```
    1: // see:   http://llvm.org/docs/CodingStandards.html#include-style
    2: #include "geom.h"
    3:
    4: #include <algorithm>
    5: #include <cassert>
    6: #include <fstream>
    7: #include <iostream>
    8: #include <list>
    9: #include <string>
   10: #include <vector>
   11: using namespace std;
   12:
   13: Mesh::Mesh(int ndim, int nvert_e, int ndof_e)
   14:     : _nelem(0), _nvert_e(nvert_e), _ndof_e(ndof_e), _nnode(0), _ndim(ndim), _ia(0),
 _xc(0)
   15: {
   16: }
   17:
   18: Mesh::~Mesh()
   19: {}
   20:
   21: void Mesh::SetValues(std::vector<double> &v, const std::function<double(double, doub
le)> &func) const
   22: {
   23:     int const nnode = Nnodes();             // number of vertices in mesh
   24:     assert( nnode == static_cast<int>(v.size()) );
   25:     for (int k = 0; k < nnode; ++k)
   26:     {
   27:         v[k] = func( _xc[2 * k], _xc[2 * k + 1] );
   28:     }
   29: }
   30:
   31:
   32: void Mesh::Debug() const
   33: {
   34:     cout << "\n ################ Debug  M E S H  ####################\n";
   35:     cout << "\n ..............    Coordinates       ..................\n";
   36:     for (int k = 0; k < _nnode; ++k)
   37:     {
   38:         cout << k << " : " ;
   39:         for (int i = 0; i < _ndof_e; ++i )
   40:         {
   41:             cout << _xc[2*k+i] << "  ";
   42:         }
   43:         cout << endl;
   44:     }
   45:     cout << "\n ..............    Elements        ..................\n";
   46:     for (int k = 0; k < _nelem; ++k)
   47:     {
   48:         cout << k << " : ";
   49:         for (int i = 0; i < _ndof_e; ++i )
   50:             cout << _ia[_ndof_e * k + i] << "  ";
   51:         cout << endl;
   52:     }
   53:     return;
   54: }
   55:
   56: void Mesh::Write_ascii_matlab(std::string const &fname, std::vector<double> const &v
) const
   57: {
   58:     assert(Nnodes() ==  static_cast<int>(v.size()));  // fits vector length to mesh
information?
   59:
   60:     ofstream fout(fname);                             // open file ASCII mode
   61:     if ( !fout.is_open() )
   62:     {
   63:         cout << "\nFile " << fname << " has not been opened.\n\n" ;
   64:         assert( fout.is_open() && "File not opened."  );
```

```
   65:     }
   66:
   67:     string const DELIMETER(" ");     // define the same delimeter as in matlab/ascii_
read*.m
   68:     int const    OFFSET(1);          // convert C-indexing to matlab
   69:
   70:     // Write data: #nodes, #space dimensions, #elements, #vertices per element
   71:     fout << Nnodes() << DELIMETER << Ndims() << DELIMETER << Nelems() << DELIMETER <
< NverticesElements() << endl;
   72:
   73:     // Write cordinates: x_k, y_k   in seperate lines
   74:     assert( Nnodes()*Ndims() ==  static_cast<int>(_xc.size()));
   75:     for (int k = 0, kj = 0; k < Nnodes(); ++k)
   76:     {
   77:         for (int j = 0; j < Ndims(); ++j, ++kj)
   78:         {
   79:             fout << _xc[kj] << DELIMETER;
   80:         }
   81:         fout << endl;
   82:     }
   83:
   84:     // Write connectivity: ia_k,0, ia_k,1 etc  in seperate lines
   85:     assert( Nelems()*NverticesElements() ==  static_cast<int>(_ia.size()));
   86:     for (int k = 0, kj = 0; k < Nelems(); ++k)
   87:     {
   88:         for (int j = 0; j < NverticesElements(); ++j, ++kj)
   89:         {
   90:             fout << _ia[kj] + OFFSET << DELIMETER;     // C to matlab
   91:         }
   92:         fout << endl;
   93:     }
   94:
   95:     // Write vector
   96:     for (int k = 0; k < Nnodes(); ++k)
   97:     {
   98:         fout << v[k] << endl;
   99:     }
  100:
  101:     fout.close();
  102:     return;
  103: }
  104:
  105:
  106: void Mesh::Visualize(std::vector<double> const &v) const
  107: {
  108:     // define external command
  109:     const string exec_m("matlab -nosplash < visualize_results.m");                 /
/ Matlab
  110:     //const string exec_m("octave --no-window-system --no-gui visualize_results.m");
 // Octave
  111:     //const string exec_m("flatpak run org.octave.Octave visualize_results.m");
 // Octave (flatpak): desktop GH
  112:
  113:     const string fname("uv.txt");
  114:     Write_ascii_matlab(fname, v);
  115:
  116:     int ierror = system(exec_m.c_str());                                  // call ext
ernal command
  117:
  118:     if (ierror != 0)
  119:     {
  120:         cout << endl << "Check path to Matlab/octave on your system" << endl;
  121:     }
  122:     cout << endl;
  123:     return;
  124: }
  125:
  126:
```

```
127:
128:
129: // ######################################################################
130: Mesh_2d_3_square::Mesh_2d_3_square(int nx, int ny, int myid, int procx, int procy)
131:     : Mesh(2, 3, 3), // two dimensions, 3 vertices, 3 dofs
132:       _myid(myid), _procx(procx), _procy(procy), _neigh{{-1, -1, -1, -1}}, _color(0)
,
133: _xl(0.0), _xr(1.0), _yb(0.0), _yt(1.0), _nx(nx), _ny(ny)
134: {
135:     //void IniGeom(int const myid, int const procx, int const procy, int neigh[], in
t &color)
136:     int const ky = _myid / _procx;
137:     int const kx = _myid % _procy;  //    MOD(myid,procx)
138:     // Determine the neighbors of domain/rank myid
139:     _neigh[0] = (ky == 0)       ?  -1 : _myid - _procx;    //   South
140:     _neigh[1] = (kx == _procx - 1) ?  -1 : _myid + 1;      //   East
141:     _neigh[2] = (ky == _procy - 1) ?  -1 : _myid + _procx;  //   North
142:     _neigh[3] = (kx == 0)       ?  -1 : _myid - 1;        //   West
143:
144:     _color = (kx + ky) & 1 ;
145:
146:     // subdomain is part of unit square
147:     double const hx = 1. / _procx;
148:     double const hy = 1. / _procy;
149:     _xl = kx * hx;                      //  left
150:     _xr = (kx + 1) * hx;                //  right
151:     _yb = ky * hy;                      //  bottom
152:     _yt = (ky + 1) * hy;                //  top
153:
154:     // Calculate coordinates
155:     int const nnode = (_nx + 1) * (_ny + 1); // number of nodes
156:     Resize_Coords(nnode, 2);                 // coordinates in 2D [nnode][ndim]
157:     GetCoordsInRectangle(_nx, _ny, _xl, _xr, _yb, _yt, GetCoords().data());
158:
159:     // Calculate element connectivity (linear triangles)
160:     int const nelem = 2 * _nx * _ny;         // number of elements
161:     Resize_Connectivity(nelem, 3);           // connectivity matrix [nelem][3]
162:     GetConnectivityInRectangle(_nx, _ny, GetConnectivity().data());
163:
164:     return;
165: }
166:
167:
168: void Mesh_2d_3_square::SetU(std::vector<double> &u) const
169: {
170:     int dx    = _nx + 1;
171: #pragma omp parallel for shared(_ny)
172:     for (int j = 0; j <= _ny; ++j)
173:     {
174:         int k = j * dx;
175:         for (int i = 0; i <= _nx; ++i, ++k)
176:         {
177:             u[k] = 0.0;
178:         }
179:     }
180:
181: }
182:
183: void Mesh_2d_3_square::SetF(std::vector<double> &f) const
184: {
185:     int dx    = _nx + 1;
186: #pragma omp parallel for shared(_ny)
187:     for (int j = 0; j <= _ny; ++j)
188:     {
189:         int k = j * dx;
190:         for (int i = 0; i <= _nx; ++i, ++k)
191:         {
192:             f[k] = 1.0;
```

```
  193:         }
  194:     }
  195:
  196: }
  197:
  198:
  199: std::vector<int> Mesh_2d_3_square::Index_DirichletNodes() const
  200: {
  201:     int const dx = 1,
  202:               dy = _nx + 1,
  203:               bl  = 0,
  204:               br  = _nx,
  205:               tl  = _ny * (_nx + 1),
  206:               tr  = (_ny + 1) * (_nx + 1) - 1;
  207:     int const start[4] = { bl, br, tl, bl},
  208:                         end[4] = { br, tr, tr, tl},
  209:                                  step[4] = { dx, dy, dx, dy};
  210:
  211:     vector<int> idx(0);
  212: #pragma omp parallel for shared(_neigh) reduction(VecAppend: idx)
  213:     for (int j = 0; j < 4; j++)
  214:     {
  215:         if (_neigh[j] < 0)
  216:         {
  217:             for (int i = start[j]; i <= end[j]; i += step[j])
  218:             {
  219:                 idx.push_back(i);        // node i is Dirichlet node
  220:             }
  221:         }
  222:     }
  223:     //    remove multiple elements
  224:     sort(idx.begin(), idx.end());                           // sort
  225:     idx.erase( unique(idx.begin(), idx.end()), idx.end() ); // remove duplicate data
  226:
  227:     return idx;
  228: }
  229:
  230: void Mesh_2d_3_square::SaveVectorP(std::string const &name, vector<double> const &u)
 const
  231: {
  232: //  construct the file name for subdomain myid
  233:     const string tmp( std::to_string(_myid / 100) + to_string((_myid % 100) / 10) +
to_string(_myid % 10) );
  234:
  235:     const string namep = name + "." + tmp;
  236:     ofstream ff(namep.c_str());
  237:     ff.precision(6);
  238:     ff.setf(ios::fixed, ios::floatfield);
  239:
  240:     // assumes tensor product grid in unit square; rowise numbered (as generated in
class constructor)
  241:     // output is provided for tensor product grid visualization ( similar to Matlab-
surf() )
  242:     auto const &xc = GetCoords();
  243:     int k = 0;
  244:     for (int j = 0; j <= _ny; ++j)
  245:     {
  246:         for (int i = 0; i <= _nx; ++i, ++k)
  247:             ff << xc[2 * k + 0] << "   " << xc[2 * k + 1] << "   " << u[k] << endl;
  248:         ff << endl;
  249:     }
  250:
  251:     ff.close();
  252:     return;
  253: }
  254:
  255: void Mesh_2d_3_square::GetCoordsInRectangle(int const nx, int const ny,
  256:         double const xl, double const xr, double const yb, double const yt,
```

```
  257:         double xc[])
  258: {
  259:     const double hx = (xr - xl) / nx,
  260:                  hy = (yt - yb) / ny;
  261:
  262:
  263:     int k = 0;
  264: #pragma omp parallel for shared(nx, ny, hx, hy, yb, xl, xc, k)
  265:     for (int j = 0; j <= ny; ++j)
  266:     {
  267:         const double y0 = yb + j * hy;
  268:         #pragma omp critical
  269:         for (int i = 0; i <= nx; ++i, k += 2)
  270:         {
  271:             xc[k  ] = xl + i * hx;
  272:             xc[k + 1] = y0;
  273:         }
  274:     }
  275:
  276:     return;
  277: }
  278:
  279: void Mesh_2d_3_square::GetConnectivityInRectangle(int const nx, int const ny, int ia
[])
  280: {
  281:     const int dx = nx + 1;
  282:     int k  = 0;
  283:     int l  = 0;
  284:
  285:     for (int j = 0; j < ny; ++j, ++k)
  286:     {
  287:         for (int i = 0; i < nx; ++i, ++k)
  288:         {
  289:             ia[l  ] = k;
  290:             ia[l + 1] = k + 1;
  291:             ia[l + 2] = k + dx + 1;
  292:             l += 3;
  293:             ia[l  ] = k;
  294:             ia[l + 1] = k + dx;
  295:             ia[l + 2] = k + dx + 1;
  296:             l += 3;
  297:         }
  298:     }
  299:     return;
  300: }
  301:
  302: // ################### still some old code (--> MPI) ############################
  303:
  304:
  305: //  Copies the values of w corresponding to the boundary
  306: //  South (ib==1), East (ib==2), North (ib==3), West (ib==4)
  307:
  308: void GetBound(int const ib, int const nx, int const ny, double const w[], double s[]
)
  309: {
  310:     const int //dx = 1,
  311:     dy = nx + 1,
  312:     bl  = 0,
  313:     br  = nx,
  314:     tl  = ny * (nx + 1),
  315:     tr  = (ny + 1) * (nx + 1) - 1;
  316:     switch (ib)
  317:     {
  318:         case 1:
  319:         {
  320:             for (int i = bl, j = 0; i <= br; ++i, ++j)
  321:                 s[j] = w[i];
  322:             break;
```

```
  323:         }
  324:         case 3:
  325:         {
  326:             for (int i = tl, j = 0; i <= tr; ++i, ++j)
  327:                 s[j] = w[i];
  328:             break;
  329:         }
  330:         case 4:
  331:         {
  332:             for (int i = bl, j = 0; i <= tl; i += dy, ++j)
  333:                 s[j] = w[i];
  334:             break;
  335:         }
  336:         case 2:
  337:         {
  338:             for (int i = br, j = 0; i <= tr; i += dy, ++j)
  339:                 s[j] = w[i];
  340:             break;
  341:         }
  342:         default:
  343:         {
  344:             cout << endl << "Wrong parameter  ib in " << __FILE__ << ":" << __LINE__
 << endl;
  345:         }
  346:     }
  347:     return;
  348: }
  349:
  350: // ------------------------------------------------------------------------------
------------------------
  351: // Computes w:  = w + s at nodes on  the boundary
  352: // South (ib == 1), East (ib == 2), North (ib == 3), West (ib == 4)
  353:
  354: void AddBound(int const ib, int const nx, int const ny, double w[], double const s[]
)
  355: {
  356:     int const dy = nx + 1,
  357:               bl  = 0,
  358:               br  = nx,
  359:               tl  = ny * (nx + 1),
  360:               tr  = (ny + 1) * (nx + 1) - 1;
  361:     switch (ib)
  362:     {
  363:         case 1:
  364:         {
  365:             for (int i = bl, j = 0; i <= br; ++i, ++j)
  366:                 w[i] += s[j];
  367:             break;
  368:         }
  369:         case 3:
  370:         {
  371:             for (int i = tl, j = 0; i <= tr; ++i, ++j)
  372:                 w[i] += s[j];
  373:             break;
  374:         }
  375:         case 4:
  376:         {
  377:             for (int i = bl, j = 0; i <= tl; i += dy, ++j)
  378:                 w[i] += s[j];
  379:             break;
  380:         }
  381:         case 2:
  382:         {
  383:             for (int i = br, j = 0; i <= tr; i += dy, ++j)
  384:                 w[i] += s[j];
  385:             break;
  386:         }
  387:         default:
```

```
  388:         {
  389:             cout << endl << "Wrong parameter  ib in " << __FILE__ << ":" << __LINE__
 << endl;
  390:         }
  391:     }
  392:     return;
  393: }
  394:
  395:
  396: // ##################################################################
  397:
  398: Mesh_2d_3_matlab::Mesh_2d_3_matlab(string const &fname)
  399:     : Mesh(2, 3, 3),  // two dimensions, 3 vertices, 3 dofs
  400:       bedges(0)
  401: {
  402:     ifstream ifs(fname);
  403:     if (!(ifs.is_open() && ifs.good()))
  404:     {
  405:         cerr << "Mesh_2d_3_matlab: Error cannot open file " << fname << endl;
  406:         assert(ifs.is_open());
  407:     }
  408:
  409:     int const OFFSET(1);              // Matlab to C indexing
  410:     cout << "ASCI file  " << fname << "  opened" << endl;
  411:
  412:     // Read some mesh constants
  413:     int nnode, ndim, nelem, nvert_e;
  414:     ifs >> nnode >> ndim >> nelem >> nvert_e;
  415:     cout << nnode << "  " << ndim << "  " << nelem << "  " << nvert_e << endl;
  416:     assert(ndim == 2 && nvert_e == 3);
  417:
  418:     // Allocate memory
  419:     Resize_Coords(nnode, ndim);                  // coordinates in 2D [nnode][ndim]
  420:     Resize_Connectivity(nelem, nvert_e);         // connectivity matrix [nelem][nvert
]
  421:
  422:     // Read ccordinates
  423:     auto &xc = GetCoords();
  424:     for (int k = 0; k < nnode * ndim; ++k)
  425:     {
  426:         ifs >> xc[k];
  427:     }
  428:
  429:     // Read connectivity
  430:     auto &ia = GetConnectivity();
  431:     for (int k = 0; k < nelem * nvert_e; ++k)
  432:     {
  433:         ifs >> ia[k];
  434:         ia[k] -= OFFSET;                // Matlab to C indexing
  435:     }
  436:
  437:     // additional read of boundary information (only start/end point)
  438:     int nbedges;
  439:     ifs >> nbedges;
  440:
  441:     bedges.resize(nbedges * 2);
  442:     for (int k = 0; k < nbedges * 2; ++k)
  443:     {
  444:         ifs >> bedges[k];
  445:         bedges[k] -= OFFSET;            // Matlab to C indexing
  446:     }
  447:
  448:     return;
  449: }
  450:
  451: // binary
  452: //{
  453: //ifstream ifs(fname, ios_base::in | ios_base::binary);
```

```
  454: //if(!(ifs.is_open() && ifs.good()))
  455: //{
  456: //cerr << "ReadBinMatrix: Error cannot open file " << file << endl;
  457: //assert(ifs.is_open());
  458: //}
  459: //cout << "ReadBinMatrix: file opened" << file << endl;
  460:
  461:
  462: //}
  463:
  464: // binaryIO.cpp
  465: //void read_binMatrix(const string& file, vector<int> &cnt, vector<int> &col, vector
<double> &ele)
  466: //{
  467:
  468: //ifstream ifs(file, ios_base::in | ios_base::binary);
  469:
  470: //if(!(ifs.is_open() && ifs.good()))
  471: //{
  472: //cerr << "ReadBinMatrix: Error cannot open file " << file << endl;
  473: //assert(ifs.is_open());
  474: //}
  475: //cout << "ReadBinMatrix: Opened file " << file << endl;
  476:
  477: //int _size;
  478:
  479: //ifs.read(reinterpret_cast<char*>(&_size), sizeof(int));   // old: ifs.read((char*)
&_size, sizeof(int));
  480: //cnt.resize(_size);
  481: //cout << "ReadBinMatrix: cnt size: " << _size << endl;
  482:
  483: //ifs.read((char*)&_size, sizeof(int));
  484: //col.resize(_size);
  485: //cout << "ReadBinMatrix: col size: " << _size << endl;
  486:
  487: //ifs.read((char*)&_size, sizeof(int));
  488: //ele.resize(_size);
  489: //cout << "ReadBinMatrix: ele size: " << _size << endl;
  490:
  491:
  492: //ifs.read((char*)cnt.data(), cnt.size() * sizeof(int));
  493: //ifs.read((char*)col.data(), col.size() * sizeof(int));
  494: //ifs.read((char*)ele.data(), ele.size() * sizeof(double));
  495:
  496: //ifs.close();
  497: //cout << "ReadBinMatrix: Finished reading matrix.." << endl;
  498:
  499: //}
  500:
  501:
  502: std::vector<int> Mesh_2d_3_matlab::Index_DirichletNodes() const
  503: {
  504:     vector<int> idx(bedges);                                  // copy
  505:
  506:     sort(idx.begin(), idx.end());                             // sort
  507:     idx.erase( unique(idx.begin(), idx.end()), idx.end() ); // remove duplicate data
  508:
  509:     return idx;
  510: }
  511:
  512:
  513:
  514:
  515:
  516:
  517:
  518:
  519:
```

```
  520:
  521:
  522:
  523:
  524:
  525:
  526:
  527:
  528:
  529:
```

```
    1: #ifndef GEOM_FILE
    2: #define GEOM_FILE
    3: #include <array>
    4: #include <functional>             // function; C++11
    5: #include <string>
    6: #include <vector>
    7:
    8: // Declare append reduction operation for STL-Vectors
    9: #pragma omp declare reduction(VecAppend : std::vector<int>  : omp_out.insert(omp_out
.end(), omp_in.begin(), omp_in.end())) \
   10:  initializer (omp_priv=omp_orig)
   11:
   12:
   13: /**
   14:  * Basis class for finite element meshes.
   15:  */
   16: class Mesh
   17: {
   18: public:
   19:     /**
   20:      * Constructor initializing the members with default values.
   21:      *
   22:      * @param[in] ndim  space dimensions (dimension for coordinates)
   23:      * @param[in] nvert_e  number of vertices per element (dimension for connectivi
ty)
   24:      * @param[in] ndof_e   degrees of freedom per element (= @p nvert_e for linear
elements)
   25:      */
   26:     explicit Mesh(int ndim, int nvert_e = 0, int ndof_e = 0);
   27:
   28:     /**
   29:      * Destructor.
   30:      *
   31:      * See clang warning on
   32:      * <a href="https://stackoverflow.com/questions/28786473/clang-no-out-of-line-vi
rtual-method-definitions-pure-abstract-c-class/40550578">weak-vtables</a>.
   33:      */
   34:     virtual ~Mesh();
   35:
   36:     /**
   37:      * Number of finite elements in (sub)domain.
   38:      * @return number of elements.
   39:      */
   40:     int Nelems() const
   41:     {
   42:         return _nelem;
   43:     }
   44:
   45:     /**
   46:      * Global number of vertices for each finite element.
   47:      * @return number of vertices per element.
   48:      */
   49:     int NverticesElements() const
   50:     {
   51:         return _nvert_e;
   52:     }
   53:
   54:     /**
   55:      * Global number of degrees of freedom (dof) for each finite element.
   56:      * @return degrees of freedom per element.
   57:      */
   58:     int NdofsElement() const
   59:     {
   60:         return _ndof_e;
   61:     }
   62:
   63:     /**
   64:      * Number of vertices in mesh.
```

```
    65:       * @return number of vertices.
    66:       */
    67:      int Nnodes() const
    68:      {
    69:          return _nnode;
    70:      }
    71:
    72:      /**
    73:       * Space dimension.
    74:       * @return number of dimensions.
    75:       */
    76:      int Ndims() const
    77:      {
    78:          return _ndim;
    79:      }
    80:
    81:      /**
    82:       * (Re-)Allocates memory for the element connectivity and redefines the appropri
ate dimensions.
    83:        *
    84:       * @param[in] nelem    number of elements
    85:        * @param[in] nvert_e  number of vertices per element
    86:       */
    87:      void Resize_Connectivity(int nelem, int nvert_e)
    88:      {
    89:          SetNelem(nelem);                // number of elements
    90:          SetNverticesElement(nvert_e);  // vertices per element
    91:          _ia.resize(nelem * nvert_e);
    92:      }
    93:
    94:      /**
    95:       * Read connectivity information (g1,g2,g3)_i.
    96:       * @return convectivity vector [nelems*ndofs].
    97:       */
    98:      const std::vector<int>  &GetConnectivity() const
    99:      {
   100:          return _ia;
   101:      }
   102:
   103:      /**
   104:       * Access/Change connectivity information (g1,g2,g3)_i.
   105:       * @return convectivity vector [nelems*ndofs].
   106:       */
   107:      std::vector<int>  &GetConnectivity()
   108:      {
   109:          return _ia;
   110:      }
   111:
   112:      /**
   113:       * (Re-)Allocates memory for the element connectivity and redefines the appropri
ate dimensions.
   114:        *
   115:       * @param[in] nnodes    number of nodes
   116:        * @param[in] ndim      space dimension
   117:       */
   118:      void Resize_Coords(int nnodes, int ndim)
   119:      {
   120:          SetNnode(nnodes);        // number of nodes
   121:          SetNdim(ndim);           // space dimension
   122:          _xc.resize(nnodes * ndim);
   123:      }
   124:
   125:      /**
   126:       * Read coordinates of vertices (x,y)_i.
   127:       * @return coordinates vector [nnodes*2].
   128:       */
   129:      const std::vector<double> &GetCoords() const
   130:      {
```

```
  131:         return _xc;
  132:     }
  133:
  134:     /**
  135:      * Access/Change coordinates of vertices (x,y)_i.
  136:      * @return coordinates vector [nnodes*2].
  137:      */
  138:     std::vector<double> &GetCoords()
  139:     {
  140:         return _xc;
  141:     }
  142:
  143:     /**
  144:      * Calculate values in vector @p v via function @p func(x,y)
  145:      * @param[in] v     vector
  146:       * @param[in] func  function of (x,y) returning a double value.
  147:      */
  148:     void SetValues(std::vector<double> &v, const std::function<double(double, double
)> &func) const;
  149:
  150:     /**
  151:      * Prints the information for a finite element mesh
  152:      */
  153:     void Debug() const;
  154:
  155:     /**
  156:      * Determines the indices of those vertices with Dirichlet boundary conditions
  157:      * @return index vector.
  158:      */
  159:     virtual std::vector<int> Index_DirichletNodes() const = 0;
  160:
  161:     /**
  162:      * Write vector @p v toghether with its mesh information to an ASCii file @p fna
me.
  163:       *
  164:       * The data are written in C-style.
  165:       *
  166:       * @param[in] fname  file name
  167:       * @param[in] v      vector
  168:      */
  169:     void Write_ascii_matlab(std::string const &fname, std::vector<double> const &v)
const;
  170:
  171:     /**
  172:      * Visualize @p v together with its mesh information via matlab or octave.
  173:       *
  174:       * Comment/uncomment those code lines in method Mesh:Visualize (geom.cpp)
  175:       * that are supported on your system.
  176:       *
  177:       * @param[in] v    vector
  178:       *
  179:       * @warning matlab files ascii_read_meshvector.m  visualize_results.m
  180:       *          must be in the executing directory.
  181:      */
  182:     void Visualize(std::vector<double> const &v) const;
  183:
  184:
  185: protected:
  186:     void SetNelem(int nelem)
  187:     {
  188:         _nelem = nelem;
  189:     }
  190:
  191:     void SetNverticesElement(int nvert)
  192:     {
  193:         _nvert_e = nvert;
  194:     }
  195:
```

```
196:     void SetNdofsElement(int ndof)
197:     {
198:         _ndof_e = ndof;
199:     }
200:
201:     void SetNnode(int nnode)
202:     {
203:         _nnode = nnode;
204:     }
205:
206:     void SetNdim(int ndim)
207:     {
208:         _ndim = ndim;
209:     }
210:
211: private:
212:     int _nelem;          //!< number elements
213:     int _nvert_e;        //!< number of vertices per element
214:     int _ndof_e;         //!< degrees of freedom (d.o.f.) per element
215:     int _nnode;          //!< number nodes/vertices
216:     int _ndim;           //!< space dimension of the problem (1, 2, or 3)
217:     std::vector<int> _ia;    //!< element connectivity
218:     std::vector<double> _xc; //!< coordinates
219: };
220:
221: /**
222:  * 2D finite element mesh of the square consiting of linear triangular elements.
223:  */
224: class Mesh_2d_3_square: public Mesh
225: {
226: public:
227:     /**
228:      * Generates the f.e. mesh for the unit square.
229:      *
230:      * @param[in] nx    number of discretization intervals in x-direction
231:      * @param[in] ny    number of discretization intervals in y-direction
232:      * @param[in] myid  my MPI-rank / subdomain
233:      * @param[in] procx number of ranks/subdomains in x-direction
234:      * @param[in] procy number of processes in y-direction
235:     */
236:     Mesh_2d_3_square(int nx, int ny, int myid = 0, int procx = 1, int procy = 1);
237:
238:     /**
239:      * Destructor
240:      */
241:     ~Mesh_2d_3_square() override
242:     {}
243:
244:     /**
245:      * Set solution vector based on a tensor product grid in the rectangle.
246:      * @param[in] u solution vector
247:      */
248:     void SetU(std::vector<double> &u) const;
249:
250:     /**
251:      * Set right hand side (rhs) vector on a tensor product grid in the rectangle.
252:      * @param[in] f rhs vector
253:      */
254:     void SetF(std::vector<double> &f) const;
255:
256:     /**
257:      * Determines the indices of those vertices with Dirichlet boundary conditions
258:      * @return index vector.
259:      */
260:     std::vector<int> Index_DirichletNodes() const override;
261:
262:     /**
263:       * Stores the values of vector @p u of (sub)domain into a file @p name for furt
```

```
her processing in gnuplot.
  264:       * The file stores rowise the x- and y- coordinates together with the value fro
m  @p u .
  265:       * The domain [@p xl, @p xr] x [@p yb, @p yt] is discretized into @p nx x @p ny
 intervals.
  266:       *
  267:       * @param[in] name basename of file name (file name will be extended by the ran
k number)
  268:       * @param[in] u    local vector
  269:       *
  270:       * @warning   Assumes tensor product grid in unit square; rowise numbered
  271:       *            (as generated in class constructor).
  272:       *            The output is provided for tensor product grid visualization
  273:       *            ( similar to Matlab-surf() ).
  274:       *
  275:       * @see Mesh_2d_3_square
  276:       */
  277:     void SaveVectorP(std::string const &name, std::vector<double> const &u) const;
  278:
  279:     // here will still need to implement in the class
  280:     //  GetBound(), AddBound()
  281:     //  or better a generalized way with indices and their appropriate ranks for MPI
 communication
  282:
  283: private:
  284:     /**
  285:       * Determines the coordinates of the dicretization nodes of the domain [@p xl,
@p xr] x [@p yb, @p yt]
  286:       * which is discretized into @p nx x @p ny intervals.
  287:       *
  288:       * @param[in] ny   number of discretization intervals in y-direction
  289:       * @param[in] xl   x-coordinate of left boundary
  290:       * @param[in] xr   x-coordinate of right boundary
  291:       * @param[in] yb   y-coordinate of lower boundary
  292:       * @param[in] yt   y-coordinate of upper boundary
  293:       * @param[out] xc  coordinate vector of length 2n with x(2*k,2*k+1) as coodinat
es of node k
  294:       */
  295:
  296:     void GetCoordsInRectangle(int nx, int ny, double xl, double xr, double yb, doubl
e yt,
  297:                               double xc[]);
  298:     /**
  299:       * Determines the element connectivity of linear triangular elements of a FEM d
iscretization
  300:       * of a rectangle using @p nx x @p ny equidistant intervals for discretization.
  301:       * @param[in] nx   number of discretization intervals in x-direction
  302:       * @param[in] ny   number of discretization intervals in y-direction
  303:       * @param[out] ia  element connectivity matrix with ia(3*s,3*s+1,3*s+2) as node
 numbers od element s
  304:       */
  305:     void GetConnectivityInRectangle(int nx, int ny, int ia[]);
  306:
  307: private:
  308:     int _myid;          //!< my MPI rank
  309:     int _procx;         //!< number of MPI ranks in x-direction
  310:     int _procy;         //!< number of MPI ranks in y-direction
  311:     std::array<int, 4> _neigh; //!< MPI ranks of neighbors (negative: no neighbor bu
t b.c.)
  312:     int _color;         //!< red/black coloring (checker board) of subdomains
  313:
  314:     double _xl;         //!< x coordinate of lower left  corner of square
  315:     double _xr;         //!< x coordinate of lower right corner of square
  316:     double _yb;         //!< y coordinate or lower left  corner of square
  317:     double _yt;         //!< y coordinate of upper right corner of square
  318:     int    _nx;         //!< number of intervals in x-direction
  319:     int    _ny;         //!< number of intervals in y-direction
  320: };
```

```
  321:
  322: // ################### still some old code (--> MPI) ############################
  323: /**
  324:  * Copies the values of @p w corresponding to boundary @p ib
  325:  * onto vector s.  South (ib==1), East (ib==2), North (ib==3), West (ib==4).
  326:  * The vector @p s has to be long enough!!
  327:  * @param[in] ib    my local boundary
  328:  * @param[in] nx    number of discretization intervals in x-direction
  329:  * @param[in] ny    number of discretization intervals in y-direction
  330:  * @param[in] w     vector for all nodes of local discretization
  331:  * @param[out] s    short vector with values on boundary @p ib
  332: */
  333: // GH_NOTE: Absicherung bei s !!
  334: void GetBound(int ib, int nx, int ny, double const w[], double s[]);
  335:
  336:
  337: /**
  338:  * Computes @p w := @p w + @p s  at the interface/boundary nodes on the
  339:  * boundary @p ib .  South (ib==1), East (ib==2), North (ib==3), West (ib==4)
  340:  * @param[in] ib    my local boundary
  341:  * @param[in] nx    number of discretization intervals in x-direction
  342:  * @param[in] ny    number of discretization intervals in y-direction
  343:  * @param[in,out] w vector for all nodes of local discretization
  344:  * @param[in] s     short vector with values on boundary @p ib
  345: */
  346: void AddBound(int ib, int nx, int ny, double w[], double const s[]);
  347:
  348: // ################### Mesh from Matlab ############################
  349: /**
  350:  * 2D finite element mesh of the square consiting of linear triangular elements.
  351:  */
  352: class Mesh_2d_3_matlab: public Mesh
  353: {
  354: public:
  355:     /**
  356:      * Reads mesh data from a binary file.
  357:      *
  358:      * File format, see ascii_write_mesh.m
  359:      *
  360:      * @param[in] fname file name
  361:     */
  362:     explicit Mesh_2d_3_matlab(std::string const &fname);
  363:
  364:     /**
  365:      * Determines the indices of those vertices with Dirichlet boundary conditions.
  366:      * @return index vector.
  367:       *
  368:       * @warning All boundary nodes are considered as Dirchlet nodes.
  369:      */
  370:     std::vector<int> Index_DirichletNodes() const override;
  371:
  372: private:
  373:     /**
  374:      * Determines the indices of those vertices with Dirichlet boundary conditions
  375:      * @return index vector.
  376:      */
  377:     int Nnbedges() const
  378:     {
  379:         return static_cast<int>(bedges.size());
  380:     }
  381:
  382:     std::vector<int> bedges;     //!< boundary edges [nbedges][2] storing start/end
vertex
  383:
  384: };
  385:
  386: #endif
```

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

#include <algorithm>
#include <cassert>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <list>
#include <vector>
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 !!
```

```
   68:             {
   69:                 cout << "Error in AddElem: (" << ii << "," << jj << ") ["
   70:                      << ial[0] << "," << ial[1] << "," << ial[2] << "]\n";
   71:                 assert(!not_found);
   72:             }
   73: #endif
   74:             sk[ip - 1] += ske[i][j];
   75:         }
   76:         f[ii] += fe[i];
   77:     }
   78: }
   79:
   80:
   81: // ----------------------------------------------------------------------------
   82:
   83:
   84:
   85:
   86: // ############################################################################
   87:
   88: CRS_Matrix::CRS_Matrix(Mesh const &mesh)
   89:     : _mesh(mesh), _nrows(0), _nnz(0), _id(0), _ik(0), _sk(0)
   90: {
   91:     Derive_Matrix_Pattern();
   92:     return;
   93: }
   94:
   95: void CRS_Matrix::Derive_Matrix_Pattern()
   96: {
   97:     int const nelem(_mesh.Nelems());
   98:     int const ndof_e(_mesh.NdofsElement());
   99:     auto const &ia(_mesh.GetConnectivity());
  100: //  Determine the number of matrix rows
  101:     _nrows = *max_element(ia.cbegin(), ia.cbegin() + ndof_e * nelem);
  102:     ++_nrows;                                   // node numberng: 0 ... nnode-1
  103:     assert(*min_element(ia.cbegin(), ia.cbegin() + ndof_e * nelem) == 0); // numberi
ng starts with 0 ?
  104:
  105: //  Collect for each node those nodes it is connected to (multiple entries)
  106: //  Detect the neighboring nodes
  107:     vector< list<int> > cc(_nrows);             //  cc[i] is the  list of nodes a no
de i is connected to
  108:     for (int i = 0; i < nelem; ++i)
  109:     {
  110:         int const idx = ndof_e * i;
  111:         for (int k = 0; k < ndof_e; ++k)
  112:         {
  113:             list<int> &cck = cc.at(ia[idx + k]);
  114:             cck.insert( cck.end(), ia.cbegin() + idx, ia.cbegin() + idx + ndof_e );
  115:         }
  116:     }
  117: //  Delete the multiple entries
  118:     _nnz = 0;
  119:     for (auto &it : cc)
  120:     {
  121:         it.sort();
  122:         it.unique();
  123:         _nnz += static_cast<int>(it.size());
  124:         // cout << it.size() << " :: "; copy(it->begin(),it->end(), ostream_iterator
<int,char>(cout,"  ")); cout << endl;
  125:     }
  126:
  127: // CSR data allocation
  128:     _id.resize(_nrows + 1);                   // Allocate memory for CSR row pointer
  129:     _ik.resize(_nnz);                         // Allocate memory for CSR column index
 vector
  130:
  131: //  copy CSR data
```

```
  132:     _id[0] = 0;                                  // begin of first row
  133:     for (size_t i = 0; i < cc.size(); ++i)
  134:     {
  135:         //cout << i << "   " << nid.at(i) << endl;;
  136:         const list<int> &ci = cc.at(i);
  137:         const auto nci = static_cast<int>(ci.size());
  138:         _id[i + 1] = _id[i] + nci; // begin of next line
  139:         copy(ci.begin(), ci.end(), _ik.begin() + _id[i] );
  140:     }
  141:
  142:     assert(_nnz == _id[_nrows]);
  143:     _sk.resize(_nnz);                        // Allocate memory for CSR column index v
ector
  144:     return;
  145: }
  146:
  147:
  148: void CRS_Matrix::Debug() const
  149: {
  150: //  ID points to first entry of row
  151: //  no symmetry assumed
  152:     cout << "\nMatrix  (nnz = " << _id[_nrows] << ")\n";
  153:
  154:     for (int row = 0; row < _nrows; ++row)
  155:     {
  156:         cout << "Row " << row << " : ";
  157:         int const id1 = _id[row];
  158:         int const id2 = _id[row + 1];
  159:         for (int j = id1; j < id2; ++j)
  160:         {
  161:             cout.setf(ios::right, ios::adjustfield);
  162:             cout << "[" << setw(2) << _ik[j] << "]  " << setw(4) << _sk[j] << "  ";
  163:         }
  164:         cout << endl;
  165:     }
  166:     return;
  167: }
  168:
  169: void CRS_Matrix::CalculateLaplace(vector<double> &f)
  170: {
  171:     assert(_mesh.NdofsElement() == 3);               // only for triangular, linear
elements
  172:     //cout << _nnz << " vs. " << _id[_nrows] << "  " << _nrows<< endl;
  173:     assert(_nnz == _id[_nrows]);
  174:
  175: #pragma omp parallel for shared(_nrows, _sk)
  176:     for (int k = 0; k < _nrows; ++k)
  177:     {
  178:         _sk[k] = 0.0;
  179:     }
  180: #pragma omp parallel for shared(_nrows, f)
  181:     for (int k = 0; k < _nrows; ++k)
  182:     {
  183:         f[k] = 0.0;
  184:     }
  185:
  186:
  187:     //  Loop over all elements
  188:     auto const nelem = _mesh.Nelems();
  189:     auto const &ia   = _mesh.GetConnectivity();
  190:     auto const &xc   = _mesh.GetCoords();
  191:
  192: #pragma omp parallel for
  193:     for (int i = 0; i < nelem; ++i)
  194:     {
  195:         double ske[3][3], fe[3];
  196:         CalcElem(ia.data() + 3 * i, xc.data(), ske, fe);
  197:         AddElem_3(ia.data() + 3 * i, ske, fe, f);
```

```
  198:     }
  199:     //Debug();
  200:
  201:     return;
  202: }
  203:
  204: void CRS_Matrix::ApplyDirichletBC(std::vector<double> const &u, std::vector<double>
&f)
  205: {
  206:     double const PENALTY = 1e6;
  207:     auto const idx = _mesh.Index_DirichletNodes();
  208:     int const nidx = static_cast<int>(idx.size());
  209:
  210: #pragma omp parallel for shared(f)
  211:     for (int row = 0; row < nidx; ++row)
  212:     {
  213:         int const k = idx[row];
  214:         int const id1 = fetch(k, k); // Find diagonal entry of row
  215:         assert(id1 >= 0);
  216:         _sk[id1] += PENALTY;            // matrix weighted scaling feasible
  217:         f[k] += PENALTY * u[k];
  218:     }
  219:
  220:     return;
  221: }
  222:
  223: void CRS_Matrix::GetDiag(vector<double> &d) const
  224: {
  225:     assert( _nrows == static_cast<int>(d.size()) );
  226:
  227: #pragma omp parallel for shared(d, _nrows)
  228:     for (int row = 0; row < _nrows; ++row)
  229:     {
  230:         const int ia = fetch(row, row); // Find diagonal entry of row
  231:         assert(ia >= 0);
  232:         d[row] = _sk[ia];
  233:     }
  234:     return;
  235: }
  236:
  237: bool CRS_Matrix::Compare2Old(int nnode, int const id[], int const ik[], double const
 sk[]) const
  238: {
  239:     bool bn = (nnode == _nrows);     // number of rows
  240:     if (!bn)
  241:     {
  242:         cout << "#########   Error: " << "number of rows" << endl;
  243:     }
  244:
  245:     bool bz = (id[nnode] == _nnz);   // number of non zero elements
  246:     if (!bz)
  247:     {
  248:         cout << "#########   Error: " << "number of non zero elements" << endl;
  249:     }
  250:
  251:     bool bd = equal(id, id + nnode + 1, _id.cbegin()); // row starts
  252:     if (!bd)
  253:     {
  254:         cout << "#########   Error: " << "row starts" << endl;
  255:     }
  256:
  257:     bool bk = equal(ik, ik + id[nnode], _ik.cbegin()); // column indices
  258:     if (!bk)
  259:     {
  260:         cout << "#########   Error: " << "column indices" << endl;
  261:     }
  262:
  263:     bool bv = equal(sk, sk + id[nnode], _sk.cbegin()); // values
```

```
264:     if (!bv)
265:     {
266:         cout << "#########   Error: " << "values" << endl;
267:     }
268:
269:     return bn && bz && bd && bk && bv;
270: }
271:
272:
273: void CRS_Matrix::Mult(vector<double> &w, vector<double> const &u) const
274: {
275:     assert( _nrows == static_cast<int>(w.size()) );
276:     assert( w.size() == u.size() );
277:
278: #pragma omp parallel for shared(w)
279:     for (int row = 0; row < _nrows; ++row)
280:     {
281:         double wi = 0.0;
282:         for (int ij = _id[row]; ij < _id[row + 1]; ++ij)
283:         {
284:             wi += _sk[ij] * u[ _ik[ij] ];
285:         }
286:         w[row] = wi;
287:     }
288:     return;
289: }
290:
291: void CRS_Matrix::Defect(vector<double> &w,
292:                         vector<double> const &f, vector<double> const &u) const
293: {
294:     assert( _nrows == static_cast<int>(w.size()) );
295:     assert( w.size() == u.size() && u.size() == f.size() );
296:
297:
298: #pragma omp parallel for shared(w, f, u, _nrows)
299:     for (int row = 0; row < _nrows; ++row)
300:     {
301:         double wi = f[row];
302:         for (int ij = _id[row]; ij < _id[row + 1]; ++ij)
303:         {
304:             wi -= _sk[ij] * u[ _ik[ij] ];
305:         }
306:         w[row] = wi;
307:     }
308:     return;
309: }
310:
311: int CRS_Matrix::fetch(int const row, int const col) const
312: {
313:     int const id2 = _id[row + 1];     // end   and
314:     int       ip  = _id[row];         // start of recent row (global index)
315:
316:     while (ip < id2 && _ik[ip] != col)  // find index col (global index)
317:     {
318:         ++ip;
319:     }
320:     if (ip >= id2)
321:     {
322:         ip = -1;
323: #ifndef NDEBUG                  // compiler option -DNDEBUG switches off the check
324:         cout << "No column  " << col << "  in row  " << row << endl;
325:         assert(ip >= id2);
326: #endif
327:     }
328:     return ip;
329: }
330:
331:
```

```
  332: // general routine for lin. triangular elements,
  333: // non-symm. matrix
  334: // node numbering in element:  a s c e n d i n g   indices !!
  335: void CRS_Matrix::AddElem_3(int const ial[3], double const ske[3][3], double const fe
[3], vector<double> &f)
  336: {
  337:     for (int i = 0; i < 3; ++i)
  338:     {
  339:         const int ii  = ial[i];           // row ii (global index)
  340:         for (int j = 0; j < 3; ++j)       // no symmetry assumed
  341:         {
  342:             const int jj = ial[j];        // column jj (global index)
  343:             int ip = fetch(ii, jj);       // find column entry jj in row ii
  344: #ifndef NDEBUG                  // compiler option -DNDEBUG switches off the check
  345:             if (ip < 0)         // no entry found !!
  346:             {
  347:                 cout << "Error in AddElem: (" << ii << "," << jj << ") ["
  348:                      << ial[0] << "," << ial[1] << "," << ial[2] << "]\n";
  349:                 assert(ip >= 0);
  350:             }
  351: #endif
  352:             _sk[ip] += ske[i][j];
  353:         }
  354:         f[ii] += fe[i];
  355:     }
  356: }
  357:
```

data race

```
    1: #ifndef GETMATRIX_FILE
    2: #define GETMATRIX_FILE
    3:
    4: #include "geom.h"
    5: #include <vector>
    6:
    7: /**
    8:  * Calculates the element stiffness matrix @p ske and the element load vector @p fe
    9:  * of one triangular element with linear shape functions.
   10:  * @param[in]   ial     node indices of the three element vertices
   11:  * @param[in]   xc      vector of node coordinates with x(2*k,2*k+1) as coodinates o
f node k
   12:  * @param[out] ske      element stiffness matrix
   13:  * @param[out] fe       element load vector
   14:  */
   15: void CalcElem(int const ial[3], double const xc[], double ske[3][3], double fe[3]);
   16:
   17: /**
   18:  * Adds the element stiffness matrix @p ske and the element load vector @p fe
   19:  * of one triangular element with linear shape functions to the appropriate position
s in
   20:  * the symmetric stiffness matrix, stored as CSR matrix K(@p sk,@p id, @p ik)
   21:  *
   22:  * @param[in] ial   node indices of the three element vertices
   23:  * @param[in] ske       element stiffness matrix
   24:  * @param[in] fe        element load vector
   25:  * @param[out] sk       vector non-zero entries of CSR matrix
   26:  * @param[in] id        index vector containing the first entry in a CSR row
   27:  * @param[in] ik        column index vector of CSR matrix
   28:  * @param[out] f        distributed local vector storing the right hand side
   29:  *
   30:  * @warning Algorithm requires indices in connectivity @p ial in ascending order.
   31:  *          Currently deprecated.
   32: */
   33: void AddElem(int const ial[3], double const ske[3][3], double const fe[3],
   34:              int const id[], int const ik[], double sk[], double f[]);
   35:
   36:
   37: // ########################################################################
   38: /**
   39:  * Square matrix in CRS format (compressed row storage; also named CSR),
   40:  * see an <a href="https://en.wikipedia.org/wiki/Sparse_matrix">introduction</a>.
   41:  */
   42: class CRS_Matrix
   43: {
   44:     public:
   45:        /**
   46:         * Intializes the CRS matrix structure from the given discetization in @p mes
h.
   47:         *
   48:         * The sparse matrix pattern is generated but the values are 0.
   49:         *
   50:         * @param[in] mesh given discretization
   51:         *
   52:         * @warning A reference to the discretization @p mesh is stored inside this c
lass.
   53:         *          Therefore, changing @p mesh outside requires also
   54:         *          to call method @p Derive_Matrix_Pattern explicitely.
   55:         *
   56:         * @see Derive_Matrix_Pattern
   57:         */
   58:        explicit CRS_Matrix(Mesh const & mesh);
   59:
   60:       /**
   61:         * Destructor.
   62:         */
   63:        ~CRS_Matrix()
   64:        {}
```

```
   65:
   66:         /**
   67:          * Generates the sparse matrix pattern and overwrites the existing pattern.
   68:          *
   69:          * The sparse matrix pattern is generated but the values are 0.
   70:          */
   71:         void Derive_Matrix_Pattern();
   72:
   73:          /**
   74:          * Calculates the entries of f.e. stiffness matrix and load/rhs vector @p f f
or the Laplace operator in 2D.
   75:          * No memory is allocated.
   76:          *
   77:          * @param[in,out] f (preallocated) rhs/load vector
   78:          */
   79:         void CalculateLaplace(std::vector<double> &f);
   80:
   81:         /**
   82:          * Applies Dirichlet boundary conditions to stiffness matrix and to load vect
or @p f.
   83:          * The <a href="https://www.jstor.org/stable/2005611?seq=1#metadata_info_tab_
contents">penalty method</a>
   84:          * is used for incorporating the given values @p u.
   85:          *
   86:          * @param[in]     u (global) vector with Dirichlet data
   87:          * @param[in,out] f load vector
   88:          */
   89:         void ApplyDirichletBC(std::vector<double> const &u, std::vector<double> &f);
   90:
   91:         /**
   92:          * Extracts the diagonal elemenst of the sparse matrix.
   93:          *
   94:          * @param[in,out]  d  (prellocated) vector of diagonal elements
   95:          */
   96:         void GetDiag(std::vector<double> &d) const;
   97:
   98:         /**
   99:          * Performs the matrix-vector product  w := K*u.
  100:          *
  101:          * @param[in,out] w resulting vector (preallocated)
  102:          * @param[in]     u vector
  103:          */
  104:         void Mult(std::vector<double> &w, std::vector<double> const &u) const;
  105:
  106:          /**
  107:          * Calculates the defect/residuum w := f - K*u.
  108:          *
  109:          * @param[in,out] w resulting vector (preallocated)
  110:          * @param[in]     f load vector
  111:          * @param[in]     u vector
  112:          */
  113:         void Defect(std::vector<double> &w,
  114:                     std::vector<double> const &f, std::vector<double> const &u) const
;
  115:
  116:         /**
  117:                   * Number rows in matrix.
  118:                   * @return number of rows.
  119:                   */
  120:         int Nrows() const
  121:            {return _nrows;}
  122:
  123:         /**
  124:                   * Show the matrix entries.
  125:                   */
  126:         void Debug() const;
  127:
  128:         /**
```

```
  129:                  * Finds in a CRS matrix the access index for an entry at row @p row
 and column @p col.
  130:                  *
  131:                  * @param[in] row       row index
  132:                  * @param[in] col       column index
  133:                  * @return index for element (@p row, @p col). If no appropriate ent
ry exists then -1 will be returned.
  134:                  *
  135:                  * @warning assert() stops the function in case that matrix element
(@p row, @p col) doesn't exist.
  136:                 */
  137:         int fetch(int row, int col) const;
  138:
  139:        /**
  140:         * Adds the element stiffness matrix @p ske and the element load vector @p fe
  141:         * of one triangular element with linear shape functions to the appropriate p
ositions in
  142:         * the stiffness matrix, stored as CSR matrix K(@p sk,@p id, @p ik).
  143:         *
  144:         * @param[in]     ial   node indices of the three element vertices
  145:         * @param[in]     ske   element stiffness matrix
  146:         * @param[in]     fe    element load vector
  147:         * @param[in,out] f         distributed local vector storing the right hand s
ide
  148:         *
  149:         * @warning Algorithm assumes  linear triangular elements (ndof_e==3).
  150:         */
  151:         void AddElem_3(int const ial[3], double const ske[3][3], double const fe[3],
std::vector<double> &f);
  152:
  153:         /**
  154:         * Compare @p this CRS matrix with an external CRS matrix stored in C-Style.
  155:         *
  156:         * The method prints statements on differences found.
  157:         *
  158:         * @param[in]     nnode  row number of external matrix
  159:         * @param[in]     id     start indices of matrix rows of external matrix
  160:         * @param[in]     ik     column indices of external matrix
  161:         * @param[in]     sk     non-zero values of external matrix
  162:         *
  163:         * @return true iff all data are identical.
  164:         */
  165:         bool Compare2Old(int nnode, int const id[], int const ik[], double const sk[]
) const;
  166:
  167:     private:
  168:         Mesh const & _mesh;       //!< reference to discretization
  169:         int _nrows;               //!< number of rows in matrix
  170:         int _nnz;                 //!< number of non-zero entries
  171:         std::vector<int> _id;     //!< start indices of matrix rows
  172:         std::vector<int> _ik;     //!< column indices
  173:         std::vector<double> _sk; //!< non-zero values
  174:
  175: };
  176:
  177:
  178: #endif
```

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

#include <cassert>
#include <cmath>
#include <iostream>
#include <omp.h>
#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 shared(u, nrows)
    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;
    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;
}

```

```
    1: #ifndef JACSOLVE_FILE
    2: #define JACSOLVE_FILE
    3: #include "getmatrix.h"
    4: #include <vector>
    5:
    6:
    7: /**
    8:  * Solves linear system of equations  K @p u = @p f  via the Jacobi iteration.
    9:  * We use a distributed symmetric  CSR matrix @p SK and initial guess of the
   10:  * solution is set to 0.
   11:  * @param[in] SK        CSR matrix
   12:  * @param[in] f         distributed local vector storing the right hand side
   13:  * @param[out] u        accumulated local vector storing the solution.
   14: */
   15: void JacobiSolve(CRS_Matrix const &SK, std::vector<double> const &f, std::vector<dou
ble> &u);
   16:
   17:
   18: #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>
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
   68:
   69:             SK.ApplyDirichletBC(uv, fv);
   70:             //SK.Compare2Old(nnode, id, ik, sk);
   71:             //SK.Debug();
   72:
   73:             auto tstart = system_clock::now();       // start timer
   74:
   75:             JacobiSolve(SK, fv, uv );          // solve the system of equations
   76:
   77:             auto tend = system_clock::now();         // end timer
   78:             auto duration = duration_cast<microseconds>(tend - tstart);
   79:             auto t1 = static_cast<double>(duration.count()) / 1e6 ;   // t1 in secon
ds
   80:             cout << "JacobiSolve: timing in sec. : " << t1 << endl;
   81:
   82:             //CompareVectors(uv, nnode, u, 1e-6);    // Check correctness
   83:
   84:             //mesh.SaveVectorP("t.dat", uv);
   85:             //mesh.Visualize(uv);
   86:
   87:         }
   88: // ##################### STL ##########################################
   89:         {
   90:             //Mesh_2d_3_matlab const mesh("square_tiny.txt");
   91:             Mesh_2d_3_matlab const mesh("square_100.txt");
   92:             //Mesh_2d_3_matlab const mesh("L_shape.txt");
   93:             //mesh.Debug();
   94:
   95:             CRS_Matrix SK(mesh);                    // CRS matrix
   96:             //SK.Debug();
   97:
   98:             vector<double> uv(SK.Nrows(), 0.0);     // temperature
   99:             vector<double> fv(SK.Nrows(), 0.0);     // r.h.s.
  100:
  101:             SK.CalculateLaplace(fv);
  102:             //SK.Debug();
  103:
  104:             //mesh.SetU(uv);          // deprecated
  105:             // Two ways to initialize the vector
  106:             //mesh.SetValues(uv,f_zero);              // user function
  107:             mesh.SetValues(uv, [](double x, double y) -> double {return 0.0 * x *y;}
); // lambda function
  108:
  109:             SK.ApplyDirichletBC(uv, fv);
  110:             //SK.Compare2Old(nnode, id, ik, sk);
  111:             //SK.Debug();
  112:
  113:             auto tstart = system_clock::now();       // start timer
  114:
  115:             JacobiSolve(SK, fv, uv );          // solve the system of equations
  116:
  117:             auto tend = system_clock::now();         // end timer
  118:             auto duration = duration_cast<microseconds>(tend - tstart);
  119:             auto t1 = static_cast<double>(duration.count()) / 1e6 ;   // t1 in secon
ds
  120:             cout << "JacobiSolve: timing in sec. : " << t1 << endl;
  121:
  122:             //mesh.Write_ascii_matlab("uv.txt", uv);
  123:             //mesh.Visualize(uv);
  124:         }
  125:         return 0;
  126:     }
  127: }
  128:
  129:
```

```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 shared(x, y, z, n)
    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 shared(x, y, z, n, alpha)
    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 shared(x, y, n) 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;
}
//***************************************************************************
```

```
   69: bool CompareVectors(std::vector<double> const &x, int const n, double const y[], dou
ble const eps)
   70: {
   71:     bool bn = (static_cast<int>(x.size()) == n);
   72:     if (!bn)
   73:     {
   74:         cout << "#########   Error: " << "number of elements" << endl;
   75:     }
   76:     //bool bv = equal(x.cbegin(),x.cend(),y);
   77:     bool bv = equal(x.cbegin(), x.cend(), y,
   78:                     [eps](double a, double b) -> bool
   79:     { return std::abs(a - b) < eps * (1.0 + 0.5 * (std::abs(a) + std::abs(a))); }
   80:                     );
   81:     if (!bv)
   82:     {
   83:         assert(static_cast<int>(x.size()) == n);
   84:         cout << "#########   Error: " << "values" << endl;
   85:     }
   86:     return bn && bv;
   87: }
```

```
    1: #ifndef VDOP_FILE
    2: #define VDOP_FILE
    3: #include <vector>
    4:
    5: /** @brief  Element-wise vector divison x_k = y_k/z_k.
    6:  *
    7:  * @param[out] x  target vector
    8:  * @param[in]  y  source vector
    9:  * @param[in]  z  source vector
   10:  *
   11:  */
   12: void vddiv(std::vector<double> & x, std::vector<double> const& y,
   13:                                     std::vector<double> const& z);
   14:
   15: /** @brief  Element-wise daxpy operation x(k) = y(k) + alpha*z(k).
   16:  *
   17:  * @param[out] x  target vector
   18:  * @param[in]  y  source vector
   19:  * @param[in]  alpha  scalar
   20:  * @param[in]  z  source vector
   21:  *
   22:  */
   23: void vdaxpy(std::vector<double> & x, std::vector<double> const& y,
   24:                         double alpha, std::vector<double> const& z );
   25:
   26:
   27: /** @brief  Calculates the Euclidian inner product of two vectors.
   28:  *
   29:  * @param[in]  x vector
   30:  * @param[in]  y vector
   31:  * @return Euclidian inner product @f$\langle x,y \rangle@f$
   32:  *
   33:  */
   34: double dscapr(std::vector<double> const& x, std::vector<double> const& y);
   35:
   36:
   37: /**
   38:  * Print entries of a vector.
   39:  * @param[in] v     vector values
   40: */
   41: void DebugVector(std::vector<double> const &v);
   42:
   43: /** @brief  Compares an STL vector with POD vector.
   44:  *
   45:  * The accuracy criteria @f$ |x_k-y_k| < \varepsilon \left({1+0.5(|x_k|+|y_k|)}\righ
t) @f$
   46:  * follows the book by
   47:  * <a href="https://www.springer.com/la/book/9783319446592">Stoyan/Baran</a>, p.8.
   48:  *
   49:  * @param[in]  x    STL vector
   50:  * @param[in]  n    length of POD vector
   51:  * @param[in]  y    POD vector
   52:  * @param[in]  eps  relative accuracy criteria (default := 0.0).
   53:  * @return true iff pairwise vector elements are relatively close to each other.
   54:  *
   55:  */
   56: bool CompareVectors(std::vector<double> const& x, int n, double const y[], double co
nst eps=0.0);
   57:
   58: #endif
```