xr, double const yb, double const yt,
        double xc[])
{
    const double hx = (xr - xl) / nx,
```

```
                hy = (yt - yb) / ny;

    int k = 0;
    for (int j = 0; j <= ny; ++j)
    {
        const double y0 = yb + j * hy;
        for (int i = 0; i <= nx; ++i, k += 2)
        {
            xc[k  ] = xl + i * hx;
            xc[k + 1] = y0;
        }
    }

    return;
}

void Mesh_2d_3_square::GetConnectivityInRectangle(int const nx, int const ny, int ia
[])
{
    const int dx = nx + 1;
    int k  = 0;
    int l  = 0;
    for (int j = 0; j < ny; ++j, ++k)
    {
        for (int i = 0; i < nx; ++i, ++k)
        {
            ia[l  ] = k;
            ia[l + 1] = k + 1;
            ia[l + 2] = k + dx + 1;
            l += 3;
            ia[l  ] = k;
            ia[l + 1] = k + dx;
            ia[l + 2] = k + dx + 1;
            l += 3;
        }
    }
    return;
}

// ################### still some old code (--> MPI) ###########################


//  Copies the values of w corresponding to the boundary
//  South (ib==1), East (ib==2), North (ib==3), West (ib==4)

void GetBound(int const ib, int const nx, int const ny, double const w[], double s[]
)
{
    const int //dx = 1,
    dy = nx + 1,
    bl   = 0,
    br   = nx,
    tl   = ny * (nx + 1),
    tr   = (ny + 1) * (nx + 1) - 1;
    switch (ib)
    {
        case 1:
        {
            for (int i = bl, j = 0; i <= br; ++i, ++j)
                s[j] = w[i];
            break;
        }
        case 3:
        {
            for (int i = tl, j = 0; i <= tr; ++i, ++j)
                s[j] = w[i];
            break;
        }
```

```cpp
        case 4:
        {
            for (int i = bl, j = 0; i <= tl; i += dy, ++j)
                s[j] = w[i];
            break;
        }
        case 2:
        {
            for (int i = br, j = 0; i <= tr; i += dy, ++j)
                s[j] = w[i];
            break;
        }
        default:
        {
            cout << endl << "Wrong parameter  ib in " << __FILE__ << ":" << __LINE__ << endl;
        }
    }
    return;
}

// ----------------------------------------------------------------------------------------------------------
// Computes w:  = w + s at nodes on  the boundary
// South (ib == 1), East (ib == 2), North (ib == 3), West (ib == 4)

void AddBound(int const ib, int const nx, int const ny, double w[], double const s[])
{
    int const dy = nx + 1,
              bl  = 0,
              br  = nx,
              tl  = ny * (nx + 1),
              tr  = (ny + 1) * (nx + 1) - 1;
    switch (ib)
    {
        case 1:
        {
            for (int i = bl, j = 0; i <= br; ++i, ++j)
                w[i] += s[j];
            break;
        }
        case 3:
        {
            for (int i = tl, j = 0; i <= tr; ++i, ++j)
                w[i] += s[j];
            break;
        }
        case 4:
        {
            for (int i = bl, j = 0; i <= tl; i += dy, ++j)
                w[i] += s[j];
            break;
        }
        case 2:
        {
            for (int i = br, j = 0; i <= tr; i += dy, ++j)
                w[i] += s[j];
            break;
        }
        default:
        {
            cout << endl << "Wrong parameter  ib in " << __FILE__ << ":" << __LINE__ << endl;
        }
    }
    return;
}
```

```


// ######################################################################

Mesh_2d_3_matlab::Mesh_2d_3_matlab(string const &fname)
    : Mesh(2, 3, 3),   // two dimensions, 3 vertices, 3 dofs
      bedges(0)
{
    ifstream ifs(fname);
    if (!(ifs.is_open() && ifs.good()))
    {
        cerr << "Mesh_2d_3_matlab: Error cannot open file " << fname << endl;
        assert(ifs.is_open());
    }

    int const OFFSET(1);              // Matlab to C indexing
    cout << "ASCI file  " << fname << "  opened" << endl;

    // Read some mesh constants
    int nnode, ndim, nelem, nvert_e;
    ifs >> nnode >> ndim >> nelem >> nvert_e;
    cout << nnode << "  " << ndim << "  " << nelem << "  " << nvert_e << endl;
    assert(ndim == 2 && nvert_e == 3);

    // Allocate memory
    Resize_Coords(nnode, ndim);                 // coordinates in 2D [nnode][ndim]
    Resize_Connectivity(nelem, nvert_e);        // connectivity matrix [nelem][nvert]

    // Read ccordinates
    auto &xc = GetCoords();
    for (int k = 0; k < nnode * ndim; ++k)
    {
        ifs >> xc[k];
    }

    // Read connectivity
    auto &ia = GetConnectivity();
    for (int k = 0; k < nelem * nvert_e; ++k)
    {
        ifs >> ia[k];
        ia[k] -= OFFSET;                // Matlab to C indexing
    }

    // additional read of boundary information (only start/end point)
    int nbedges;
    ifs >> nbedges;

    bedges.resize(nbedges * 2);
    for (int k = 0; k < nbedges * 2; ++k)
    {
        ifs >> bedges[k];
        bedges[k] -= OFFSET;            // Matlab to C indexing
    }

    return;
}

// binary
//{
//ifstream ifs(fname, ios_base::in | ios_base::binary);
//if(!(ifs.is_open() && ifs.good()))
//{
//cerr << "ReadBinMatrix: Error cannot open file " << file << endl;
//assert(ifs.is_open());
//}
//cout << "ReadBinMatrix: file opened" << file << endl;

```

```

//}

// binaryIO.cpp
//void read_binMatrix(const string& file, vector<int> &cnt, vector<int> &col, vector<double> &ele)
//{

//ifstream ifs(file, ios_base::in | ios_base::binary);

//if(!(ifs.is_open() && ifs.good()))
//{
//cerr << "ReadBinMatrix: Error cannot open file " << file << endl;
//assert(ifs.is_open());
//}
//cout << "ReadBinMatrix: Opened file " << file << endl;

//int _size;

//ifs.read(reinterpret_cast<char*>(&_size), sizeof(int));   // old: ifs.read((char*)&_size, sizeof(int));
//cnt.resize(_size);
//cout << "ReadBinMatrix: cnt size: " << _size << endl;

//ifs.read((char*)&_size, sizeof(int));
//col.resize(_size);
//cout << "ReadBinMatrix: col size: " << _size << endl;

//ifs.read((char*)&_size, sizeof(int));
//ele.resize(_size);
//cout << "ReadBinMatrix: ele size: " << _size << endl;


//ifs.read((char*)cnt.data(), cnt.size() * sizeof(int));
//ifs.read((char*)col.data(), col.size() * sizeof(int));
//ifs.read((char*)ele.data(), ele.size() * sizeof(double));

//ifs.close();
//cout << "ReadBinMatrix: Finished reading matrix.." << endl;

//}


std::vector<int> Mesh_2d_3_matlab::Index_DirichletNodes() const
{
    vector<int> idx(bedges);                              // copy

    sort(idx.begin(), idx.end());                         // sort
    idx.erase( unique(idx.begin(), idx.end()), idx.end() ); // remove duplicate data

    return idx;
}
```

```cpp
#ifndef GEOM_FILE
#define GEOM_FILE
#include <array>
#include <functional>              // function; C++11
#include <string>
#include <vector>

/**
 * Basis class for finite element meshes.
 */
class Mesh
{
public:
    /**
     * Constructor initializing the members with default values.
      *
     * @param[in] ndim  space dimensions (dimension for coordinates)
      * @param[in] nvert_e  number of vertices per element (dimension for connectivity)
      * @param[in] ndof_e   degrees of freedom per element (= @p nvert_e for linear elements)
     */
    explicit Mesh(int ndim, int nvert_e = 0, int ndof_e = 0);

    /**
     * Destructor.
     *
     * See clang warning on
     * <a href="https://stackoverflow.com/questions/28786473/clang-no-out-of-line-virtual-method-definitions-pure-abstract-c-class/40550578">weak-vtables</a>.
     */
    virtual ~Mesh();

    /**
     * Number of finite elements in (sub)domain.
     * @return number of elements.
     */
    int Nelems() const
    {
        return _nelem;
    }

    /**
     * Global number of vertices for each finite element.
     * @return number of vertices per element.
     */
    int NverticesElements() const
    {
        return _nvert_e;
    }

    /**
     * Global number of degrees of freedom (dof) for each finite element.
     * @return degrees of freedom per element.
     */
    int NdofsElement() const
    {
        return _ndof_e;
    }

    /**
     * Number of vertices in mesh.
     * @return number of vertices.
     */
    int Nnodes() const
    {
        return _nnode;
    }
```

```

    /**
     * Space dimension.
     * @return number of dimensions.
     */
    int Ndims() const
    {
        return _ndim;
    }

    /**
     * (Re-)Allocates memory for the element connectivity and redefines the appropriate dimensions.
      *
     * @param[in] nelem    number of elements
      * @param[in] nvert_e  number of vertices per element
     */
    void Resize_Connectivity(int nelem, int nvert_e)
    {
        SetNelem(nelem);                // number of elements
        SetNverticesElement(nvert_e);   // vertices per element
        _ia.resize(nelem * nvert_e);
    }

    /**
     * Read connectivity information (g1,g2,g3)_i.
     * @return convectivity vector [nelems*ndofs].
     */
    const std::vector<int>  &GetConnectivity() const
    {
        return _ia;
    }

    /**
     * Access/Change connectivity information (g1,g2,g3)_i.
     * @return convectivity vector [nelems*ndofs].
     */
    std::vector<int>  &GetConnectivity()
    {
        return _ia;
    }

    /**
     * (Re-)Allocates memory for the element connectivity and redefines the appropriate dimensions.
      *
     * @param[in] nnodes    number of nodes
      * @param[in] ndim      space dimension
     */
    void Resize_Coords(int nnodes, int ndim)
    {
        SetNnode(nnodes);       // number of nodes
        SetNdim(ndim);          // space dimension
        _xc.resize(nnodes * ndim);
    }

    /**
     * Read coordinates of vertices (x,y)_i.
     * @return coordinates vector [nnodes*2].
     */
    const std::vector<double> &GetCoords() const
    {
        return _xc;
    }

    /**
     * Access/Change coordinates of vertices (x,y)_i.
     * @return coordinates vector [nnodes*2].
```

```
     */
    std::vector<double> &GetCoords()
    {
        return _xc;
    }

    /**
     * Calculate values in vector @p v via function @p func(x,y)
     * @param[in] v     vector
      * @param[in] func  function of (x,y) returning a double value.
     */
    void SetValues(std::vector<double> &v, const std::function<double(double, double)> &func) const;

    /**
     * Prints the information for a finite element mesh
     */
    void Debug() const;

    /**
     * Determines the indices of those vertices with Dirichlet boundary conditions
     * @return index vector.
     */
    virtual std::vector<int> Index_DirichletNodes() const = 0;

    /**
     * Write vector @p v toghether with its mesh information to an ASCii file @p fname.
      *
      * The data are written in C-style.
      *
      * @param[in] fname  file name
      * @param[in] v      vector
     */
    void Write_ascii_matlab(std::string const &fname, std::vector<double> const &v) const;

    /**
     * Visualize @p v together with its mesh information via matlab or octave.
      *
      * Comment/uncomment those code lines in method Mesh:Visualize (geom.cpp)
      * that are supported on your system.
      *
      * @param[in] v    vector
      *
      * @warning matlab files ascii_read_meshvector.m  visualize_results.m
      *          must be in the executing directory.
     */
    void Visualize(std::vector<double> const &v) const;


protected:
    void SetNelem(int nelem)
    {
        _nelem = nelem;
    }

    void SetNverticesElement(int nvert)
    {
        _nvert_e = nvert;
    }

    void SetNdofsElement(int ndof)
    {
        _ndof_e = ndof;
    }

    void SetNnode(int nnode)
```

```
    {
        _nnode = nnode;
    }

    void SetNdim(int ndim)
    {
        _ndim = ndim;
    }

private:
    int _nelem;          //!< number elements
    int _nvert_e;        //!< number of vertices per element
    int _ndof_e;         //!< degrees of freedom (d.o.f.) per element
    int _nnode;          //!< number nodes/vertices
    int _ndim;           //!< space dimension of the problem (1, 2, or 3)
    std::vector<int> _ia;     //!< element connectivity
    std::vector<double> _xc; //!< coordinates
};

/**
 * 2D finite element mesh of the square consiting of linear triangular elements.
 */
class Mesh_2d_3_square: public Mesh
{
public:
    /**
     * Generates the f.e. mesh for the unit square.
     *
     * @param[in] nx    number of discretization intervals in x-direction
     * @param[in] ny    number of discretization intervals in y-direction
     * @param[in] myid  my MPI-rank / subdomain
     * @param[in] procx number of ranks/subdomains in x-direction
     * @param[in] procy number of processes in y-direction
    */
    Mesh_2d_3_square(int nx, int ny, int myid = 0, int procx = 1, int procy = 1);

    /**
     * Destructor
     */
    ~Mesh_2d_3_square() override
    {}

    /**
     * Set solution vector based on a tensor product grid in the rectangle.
     * @param[in] u solution vector
     */
    void SetU(std::vector<double> &u) const;

    /**
     * Set right hand side (rhs) vector on a tensor product grid in the rectangle.
     * @param[in] f rhs vector
     */
    void SetF(std::vector<double> &f) const;

    /**
     * Determines the indices of those vertices with Dirichlet boundary conditions
     * @return index vector.
     */
    std::vector<int> Index_DirichletNodes() const override;

    /**
       * Stores the values of vector @p u of (sub)domain into a file @p name for further processing in gnuplot.
       * The file stores rowise the x- and y- coordinates together with the value from  @p u .
       * The domain [@p xl, @p xr] x [@p yb, @p yt] is discretized into @p nx x @p ny intervals.
       *
```

```
     * @param[in] name basename of file name (file name will be extended by the rank number)
     * @param[in] u    local vector
     *
     * @warning   Assumes tensor product grid in unit square; rowise numbered
     *            (as generated in class constructor).
     *            The output is provided for tensor product grid visualization
     *            ( similar to Matlab-surf() ).
     *
     * @see Mesh_2d_3_square
     */
    void SaveVectorP(std::string const &name, std::vector<double> const &u) const;

    // here will still need to implement in the class
    //  GetBound(), AddBound()
    //  or better a generalized way with indices and their appropriate ranks for MPI communication

private:
    /**
     * Determines the coordinates of the dicretization nodes of the domain [@p xl, @p xr] x [@p yb, @p yt]
     * which is discretized into @p nx x @p ny intervals.
     *
     * @param[in] ny   number of discretization intervals in y-direction
     * @param[in] xl   x-coordinate of left boundary
     * @param[in] xr   x-coordinate of right boundary
     * @param[in] yb   y-coordinate of lower boundary
     * @param[in] yt   y-coordinate of upper boundary
     * @param[out] xc  coordinate vector of length 2n with x(2*k,2*k+1) as coodinates of node k
     */

    void GetCoordsInRectangle(int nx, int ny, double xl, double xr, double yb, double yt,
                              double xc[]);
    /**
     * Determines the element connectivity of linear triangular elements of a FEM discretization
     * of a rectangle using @p nx x @p ny equidistant intervals for discretization.
     * @param[in] nx   number of discretization intervals in x-direction
     * @param[in] ny   number of discretization intervals in y-direction
     * @param[out] ia  element connectivity matrix with ia(3*s,3*s+1,3*s+2) as node numbers od element s
     */
    void GetConnectivityInRectangle(int nx, int ny, int ia[]);

private:
    int _myid;          //!< my MPI rank
    int _procx;         //!< number of MPI ranks in x-direction
    int _procy;         //!< number of MPI ranks in y-direction
    std::array<int, 4> _neigh; //!< MPI ranks of neighbors (negative: no neighbor but b.c.)
    int _color;         //!< red/black coloring (checker board) of subdomains

    double _xl;         //!< x coordinate of lower left  corner of square
    double _xr;         //!< x coordinate of lower right corner of square
    double _yb;         //!< y coordinate or lower left  corner of square
    double _yt;         //!< y coordinate of upper right corner of square
    int    _nx;         //!< number of intervals in x-direction
    int    _ny;         //!< number of intervals in y-direction
};

// #################### still some old code (--> MPI) ############################
/**
 * Copies the values of @p w corresponding to boundary @p ib
 * onto vector s.  South (ib==1), East (ib==2), North (ib==3), West (ib==4).
 * The vector @p s has to be long enough!!
```

```
 * @param[in] ib    my local boundary
 * @param[in] nx    number of discretization intervals in x-direction
 * @param[in] ny    number of discretization intervals in y-direction
 * @param[in] w     vector for all nodes of local discretization
 * @param[out] s    short vector with values on boundary @p ib
*/
// GH_NOTE: Absicherung bei s !!
void GetBound(int ib, int nx, int ny, double const w[], double s[]);


/**
 * Computes @p w := @p w + @p s  at the interface/boundary nodes on the
 * boundary @p ib .  South (ib==1), East (ib==2), North (ib==3), West (ib==4)
 * @param[in] ib    my local boundary
 * @param[in] nx    number of discretization intervals in x-direction
 * @param[in] ny    number of discretization intervals in y-direction
 * @param[in,out] w vector for all nodes of local discretization
 * @param[in] s     short vector with values on boundary @p ib
*/
void AddBound(int ib, int nx, int ny, double w[], double const s[]);

// ################### Mesh from Matlab ###########################
/**
 * 2D finite element mesh of the square consiting of linear triangular elements.
 */
class Mesh_2d_3_matlab: public Mesh
{
public:
    /**
     * Reads mesh data from a binary file.
     *
     * File format, see ascii_write_mesh.m
     *
     * @param[in] fname file name
    */
    explicit Mesh_2d_3_matlab(std::string const &fname);

    /**
     * Determines the indices of those vertices with Dirichlet boundary conditions.
     * @return index vector.
      *
      * @warning All boundary nodes are considered as Dirchlet nodes.
     */
    std::vector<int> Index_DirichletNodes() const override;

private:
    /**
     * Determines the indices of those vertices with Dirichlet boundary conditions
     * @return index vector.
     */
    int Nnbedges() const
    {
        return static_cast<int>(bedges.size());
    }

    std::vector<int> bedges;     //!< boundary edges [nbedges][2] storing start/end vertex

};

#endif
```

```
#include "getmatrix.h"
#include "userset.h"

#include <algorithm>
#include <cassert>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <list>
#include <vector>
#include <omp.h>
using namespace std;


//  general routine for lin. triangular elements

void CalcElem(int const ial[3], double const xc[], double ske[3][3], double fe[3])
//void CalcElem(const int* __restrict__ ial, const double* __restrict__ xc, double* __restrict__ ske[3], double* __restrict__ fe)
{
    const int  i1  = 2 * ial[0],   i2 = 2 * ial[1],   i3 = 2 * ial[2];
    const double x13 = xc[i3 + 0] - xc[i1 + 0],  y13 = xc[i3 + 1] - xc[i1 + 1],
                 x21 = xc[i1 + 0] - xc[i2 + 0],  y21 = xc[i1 + 1] - xc[i2 + 1],
                 x32 = xc[i2 + 0] - xc[i3 + 0],  y32 = xc[i2 + 1] - xc[i3 + 1];
    const double jac = fabs(x21 * y13 - x13 * y21);

    ske[0][0] = 0.5 / jac * (y32 * y32 + x32 * x32);
    ske[0][1] = 0.5 / jac * (y13 * y32 + x13 * x32);
    ske[0][2] = 0.5 / jac * (y21 * y32 + x21 * x32);
    ske[1][0] = ske[0][1];
    ske[1][1] = 0.5 / jac * (y13 * y13 + x13 * x13);
    ske[1][2] = 0.5 / jac * (y21 * y13 + x21 * x13);
    ske[2][0] = ske[0][2];
    ske[2][1] = ske[1][2];
    ske[2][2] = 0.5 / jac * (y21 * y21 + x21 * x21);

    const double xm    = (xc[i1 + 0] + xc[i2 + 0] + xc[i3 + 0]) / 3.0,
                 ym    = (xc[i1 + 1] + xc[i2 + 1] + xc[i3 + 1]) / 3.0;
    //fe[0] = fe[1] = fe[2] = 0.5 * jac * FunctF(xm, ym) / 3.0;
    fe[0] = fe[1] = fe[2] = 0.5 * jac * fNice(xm, ym) / 3.0;
}


// general routine for lin. triangular elements,
// non-symm. matrix
// node numbering in element:  a s c e n d i n g   indices !!
[[deprecated("Use CRS_Matrix::AddElem_3(...) instead.")]]
void AddElem(int const ial[3], double const ske[3][3], double const fe[3],
             int const id[], int const ik[], double sk[], double f[])
{
    for (int i = 0; i < 3; ++i)
    {
        const int ii  = ial[i],            // row ii (global index)
                  id1 = id[ii],            // start and
                  id2 = id[ii + 1];        // end of row ii in matrix
        int ip  = id1;
        for (int j = 0; j < 3; ++j)        // no symmetry assumed
        {
            const int jj = ial[j];
            bool not_found = true;
            do       // find entry jj (global index) in row ii
            {
                not_found = (ik[ip] != jj);
                ++ip;
            }
            while (not_found && ip < id2);

#ifndef NDEBUG                     // compiler option -DNDEBUG switches off the check
```

```
            if (not_found)     // no entry found !!
            {
                cout << "Error in AddElem: (" << ii << "," << jj << ") ["
                     << ial[0] << "," << ial[1] << "," << ial[2] << "]\n";
                assert(!not_found);
            }
#endif
            sk[ip - 1] += ske[i][j];
        }
        f[ii] += fe[i];
    }
}


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




// #####################################################################

CRS_Matrix::CRS_Matrix(Mesh const &mesh)
    : _mesh(mesh), _nrows(0), _nnz(0), _id(0), _ik(0), _sk(0)
{
    Derive_Matrix_Pattern();
    return;
}

void CRS_Matrix::Derive_Matrix_Pattern()
{
    int const nelem(_mesh.Nelems());
    int const ndof_e(_mesh.NdofsElement());
    auto const &ia(_mesh.GetConnectivity());
//  Determine the number of matrix rows
    _nrows = *max_element(ia.cbegin(), ia.cbegin() + ndof_e * nelem);
    ++_nrows;                                // node numberng: 0 ... nnode-1
    assert(*min_element(ia.cbegin(), ia.cbegin() + ndof_e * nelem) == 0); // numbering starts with 0 ?

//  Collect for each node those nodes it is connected to (multiple entries)
//  Detect the neighboring nodes
    vector< list<int> > cc(_nrows);             //  cc[i] is the  list of nodes a node i is connected to
    for (int i = 0; i < nelem; ++i)
    {
        int const idx = ndof_e * i;
        for (int k = 0; k < ndof_e; ++k)
        {
            list<int> &cck = cc.at(ia[idx + k]);
            cck.insert( cck.end(), ia.cbegin() + idx, ia.cbegin() + idx + ndof_e );
        }
    }
//  Delete the multiple entries
    _nnz = 0;
    for (auto &it : cc)
    {
        it.sort();
        it.unique();
        _nnz += static_cast<int>(it.size());
        // cout << it.size() << " :: "; copy(it->begin(),it->end(), ostream_iterator<int,char>(cout,"  ")); cout << endl;
    }

// CSR data allocation
    _id.resize(_nrows + 1);                   // Allocate memory for CSR row pointer
    _ik.resize(_nnz);                         // Allocate memory for CSR column index vector

```

```
//  copy CSR data
    _id[0] = 0;                               // begin of first row
    for (size_t i = 0; i < cc.size(); ++i)
    {
        //cout << i << "    " << nid.at(i) << endl;;
        const list<int> &ci = cc.at(i);
        const auto nci = static_cast<int>(ci.size());
        _id[i + 1] = _id[i] + nci; // begin of next line
        copy(ci.begin(), ci.end(), _ik.begin() + _id[i] );
    }

    assert(_nnz == _id[_nrows]);
    _sk.resize(_nnz);                     // Allocate memory for CSR column index vector
    return;
}


void CRS_Matrix::Debug() const
{
//  ID points to first entry of row
//  no symmetry assumed
    cout << "\nMatrix  (nnz = " << _id[_nrows] << ")\n";

    for (int row = 0; row < _nrows; ++row)
    {
        cout << "Row " << row << " : ";
        int const id1 = _id[row];
        int const id2 = _id[row + 1];
        for (int j = id1; j < id2; ++j)
        {
            cout.setf(ios::right, ios::adjustfield);
            cout << "[" << setw(2) << _ik[j] << "]  " << setw(4) << _sk[j] << "   ";
        }
        cout << endl;
    }
    return;
}

void CRS_Matrix::CalculateLaplace(vector<double> &f)
{
    double t_start = omp_get_wtime();

    double t0 ,t1,t2;
    assert(_mesh.NdofsElement() == 3);              // only for triangular, linear elements
    //cout << _nnz << " vs. " << _id[_nrows] << "  " << _nrows<< endl;
    assert(_nnz == _id[_nrows]);

    #pragma omp parallel
    {
        #pragma omp for
        for (int k = 0; k < _nrows; ++k)
            {
                _sk[k] = 0.0;
            }

        #pragma omp barrier
        #pragma omp single
        t0 = omp_get_wtime();


        #pragma omp for
        for (int k = 0; k < _nrows; ++k)
        {
            f[k] = 0.0;
        }

```

```
        #pragma omp barrier
        #pragma omp single
        t1 = omp_get_wtime();


        double ske[3][3], fe[3];
        //  Loop over all elements
        auto const nelem = _mesh.Nelems();
        auto const &ia   = _mesh.GetConnectivity();
        auto const &xc   = _mesh.GetCoords();

        #pragma omp barrier
        #pragma omp single
        t2 = omp_get_wtime();

        #pragma omp for
        for (int i = 0; i < nelem; ++i)
        {
            CalcElem(ia.data() + 3 * i, xc.data(), ske, fe);
            AddElem_3(ia.data() + 3 * i, ske, fe, f);
        }
        //Debug();

    }
    double t3 = omp_get_wtime();
    cout << "Zero matrix: " << (t0 - t_start) << " sec\n";
    cout << "Zero RHS: " << (t1 - t0)      << " sec\n";
    cout << "Element loop: " << (t3 - t2)     << " sec\n";
    cout << "Total assembly:" << (t3 - t_start) << " sec\n";



}

void CRS_Matrix::ApplyDirichletBC(std::vector<double> const &u, std::vector<double> &f)
{
    double const PENALTY = 1e6;
    auto const idx = _mesh.Index_DirichletNodes();
    int const nidx = static_cast<int>(idx.size());

    for (int row = 0; row < nidx; ++row)
    {
        int const k = idx[row];
        int const id1 = fetch(k, k); // Find diagonal entry of row
        assert(id1 >= 0);
        _sk[id1] += PENALTY;              // matrix weighted scaling feasible
        f[k] += PENALTY * u[k];
    }

    return;
}

void CRS_Matrix::GetDiag(vector<double> &d) const
{
    assert( _nrows == static_cast<int>(d.size()) );

    for (int row = 0; row < _nrows; ++row)
```

parallel

```
    {
        const int ia = fetch(row, row); // Find diagonal entry of row
        assert(ia >= 0);
        d[row] = _sk[ia];
    }
    return;
}

bool CRS_Matrix::Compare2Old(int nnode, int const id[], int const ik[], double const sk[]) const
```

```
{
    bool bn = (nnode == _nrows);      // number of rows
    if (!bn)
    {
        cout << "#########   Error: " << "number of rows" << endl;
    }

    bool bz = (id[nnode] == _nnz);    // number of non zero elements
    if (!bz)
    {
        cout << "#########   Error: " << "number of non zero elements" << endl;
    }

    bool bd = equal(id, id + nnode + 1, _id.cbegin()); // row starts
    if (!bd)
    {
        cout << "#########   Error: " << "row starts" << endl;
    }

    bool bk = equal(ik, ik + id[nnode], _ik.cbegin()); // column indices
    if (!bk)
    {
        cout << "#########   Error: " << "column indices" << endl;
    }

    bool bv = equal(sk, sk + id[nnode], _sk.cbegin()); // values
    if (!bv)
    {
        cout << "#########   Error: " << "values" << endl;
    }

    return bn && bz && bd && bk && bv;
}


void CRS_Matrix::Mult(vector<double> &w, vector<double> const &u) const
{
    assert( _nrows == static_cast<int>(w.size()) );
    assert( w.size() == u.size() );

    for (int row = 0; row < _nrows; ++row)
    {
        double wi = 0.0;
        for (int ij = _id[row]; ij < _id[row + 1]; ++ij)
        {
            wi += _sk[ij] * u[ _ik[ij] ];
        }
        w[row] = wi;
    }
    return;
}

void CRS_Matrix::Defect(vector<double> &w,
                        vector<double> const &f, vector<double> const &u) const
{
    assert( _nrows == static_cast<int>(w.size()) );
    assert( w.size() == u.size() && u.size() == f.size() );

    #pragma omp parallel for
    for (int row = 0; row < _nrows; ++row)
    {
        double wi = f[row];
        for (int ij = _id[row]; ij < _id[row + 1]; ++ij)
        {
            wi -= _sk[ij] * u[ _ik[ij] ];
        }
        w[row] = wi;
    }
```

```
    return;
}

int CRS_Matrix::fetch(int const row, int const col) const
{
    int const id2 = _id[row + 1];    // end   and
    int       ip  = _id[row];        // start of recent row (global index)

    while (ip < id2 && _ik[ip] != col)  // find index col (global index)
    {
        ++ip;
    }
    if (ip >= id2)
    {
        ip = -1;
#ifndef NDEBUG                 // compiler option -DNDEBUG switches off the check
        cout << "No column  " << col << "  in row  " << row << endl;
        assert(ip >= id2);
#endif
    }
    return ip;
}


// general routine for lin. triangular elements,
// non-symm. matrix
// node numbering in element:  a s c e n d i n g   indices !!
void CRS_Matrix::AddElem_3(int const ial[3], double const ske[3][3], double const fe[3], vector<double> &f)
{
    for (int i = 0; i < 3; ++i)
    {
        const int ii  = ial[i];           // row ii (global index)
        for (int j = 0; j < 3; ++j)       // no symmetry assumed
        {
            const int jj = ial[j];        // column jj (global index)
            int ip = fetch(ii, jj);       // find column entry jj in row ii
#ifndef NDEBUG                 // compiler option -DNDEBUG switches off the check
            if (ip < 0)          // no entry found !!
            {
                cout << "Error in AddElem: (" << ii << "," << jj << ") ["
                     << ial[0] << "," << ial[1] << "," << ial[2] << "]\n";
                assert(ip >= 0);
            }
#endif
            #pragma omp atomic
            _sk[ip] += ske[i][j];
        }
        #pragma omp atomic
        f[ii] += fe[i];
    }
}

```

```
#ifndef GETMATRIX_FILE
#define GETMATRIX_FILE

#include "geom.h"
#include <vector>

/**
 * Calculates the element stiffness matrix @p ske and the element load vector @p fe
 * of one triangular element with linear shape functions.
 * @param[in]   ial     node indices of the three element vertices
 * @param[in]   xc      vector of node coordinates with x(2*k,2*k+1) as coodinates of node k
 * @param[out] ske      element stiffness matrix
 * @param[out] fe       element load vector
 */
void CalcElem(int const ial[3], double const xc[], double ske[3][3], double fe[3]);

/**
 * Adds the element stiffness matrix @p ske and the element load vector @p fe
 * of one triangular element with linear shape functions to the appropriate positions in
 * the symmetric stiffness matrix, stored as CSR matrix K(@p sk,@p id, @p ik)
 *
 * @param[in] ial   node indices of the three element vertices
 * @param[in] ske       element stiffness matrix
 * @param[in] fe        element load vector
 * @param[out] sk       vector non-zero entries of CSR matrix
 * @param[in] id        index vector containing the first entry in a CSR row
 * @param[in] ik        column index vector of CSR matrix
 * @param[out] f        distributed local vector storing the right hand side
 *
 * @warning Algorithm requires indices in connectivity @p ial in ascending order.
 *          Currently deprecated.
*/
void AddElem(int const ial[3], double const ske[3][3], double const fe[3],
             int const id[], int const ik[], double sk[], double f[]);


// #####################################################################
/**
 * Square matrix in CRS format (compressed row storage; also named CSR),
 * see an <a href="https://en.wikipedia.org/wiki/Sparse_matrix">introduction</a>.
 */
class CRS_Matrix
{
    public:
       /**
        * Intializes the CRS matrix structure from the given discetization in @p mesh.
        *
        * The sparse matrix pattern is generated but the values are 0.
        *
        * @param[in] mesh given discretization
        *
        * @warning A reference to the discretization @p mesh is stored inside this class.
        *          Therefore, changing @p mesh outside requires also
        *          to call method @p Derive_Matrix_Pattern explicitely.
        *
        * @see Derive_Matrix_Pattern
        */
       explicit CRS_Matrix(Mesh const & mesh);

      /**
        * Destructor.
        */
       ~CRS_Matrix()
       {}
```

```
        /**
         * Generates the sparse matrix pattern and overwrites the existing pattern.
         *
         * The sparse matrix pattern is generated but the values are 0.
        */
        void Derive_Matrix_Pattern();

         /**
         * Calculates the entries of f.e. stiffness matrix and load/rhs vector @p f for the Laplace operator in 2D.
         * No memory is allocated.
         *
         * @param[in,out] f (preallocated) rhs/load vector
         */
        void CalculateLaplace(std::vector<double> &f);

        /**
         * Applies Dirichlet boundary conditions to stiffness matrix and to load vector @p f.
         * The <a href="https://www.jstor.org/stable/2005611?seq=1#metadata_info_tab_contents">penalty method</a>
         * is used for incorporating the given values @p u.
         *
         * @param[in]     u (global) vector with Dirichlet data
         * @param[in,out] f load vector
         */
        void ApplyDirichletBC(std::vector<double> const &u, std::vector<double> &f);

        /**
         * Extracts the diagonal elemenst of the sparse matrix.
         *
         * @param[in,out]  d  (prellocated) vector of diagonal elements
         */
        void GetDiag(std::vector<double> &d) const;

        /**
         * Performs the matrix-vector product  w := K*u.
         *
         * @param[in,out] w resulting vector (preallocated)
         * @param[in]     u vector
         */
        void Mult(std::vector<double> &w, std::vector<double> const &u) const;

         /**
         * Calculates the defect/residuum w := f - K*u.
         *
         * @param[in,out] w resulting vector (preallocated)
         * @param[in]     f load vector
         * @param[in]     u vector
         */
        void Defect(std::vector<double> &w,
                    std::vector<double> const &f, std::vector<double> const &u) const;

        /**
                 * Number rows in matrix.
                 * @return number of rows.
                 */
        int Nrows() const
           {return _nrows;}

        /**
                 * Show the matrix entries.
                 */
        void Debug() const;

        /**
```

```
                 * Finds in a CRS matrix the access index for an entry at row @p row and column @p col.
                 *
                 * @param[in] row       row index
                 * @param[in] col       column index
                 * @return index for element (@p row, @p col). If no appropriate entry exists then -1 will be returned.
                 *
                 * @warning assert() stops the function in case that matrix element (@p row, @p col) doesn't exist.
                */
        int fetch(int row, int col) const;

       /**
         * Adds the element stiffness matrix @p ske and the element load vector @p fe
         * of one triangular element with linear shape functions to the appropriate positions in
         * the stiffness matrix, stored as CSR matrix K(@p sk,@p id, @p ik).
         *
         * @param[in]     ial   node indices of the three element vertices
         * @param[in]     ske   element stiffness matrix
         * @param[in]     fe    element load vector
         * @param[in,out] f         distributed local vector storing the right hand side
         *
         * @warning Algorithm assumes  linear triangular elements (ndof_e==3).
        */
        void AddElem_3(int const ial[3], double const ske[3][3], double const fe[3], std::vector<double> &f);

         /**
         * Compare @p this CRS matrix with an external CRS matrix stored in C-Style.
         *
         * The method prints statements on differences found.
         *
         * @param[in]     nnode  row number of external matrix
         * @param[in]     id     start indices of matrix rows of external matrix
         * @param[in]     ik     column indices of external matrix
         * @param[in]     sk     non-zero values of external matrix
         *
         * @return true iff all data are identical.
         */
        bool Compare2Old(int nnode, int const id[], int const ik[], double const sk[]) const;

     private:
        Mesh const & _mesh;      //!< reference to discretization
        int _nrows;              //!< number of rows in matrix
        int _nnz;                //!< number of non-zero entries
        std::vector<int> _id;    //!< start indices of matrix rows
        std::vector<int> _ik;    //!< column indices
        std::vector<double> _sk; //!< non-zero values

};


#endif
```

```cpp
#include "vdop.h"
#include "getmatrix.h"
#include "jacsolve.h"

#include <cassert>
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;

// #####################################################################
//      const int neigh[], const int color,
//      const MPI::Intracomm& icomm,
void JacobiSolve(CRS_Matrix const &SK, vector<double> const &f, vector<double> &u)
{
    const double omega   = 1.0;
    const int    maxiter = 1000;
    const double tol  = 1e-5,                   // tolerance
                 tol2 = tol * tol;              // tolerance^2

    int nrows = SK.Nrows();                     // number of rows == number of columns
    assert( nrows == static_cast<int>(f.size()) && f.size() == u.size() );

    cout << endl << " Start Jacobi solver for " << nrows << " d.o.f.s"  << endl;
    //  Choose initial guess
    #pragma omp parallel for
    for (int k = 0; k < nrows; ++k)
    {
        u[k] = 0.0;                             //  u := 0
    }

    vector<double> dd(nrows);                   // matrix diagonal
    vector<double>  r(nrows);                   // residual
    vector<double>  w(nrows);                   // correction

    SK.GetDiag(dd);                             //  dd := diag(K)
    ////DebugVector(dd);{int ijk; cin >> ijk;}

    //  Initial sweep
    SK.Defect(r, f, u);                         //  r := f - K*u

    vddiv(w, r, dd);                            //  w := D^{-1}*r
    double sigma0 = dscapr(w, r);               // s0 := <w,r>

    // Iteration sweeps
    int iter  = 0;
    double sigma = sigma0;
    // i dont parallelize this cause its iteration dependent
    while ( sigma > tol2 * sigma0 && maxiter > iter)
    {
        ++iter;
        vdaxpy(u, u, omega, w );                //  u := u + om*w
        SK.Defect(r, f, u);                     //  r := f - K*u
        vddiv(w, r, dd);                        //  w := D^{-1}*r
        sigma = dscapr(w, r);                   // s0 := <w,r>
//              cout << "Iteration " << iter << " : " << sqrt(sigma/sigma0) << endl;
    }
    cout << "aver. Jacobi rate :  " << exp(log(sqrt(sigma / sigma0)) / iter) << "  (" << iter << " iter)" << endl;
    cout << "final error: " << sqrt(sigma / sigma0) << " (rel)   " << sqrt(sigma) << " (abs)\n";

    return;
}
```

```cpp
#ifndef JACSOLVE_FILE
#define JACSOLVE_FILE
#include "getmatrix.h"
#include <vector>


/**
 * Solves linear system of equations  K @p u = @p f  via the Jacobi iteration.
 * We use a distributed symmetric  CSR matrix @p SK and initial guess of the
 * solution is set to 0.
 * @param[in] SK        CSR matrix
 * @param[in] f         distributed local vector storing the right hand side
 * @param[out] u        accumulated local vector storing the solution.
*/
void JacobiSolve(CRS_Matrix const &SK, std::vector<double> const &f, std::vector<double> &u);


#endif
```

```cpp
//              MPI code in C++.
//              See [Gropp/Lusk/Skjellum, "Using MPI", p.33/41 etc.]
//              and  /opt/mpich/include/mpi2c++/comm.h  for details

#include "geom.h"
#include "getmatrix.h"
#include "jacsolve.h"
#include "userset.h"
#include "vdop.h"

#include <chrono>             // timing
#include <cmath>
#include <iostream>
#include <omp.h>
using namespace std;
using namespace std::chrono;  // timing


int main(int, char ** )
{
    const int numprocs = 1;
    const int myrank   = 0;
    if (myrank == 0)
    {
        cout << "\n There are " << numprocs << " processes running.\n \n";
    }

    const auto procx = static_cast<int>(sqrt(numprocs + 0.0));
    const int  procy = procx;

    if (procy * procx != numprocs)
    {
        cout << "\n Wrong number of processors !\n \n";
    }
    else
    {
// #####################################################################
//      Here starts the real code
// #####################################################################
        //bool ScaleUp = !true;
        int nx, ny, NXglob, NYglob; /* number of local intervals on (xl,xr)=:nx, (yb,yt)=:ny */
        //nx = 1024;
        //ny = 1024;
        nx = 100;
        ny = 100;
        NXglob = nx * procx;
        NYglob = ny * procy;
        cout << "Intervalls: " << NXglob << " x " << NYglob << endl;

// ##################### STL ###########################################
        {
            Mesh_2d_3_square const mesh(nx, ny);
            //mesh.Debug();

            CRS_Matrix SK(mesh);                   // CRS matrix
            //SK.Debug();
            vector<double> uv(SK.Nrows(), 0.0);    // temperature
            vector<double> fv(SK.Nrows(), 0.0);    // r.h.s.

            SK.CalculateLaplace(fv);
            //SK.Debug();

            //mesh.SetU(uv);         // deprecated
            //mesh.SetF(fv);         // deprecated
            // Two ways to initialize the vector
            //mesh.SetValues(uv,f_zero);             // functional
            mesh.SetValues(uv, [](double x, double y) -> double {return 0.0 * x *y;}
```

```
); // lambda function

            SK.ApplyDirichletBC(uv, fv);
            //SK.Compare2Old(nnode, id, ik, sk);
            //SK.Debug();

            auto tstart = system_clock::now();       // start timer

            JacobiSolve(SK, fv, uv );          // solve the system of equations

            auto tend = system_clock::now();         // end timer
            auto duration = duration_cast<microseconds>(tend - tstart);
            auto t1 = static_cast<double>(duration.count()) / 1e6 ;    // t1 in seconds
            cout << "JacobiSolve: timing in sec. : " << t1 << endl;

            //CompareVectors(uv, nnode, u, 1e-6);    // Check correctness

            //mesh.SaveVectorP("t.dat", uv);
            //mesh.Visualize(uv);

        }
// #################### STL ##########################################
        {
            //Mesh_2d_3_matlab const mesh("square_tiny.txt");
            Mesh_2d_3_matlab const mesh("square_100.txt");
            //Mesh_2d_3_matlab const mesh("L_shape.txt");
            //mesh.Debug();

            CRS_Matrix SK(mesh);                    // CRS matrix
            //SK.Debug();

            vector<double> uv(SK.Nrows(), 0.0);    // temperature
            vector<double> fv(SK.Nrows(), 0.0);    // r.h.s.

            SK.CalculateLaplace(fv);
            //SK.Debug();

            //mesh.SetU(uv);         // deprecated
            // Two ways to initialize the vector
            //mesh.SetValues(uv,f_zero);             // user function
            mesh.SetValues(uv, [](double x, double y) -> double {return 0.0 * x *y;}
); // lambda function

            SK.ApplyDirichletBC(uv, fv);
            //SK.Compare2Old(nnode, id, ik, sk);
            //SK.Debug();

            auto tstart = system_clock::now();       // start timer

            JacobiSolve(SK, fv, uv );          // solve the system of equations

            auto tend = system_clock::now();         // end timer
            auto duration = duration_cast<microseconds>(tend - tstart);
            auto t1 = static_cast<double>(duration.count()) / 1e6 ;    // t1 in seconds
            cout << "JacobiSolve: timing in sec. : " << t1 << endl;

            //mesh.Write_ascii_matlab("uv.txt", uv);
            //mesh.Visualize(uv);

        }
        return 0;
    }
}
```

```cpp
#include "userset.h"
#include <cmath>


double FunctF(double const x, double const y)
{
// return  std::sin(3.14159*1*x)*std::sin(3.14159*1*y);
//  return 16.0*1024. ;
// return (double)1.0 ;
    return x * x * std::sin(2.5 * 3.14159 * y);
}

double FunctU(const double /* x */, double const /* y */)
{
    return 1.0 ;
}
```

```
#ifndef USERSET_FILE
#define USERSET_FILE
#include <cmath>
/**
 * User function: f(@p x,@p y)
 * @param[in] x         x-coordinate of discretization point
 * @param[in] y         y-coordinate of discretization point
 * @return  value for right hand side f(@p x,@p y)
 */
double FunctF(double const x, double const y);

/**
 * User function: u(@p x,@p y)
 * @param[in] x         x-coordinate of discretization point
 * @param[in] y         y-coordinate of discretization point
 * @return  value for solution vector u(@p x,@p y)
 */
double FunctU(double const x, double const y);


/**
 * User function: f(@p x,@p y) = @f$ x^2 \sin(2.5\pi y)@f$.
 * @param[in] x         x-coordinate of discretization point
 * @param[in] y         y-coordinate of discretization point
 * @return  value f(@p x,@p y)
 */
inline double fNice(double const x, double const y)
{
    return x * x * std::sin(2.5 * 3.14159 * y);
}

/**
 * User function: f(@p x,@p y) = 0$.
 * @param[in] x         x-coordinate of discretization point
 * @param[in] y         y-coordinate of discretization point
 * @return  value 0
 */
inline double f_zero(double const x, double const y)
//double f_zero(double const  /*x*/, double const /*y*/)
{
    return 0.0 + 0.0*(x+y);
}

#endif
```

```cpp
#include "vdop.h"
#include <cassert>                  // assert()
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;


void vddiv(vector<double> &x, vector<double> const &y,
           vector<double> const &z)
{
    assert( x.size() == y.size() && y.size() == z.size() );
    size_t n = x.size();
    #pragma omp parallel for
    for (size_t k = 0; k < n; ++k)
    {
        x[k] = y[k] / z[k];
    }
    return;
}

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

void vdaxpy(std::vector<double> &x, std::vector<double> const &y,
            double alpha, std::vector<double> const &z )
{
    assert( x.size() == y.size() && y.size() == z.size() );
    size_t n = x.size();
    #pragma omp parallel for
    for (size_t k = 0; k < n; ++k)
    {
        x[k] = y[k] + alpha * z[k];
    }
    return;
}
//******************************************************************************

double dscapr(std::vector<double> const &x, std::vector<double> const &y)
{
    assert( x.size() == y.size());
    size_t n = x.size();

    double    s = 0.0;
     #pragma omp parallel for reduction(+:s)
    for (size_t k = 0; k < n; ++k)
    {
        s += x[k] * y[k];
    }

    return s;
}

//******************************************************************************
void DebugVector(vector<double> const &v)
{
    cout << "\nVector  (nnode = " << v.size() << ")\n";
    for (size_t j = 0; j < v.size(); ++j)
    {
        cout.setf(ios::right, ios::adjustfield);
        cout << v[j] << "   ";
    }
    cout << endl;

    return;
}
//******************************************************************************
bool CompareVectors(std::vector<double> const &x, int const n, double const y[], double const eps)
```

```cpp
{
    bool bn = (static_cast<int>(x.size()) == n);
    if (!bn)
    {
        cout << "#########   Error: " << "number of elements" << endl;
    }
    //bool bv = equal(x.cbegin(),x.cend(),y);
    bool bv = equal(x.cbegin(), x.cend(), y,
                    [eps](double a, double b) -> bool
    { return std::abs(a - b) < eps * (1.0 + 0.5 * (std::abs(a) + std::abs(a))); }
                   );
    if (!bv)
    {
        assert(static_cast<int>(x.size()) == n);
        cout << "#########   Error: " << "values" << endl;
    }
    return bn && bv;
}
```

```cpp
#ifndef VDOP_FILE
#define VDOP_FILE
#include <vector>

/** @brief  Element-wise vector divison x_k = y_k/z_k.
 *
 * @param[out] x  target vector
 * @param[in]  y  source vector
 * @param[in]  z  source vector
 *
 */
void vddiv(std::vector<double> & x, std::vector<double> const& y,
                                    std::vector<double> const& z);

/** @brief  Element-wise daxpy operation x(k) = y(k) + alpha*z(k).
 *
 * @param[out] x  target vector
 * @param[in]  y  source vector
 * @param[in]  alpha  scalar
 * @param[in]  z  source vector
 *
 */
void vdaxpy(std::vector<double> & x, std::vector<double> const& y,
                       double alpha, std::vector<double> const& z );


/** @brief  Calculates the Euclidian inner product of two vectors.
 *
 * @param[in]  x vector
 * @param[in]  y vector
 * @return Euclidian inner product @f$\langle x,y \rangle@f$
 *
 */
double dscapr(std::vector<double> const& x, std::vector<double> const& y);


/**
 * Print entries of a vector.
 * @param[in] v     vector values
*/
void DebugVector(std::vector<double> const &v);

/** @brief  Compares an STL vector with POD vector.
 *
 * The accuracy criteria @f$ |x_k-y_k| < \varepsilon \left({1+0.5(|x_k|+|y_k|)}\right) @f$
 * follows the book by
 * <a href="https://www.springer.com/la/book/9783319446592">Stoyan/Baran</a>, p.8.
 *
 * @param[in]  x    STL vector
 * @param[in]  n    length of POD vector
 * @param[in]  y    POD vector
 * @param[in]  eps  relative accuracy criteria (default := 0.0).
 * @return true iff pairwise vector elements are relatively close to each other.
 *
 */
bool CompareVectors(std::vector<double> const& x, int n, double const y[], double const eps=0.0);

#endif
```