

```

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, double)> &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 DELIMITER(" ");    // define the same delimiter 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() << DELIMITER << Ndims() << DELIMITER << Nelems() << DELIMITER <
< NverticesElements() << endl;
72:
73:     // Write cordينات: 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] << DELIMITER;
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 << DELIMITER;    // 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: ,
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:     for (int j = 0; j <= _ny; ++j)
172:     {
173:         int k = j * dx;
174:         for (int i = 0; i <= _nx; ++i, ++k)
175:         {
176:             u[k] = 0.0;
177:         }
178:     }
179: }
180:
181:
182: void Mesh_2d_3_square::SetF(std::vector<double> &f) const
183: {
184:     int dx = _nx + 1;
185:     for (int j = 0; j <= _ny; ++j)
186:     {
187:         int k = j * dx;
188:         for (int i = 0; i <= _nx; ++i, ++k)
189:         {
190:             f[k] = 1.0;
191:         }
192:     }

```

```

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

```

```

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

```

```

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

```

```

387:
388:
389: // #####
390:
391: Mesh_2d_3_matlab::Mesh_2d_3_matlab(string const &fname)
392:     : Mesh(2, 3, 3), // two dimensions, 3 vertices, 3 dofs
393:     bedges(0)
394: {
395:     ifstream ifs(fname);
396:     if (!(ifs.is_open() && ifs.good()))
397:     {
398:         cerr << "Mesh_2d_3_matlab: Error cannot open file " << fname << endl;
399:         assert(ifs.is_open());
400:     }
401:
402:     int const OFFSET(1); // Matlab to C indexing
403:     cout << "ASCII file " << fname << " opened" << endl;
404:
405:     // Read some mesh constants
406:     int nnode, ndim, nele, nvert_e;
407:     ifs >> nnode >> ndim >> nele >> nvert_e;
408:     cout << nnode << " " << ndim << " " << nele << " " << nvert_e << endl;
409:     assert(ndim == 2 && nvert_e == 3);
410:
411:     // Allocate memory
412:     Resize_Coords(nnode, ndim); // coordinates in 2D [nnode][ndim]
413:     Resize_Connectivity(nele, nvert_e); // connectivity matrix [nele][nvert
]
414:
415:     // Read coordinates
416:     auto &xc = GetCoords();
417:     for (int k = 0; k < nnode * ndim; ++k)
418:     {
419:         ifs >> xc[k];
420:     }
421:
422:     // Read connectivity
423:     auto &ia = GetConnectivity();
424:     for (int k = 0; k < nele * nvert_e; ++k)
425:     {
426:         ifs >> ia[k];
427:         ia[k] -= OFFSET; // Matlab to C indexing
428:     }
429:
430:     // additional read of boundary information (only start/end point)
431:     int nbedges;
432:     ifs >> nbedges;
433:
434:     bedges.resize(nbedges * 2);
435:     for (int k = 0; k < nbedges * 2; ++k)
436:     {
437:         ifs >> bedges[k];
438:         bedges[k] -= OFFSET; // Matlab to C indexing
439:     }
440:
441:     return;
442: }
443:
444: // binary
445: //{
446: //ifstream ifs(fname, ios_base::in | ios_base::binary);
447: //if(!(ifs.is_open() && ifs.good()))
448: //{
449: //cerr << "ReadBinMatrix: Error cannot open file " << file << endl;
450: //assert(ifs.is_open());
451: //}
452: //cout << "ReadBinMatrix: file opened" << file << endl;
453:

```

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


520:

521:

522:

```

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: /**
9:  * Basis class for finite element meshes.
10: */
11: class Mesh
12: {
13: public:
14:     /**
15:      * Constructor initializing the members with default values.
16:      *
17:      * @param[in] ndim   space dimensions (dimension for coordinates)
18:      * @param[in] nvert_e number of vertices per element (dimension for connectivity)
19:      * @param[in] ndof_e degrees of freedom per element (= @p nvert_e for linear elements)
20:      */
21:     explicit Mesh(int ndim, int nvert_e = 0, int ndof_e = 0);
22:
23:     /**
24:      * Destructor.
25:      *
26:      * See clang warning on
27:      * <a href="https://stackoverflow.com/questions/28786473/clang-no-out-of-line-virtual-method-definitions-pure-abstract-c-class/40550578">weak-vtables</a>.
28:      */
29:     virtual ~Mesh();
30:
31:     /**
32:      * Number of finite elements in (sub)domain.
33:      * @return number of elements.
34:      */
35:     int Nelems() const
36:     {
37:         return _nelem;
38:     }
39:
40:     /**
41:      * Global number of vertices for each finite element.
42:      * @return number of vertices per element.
43:      */
44:     int NverticesElements() const
45:     {
46:         return _nvert_e;
47:     }
48:
49:     /**
50:      * Global number of degrees of freedom (dof) for each finite element.
51:      * @return degrees of freedom per element.
52:      */
53:     int NdofsElement() const
54:     {
55:         return _ndof_e;
56:     }
57:
58:     /**
59:      * Number of vertices in mesh.
60:      * @return number of vertices.
61:      */
62:     int Nnodes() const
63:     {
64:         return _nnode;
65:     }

```

```

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

```

```

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

```

```

197:     {
198:         _nnode = nnode;
199:     }
200:
201:     void SetNdim(int ndim)
202:     {
203:         _ndim = ndim;
204:     }
205:
206: private:
207:     int _nelem;           //!< number elements
208:     int _nvert_e;         //!< number of vertices per element
209:     int _ndof_e;          //!< degrees of freedom (d.o.f.) per element
210:     int _nnode;           //!< number nodes/vertices
211:     int _ndim;            //!< space dimension of the problem (1, 2, or 3)
212:     std::vector<int> _ia;  //!< element connectivity
213:     std::vector<double> _xc; //!< coordinates
214: };
215:
216: /**
217:  * 2D finite element mesh of the square consisting of linear triangular elements.
218:  */
219: class Mesh_2d_3_square: public Mesh
220: {
221: public:
222:     /**
223:      * Generates the f.e. mesh for the unit square.
224:      *
225:      * @param[in] nx    number of discretization intervals in x-direction
226:      * @param[in] ny    number of discretization intervals in y-direction
227:      * @param[in] myid  my MPI-rank / subdomain
228:      * @param[in] procx number of ranks/subdomains in x-direction
229:      * @param[in] procy number of processes in y-direction
230:      */
231:     Mesh_2d_3_square(int nx, int ny, int myid = 0, int procx = 1, int procy = 1);
232:
233:     /**
234:      * Destructor
235:      */
236:     ~Mesh_2d_3_square() override
237:     {}
238:
239:     /**
240:      * Set solution vector based on a tensor product grid in the rectangle.
241:      * @param[in] u solution vector
242:      */
243:     void SetU(std::vector<double> &u) const;
244:
245:     /**
246:      * Set right hand side (rhs) vector on a tensor product grid in the rectangle.
247:      * @param[in] f rhs vector
248:      */
249:     void SetF(std::vector<double> &f) const;
250:
251:     /**
252:      * Determines the indices of those vertices with Dirichlet boundary conditions
253:      * @return index vector.
254:      */
255:     std::vector<int> Index_DirichletNodes() const override;
256:
257:     /**
258:      * Stores the values of vector @p u of (sub)domain into a file @p name for further
259:      * processing in gnuplot.
260:      * The file stores rowwise the x- and y- coordinates together with the value from
261:      * @p u.
262:      * The domain [@p xl, @p xr] x [@p yb, @p yt] is discretized into @p nx x @p ny
263:      * intervals.
264:      */

```

```

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

```

```

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

```

```

1: #include "getmatrix.h"
2: #include "userset.h"
3:
4: #include <algorithm>
5: #include <cassert>
6: #include <cmath>
7: #include <iomanip>
8: #include <iostream>
9: #include <list>
10: #include <vector>
11: #include <omp.h>
12: using namespace std;
13:
14:
15: // general routine for lin. triangular elements
16:
17: void CalcElem(int const ial[3], double const xc[], double ske[3][3], double fe[3])
18: //void CalcElem(const int* __restrict__ ial, const double* __restrict__ xc, double*
__restrict__ ske[3], double* __restrict__ fe)
19: {
20:     const int i1 = 2 * ial[0], i2 = 2 * ial[1], i3 = 2 * ial[2];
21:     const double x13 = xc[i3 + 0] - xc[i1 + 0], y13 = xc[i3 + 1] - xc[i1 + 1],
22:                 x21 = xc[i1 + 0] - xc[i2 + 0], y21 = xc[i1 + 1] - xc[i2 + 1],
23:                 x32 = xc[i2 + 0] - xc[i3 + 0], y32 = xc[i2 + 1] - xc[i3 + 1];
24:     const double jac = fabs(x21 * y13 - x13 * y21);
25:
26:     ske[0][0] = 0.5 / jac * (y32 * y32 + x32 * x32);
27:     ske[0][1] = 0.5 / jac * (y13 * y32 + x13 * x32);
28:     ske[0][2] = 0.5 / jac * (y21 * y32 + x21 * x32);
29:     ske[1][0] = ske[0][1];
30:     ske[1][1] = 0.5 / jac * (y13 * y13 + x13 * x13);
31:     ske[1][2] = 0.5 / jac * (y21 * y13 + x21 * x13);
32:     ske[2][0] = ske[0][2];
33:     ske[2][1] = ske[1][2];
34:     ske[2][2] = 0.5 / jac * (y21 * y21 + x21 * x21);
35:
36:     const double xm = (xc[i1 + 0] + xc[i2 + 0] + xc[i3 + 0]) / 3.0,
37:                 ym = (xc[i1 + 1] + xc[i2 + 1] + xc[i3 + 1]) / 3.0;
38:     //fe[0] = fe[1] = fe[2] = 0.5 * jac * FunctF(xm, ym) / 3.0;
39:     fe[0] = fe[1] = fe[2] = 0.5 * jac * fNice(xm, ym) / 3.0;
40: }
41:
42:
43: // general routine for lin. triangular elements,
44: // non-symm. matrix
45: // node numbering in element: a s c e n d i n g indices !!
46: [[deprecated("Use CRS_Matrix::AddElem_3(...) instead.")]]
47: void AddElem(int const ial[3], double const ske[3][3], double const fe[3],
48:             int const id[], int const ik[], double sk[], double f[])
49: {
50:     for (int i = 0; i < 3; ++i)
51:     {
52:         const int ii = ial[i], // row ii (global index)
53:                 id1 = id[ii], // start and
54:                 id2 = id[ii + 1]; // end of row ii in matrix
55:         int ip = id1;
56:         for (int j = 0; j < 3; ++j) // no symmetry assumed
57:         {
58:             const int jj = ial[j];
59:             bool not_found = true;
60:             do // find entry jj (global index) in row ii
61:             {
62:                 not_found = (ik[ip] != jj);
63:                 ++ip;
64:             }
65:             while (not_found && ip < id2);
66:
67: #ifndef NDEBUG // compiler option -DNDEBUG switches off the check

```



```

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

```

```

132: // copy CSR data
133: _id[0] = 0; // begin of first row
134: for (size_t i = 0; i < cc.size(); ++i)
135: {
136:     //cout << i << " " << nid.at(i) << endl;;
137:     const list<int> &ci = cc.at(i);
138:     const auto nci = static_cast<int>(ci.size());
139:     _id[i + 1] = _id[i] + nci; // begin of next line
140:     copy(ci.begin(), ci.end(), _ik.begin() + _id[i] );
141: }
142:
143: assert(_nnz == _id[_nrows]);
144: _sk.resize(_nnz); // Allocate memory for CSR column index v
ector
145: return;
146: }
147:
148:
149: void CRS_Matrix::Debug() const
150: {
151: // ID points to first entry of row
152: // no symmetry assumed
153: cout << "\nMatrix (nnz = " << _id[_nrows] << ")\n";
154:
155: for (int row = 0; row < _nrows; ++row)
156: {
157:     cout << "Row " << row << " : ";
158:     int const id1 = _id[row];
159:     int const id2 = _id[row + 1];
160:     for (int j = id1; j < id2; ++j)
161:     {
162:         cout.setf(ios::right, ios::adjustfield);
163:         cout << "[" << setw(2) << _ik[j] << "]" " << setw(4) << _sk[j] << " ";
164:     }
165:     cout << endl;
166: }
167: return;
168: }
169:
170: void CRS_Matrix::CalculateLaplace(vector<double> &f)
171: {
172:     double t_start = omp_get_wtime();
173:
174:     double t0, t1, t2, t3, t4;
175:     assert(_mesh.NdofsElement() == 3); // only for triangular, linear
elements
176: //cout << _nnz << " vs. " << _id[_nrows] << " " << _nrows<< endl;
177: assert(_nnz == _id[_nrows]);
178: auto const nelem = _mesh.Nelems();
179: auto const &ia = _mesh.GetConnectivity();
180: auto const &xc = _mesh.GetCoords();
181: vector<int> elem2csr(nelem * 9);
182: vector<double> elemval(nelem * 9);
183: vector<double> elemrhs(nelem * 3);
184: #pragma omp parallel
185: {
186:     #pragma omp for
187:     for (int k = 0; k < _nrows; ++k)
188:     {
189:         _sk[k] = 0.0;
190:     }
191:
192:     #pragma omp barrier
193:     #pragma omp single
194:     t0 = omp_get_wtime();
195:
196:
197:

```

```
198:     #pragma omp for
199:     for (int k = 0; k < _nrows; ++k)
200:     {
201:         f[k] = 0.0;
202:     }
203:
204:
205:
206:     #pragma omp barrier
207:     #pragma omp single
208:     t1 = omp_get_wtime();
209:
210:
211:     #pragma omp for
212:     for (int e = 0; e < nelem; ++e)
213:     {
214:         const int* nodes = ia.data() + 3*e;
215:         for (int i = 0; i < 3; ++i)
216:         {
217:             for (int j = 0; j < 3; ++j)
218:             {
219:                 int row = nodes[i];
220:                 int col = nodes[j];
221:                 int csr_index = fetch(row, col);
222:                 elem2csr[e*9 + i*3 + j] = csr_index;
223:             }
224:         }
225:
226:     }
227:     #pragma omp barrier
228:     #pragma omp single
229:     t2 = omp_get_wtime();
230:
231:
232:     double ske[3][3], fe[3];
233:     // Loop over all elements
234:
235:
236:     #pragma omp for
237:     for (int i = 0; i < nelem; ++i)
238:     {
239:         CalcElem(ia.data() + 3 * i, xc.data(), ske, fe);
240:         //store elements
241:         for (int j = 0; j < 3; j++)
242:         {
243:
244:             for (int k = 0; k < 3; k++)
245:             {
246:                 elemval[i*9 + j*3 + k] = ske[j][k];
247:             }
248:             elemrhs[i*3 + j] = fe[j];
249:         }
250:
251:     }
252:     #pragma omp barrier
253:     #pragma omp single
254:     t3 = omp_get_wtime();
255:
256:
257:     vector<double> localSK(_nnz, 0.0);
258:     //add in local vectors/matrices
259:     #pragma omp for
260:     for (int e = 0; e < nelem; ++e)
261:     {
262:         for (int k = 0; k < 9; k++)
263:         {
264:             int csr_i = elem2csr[e*9 + k];
265:             localSK[csr_i] += elemval[e*9 + k];
```

```

266:         }
267:     }
268:     vector<double> localF(f.size(), 0.0);
269:     #pragma omp for
270:     for (int e = 0; e < nelelem; ++e)
271:     {
272:         const int* nodes = ia.data() + 3*e;
273:         for (int i = 0; i < 3; i++)
274:         {
275:             localF[nodes[i]] += elemrhs[e*3 + i];
276:         }
277:     }
278: }
279:
280: #pragma omp barrier
281: #pragma omp single
282: t4 = omp_get_wtime();
283:
284: //now add elements again into matrix
285: #pragma omp critical
286: {
287:     for (int i = 0; i < _nnz; ++i)
288:     {
289:         _sk[i] += localSK[i];
290:     }
291:
292:     for (int i = 0; i < (int)f.size(); ++i)
293:     {
294:         f[i] += localF[i];
295:     }
296: }
297:
298: //Debug();
299: }
300: double t5 = omp_get_wtime();
301:
302: cout << "Zero matrix: " << (t0 - t_start) << " sec\n";
303: cout << "Zero RHS: " << (t1 - t0) << " sec\n";
304: cout << "elem2csr: " << (t2 - t1) << " sec\n";
305: cout << "CalcElem + store: " << (t3 - t2) << " sec\n";
306: cout << "add in local vectors/matrices: " << (t4 - t3) << " sec\n";
307: cout << "add elements again into matrix: " << (t5 - t4) << " sec\n";
308: cout << "Total assembly: " << (t5 - t_start) << " sec\n";
309:
310: }
311:
312: void CRS_Matrix::ApplyDirichletBC(std::vector<double> const &u, std::vector<double>
&f)
313: {
314:     double const PENALTY = 1e6;
315:     auto const idx = _mesh.Index_DirichletNodes();
316:     int const nidx = static_cast<int>(idx.size());
317:
318:     for (int row = 0; row < nidx; ++row)
319:     {
320:         int const k = idx[row];
321:         int const id1 = fetch(k, k); // Find diagonal entry of row
322:         assert(id1 >= 0);
323:         _sk[id1] += PENALTY; // matrix weighted scaling feasible
324:         f[k] += PENALTY * u[k];
325:     }
326:
327:     return;
328: }
329:
330: void CRS_Matrix::GetDiag(vector<double> &d) const
331: {
332:     assert( _nrows == static_cast<int>(d.size()) );

```

```

333:
334:     for (int row = 0; row < _nrows; ++row)
335:     {
336:         const int ia = fetch(row, row); // Find diagonal entry of row
337:         assert(ia >= 0);
338:         d[row] = _sk[ia];
339:     }
340:     return;
341: }
342:
343: bool CRS_Matrix::Compare2Old(int nnode, int const id[], int const ik[], double const
sk[]) const
344: {
345:     bool bn = (nnode == _nrows); // number of rows
346:     if (!bn)
347:     {
348:         cout << "##### Error: " << "number of rows" << endl;
349:     }
350:
351:     bool bz = (id[nnode] == _nnz); // number of non zero elements
352:     if (!bz)
353:     {
354:         cout << "##### Error: " << "number of non zero elements" << endl;
355:     }
356:
357:     bool bd = equal(id, id + nnode + 1, _id.cbegin()); // row starts
358:     if (!bd)
359:     {
360:         cout << "##### Error: " << "row starts" << endl;
361:     }
362:
363:     bool bk = equal(ik, ik + id[nnode], _ik.cbegin()); // column indices
364:     if (!bk)
365:     {
366:         cout << "##### Error: " << "column indices" << endl;
367:     }
368:
369:     bool bv = equal(sk, sk + id[nnode], _sk.cbegin()); // values
370:     if (!bv)
371:     {
372:         cout << "##### Error: " << "values" << endl;
373:     }
374:
375:     return bn && bz && bd && bk && bv;
376: }
377:
378:
379: void CRS_Matrix::Mult(vector<double> &w, vector<double> const &u) const
380: {
381:     assert( _nrows == static_cast<int>(w.size()) );
382:     assert( w.size() == u.size() );
383:
384:     for (int row = 0; row < _nrows; ++row)
385:     {
386:         double wi = 0.0;
387:         for (int ij = _id[row]; ij < _id[row + 1]; ++ij)
388:         {
389:             wi += _sk[ij] * u[ _ik[ij] ];
390:         }
391:         w[row] = wi;
392:     }
393:     return;
394: }
395:
396: void CRS_Matrix::Defect(vector<double> &w,
vector<double> const &f, vector<double> const &u) const
397: {
398:     assert( _nrows == static_cast<int>(w.size()) );
399:

```

```

400:     assert( w.size() == u.size() && u.size() == f.size() );
401:
402:     #pragma omp parallel for
403:     for (int row = 0; row < _nrows; ++row)
404:     {
405:         double wi = f[row];
406:         for (int ij = _id[row]; ij < _id[row + 1]; ++ij)
407:         {
408:             wi -= _sk[ij] * u[ _ik[ij] ];
409:         }
410:         w[row] = wi;
411:     }
412:     return;
413: }
414:
415: int CRS_Matrix::fetch(int const row, int const col) const
416: {
417:     int const id2 = _id[row + 1];    // end and
418:     int ip = _id[row];               // start of recent row (global index)
419:
420:     while (ip < id2 && _ik[ip] != col) // find index col (global index)
421:     {
422:         ++ip;
423:     }
424:     if (ip >= id2)
425:     {
426:         ip = -1;
427: #ifndef NDEBUG // compiler option -DNDEBUG switches off the check
428:         cout << "No column " << col << " in row " << row << endl;
429:         assert(ip >= id2);
430: #endif
431:     }
432:     return ip;
433: }
434:
435:
436: // general routine for lin. triangular elements,
437: // non-symm. matrix
438: // node numbering in element: a s c e n d i n g indices !!
439: void CRS_Matrix::AddElem_3(int const ial[3], double const ske[3][3], double const fe
[3], vector<double> &f)
440: {
441:     for (int i = 0; i < 3; ++i)
442:     {
443:         const int ii = ial[i]; // row ii (global index)
444:         for (int j = 0; j < 3; ++j) // no symmetry assumed
445:         {
446:             const int jj = ial[j]; // column jj (global index)
447:             int ip = fetch(ii, jj); // find column entry jj in row ii
448: #ifndef NDEBUG // compiler option -DNDEBUG switches off the check
449:             if (ip < 0) // no entry found !!
450:             {
451:                 cout << "Error in AddElem: (" << ii << "," << jj << ") [" <<
452:                     << ial[0] << "," << ial[1] << "," << ial[2] << "]\n";
453:                 assert(ip >= 0);
454:             }
455: #endif
456:             #pragma omp atomic
457:             _sk[ip] += ske[i][j];
458:         }
459:         #pragma omp atomic
460:         f[ii] += fe[i];
461:     }
462: }
463:

```

```

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 coordinates of
12:  * node k
13:  * @param[out] ske element stiffness matrix
14:  * @param[out] fe element load vector
15:  */
16: void CalcElem(int const ial[3], double const xc[], double ske[3][3], double fe[3]);
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
20:  * in
21:  * the symmetric stiffness matrix, stored as CSR matrix K(@p sk,@p id, @p ik)
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:          * Initializes the CRS matrix structure from the given discretization in @p mesh.
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 class.
53:          * Therefore, changing @p mesh outside requires also
54:          * to call method @p Derive_Matrix_Pattern explicitly.
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 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

```

```

1: #include "vdop.h"
2: #include "getmatrix.h"
3: #include "jacsolve.h"
4:
5: #include <cassert>
6: #include <cmath>
7: #include <iostream>
8: #include <vector>
9: using namespace std;
10:
11: // #####
12: //     const int neigh[], const int color,
13: //     const MPI::Intracomm& icomm,
14: void JacobiSolve(CRS_Matrix const &SK, vector<double> const &f, vector<double> &u)
15: {
16:     const double omega = 1.0;
17:     const int maxiter = 1000;
18:     const double tol = 1e-5, // tolerance
19:                 tol2 = tol * tol; // tolerance^2
20:
21:     int nrows = SK.Nrows(); // number of rows == number of columns
22:     assert( nrows == static_cast<int>(f.size()) && f.size() == u.size() );
23:
24:     cout << endl << " Start Jacobi solver for " << nrows << " d.o.f.s" << endl;
25:     // Choose initial guess
26:     #pragma omp parallel for
27:     for (int k = 0; k < nrows; ++k)
28:     {
29:         u[k] = 0.0; // u := 0
30:     }
31:
32:     vector<double> dd(nrows); // matrix diagonal
33:     vector<double> r(nrows); // residual
34:     vector<double> w(nrows); // correction
35:
36:     SK.GetDiag(dd); // dd := diag(K)
37:     //DebugVector(dd); {int ijk; cin >> ijk;}
38:
39:     // Initial sweep
40:     SK.Defect(r, f, u); // r := f - K*u
41:
42:     vddiv(w, r, dd); // w := D^{-1}*r
43:     double sigma0 = dscapr(w, r); // s0 := <w,r>
44:
45:     // Iteration sweeps
46:     int iter = 0;
47:     double sigma = sigma0;
48:     // i dont parallelize this cause its iteration dependent
49:     while ( sigma > tol2 * sigma0 && maxiter > iter)
50:     {
51:         ++iter;
52:         vdaxpy(u, u, omega, w ); // u := u + om*w
53:         SK.Defect(r, f, u); // r := f - K*u
54:         vddiv(w, r, dd); // w := D^{-1}*r
55:         sigma = dscapr(w, r); // s0 := <w,r>
56:         // cout << "Iteration " << iter << " : " << sqrt(sigma/sigma0) << endl;
57:     }
58:     cout << "aver. Jacobi rate : " << exp(log(sqrt(sigma / sigma0)) / iter) << " (
" << iter << " iter)" << endl;
59:     cout << "final error: " << sqrt(sigma / sigma0) << " (rel) " << sqrt(sigma) <<
" (abs)\n";
60:
61:     return;
62: }
63:

```

```
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<double> &u);
16:
17:
18: #endif
```

```

1: //          MPI code in C++.
2: //          See [Gropp/Lusk/Skjellum, "Using MPI", p.33/41 etc.]
3: //          and /opt/mpich/include/mpi2c++/comm.h for details
4:
5: #include "geom.h"
6: #include "getmatrix.h"
7: #include "jacsolve.h"
8: #include "userset.h"
9: #include "vdop.h"
10:
11: #include <chrono>          // timing
12: #include <cmath>
13: #include <iostream>
14: #include <omp.h>
15: using namespace std;
16: using namespace std::chrono; // timing
17:
18:
19: int main(int, char ** )
20: {
21:     const int numprocs = 1;
22:     const int myrank   = 0;
23:     if (myrank == 0)
24:     {
25:         cout << "\n There are " << numprocs << " processes running.\n \n";
26:     }
27:
28:     const auto procx = static_cast<int>(sqrt(numprocs + 0.0));
29:     const int  procy = procx;
30:
31:     if (procy * procx != numprocs)
32:     {
33:         cout << "\n Wrong number of processors !\n \n";
34:     }
35:     else
36:     {
37: // #####
38: //      Here starts the real code
39: // #####
40: //bool ScaleUp = !true;
41:     int nx, ny, NXglob, NYglob; /* number of local intervals on (xl,xr)=:nx, (yb
,yt)=:ny */
42:     //nx = 1024;
43:     //ny = 1024;
44:     nx = 100;
45:     ny = 100;
46:     NXglob = nx * procx;
47:     NYglob = ny * procy;
48:     cout << "Intervalls: " << NXglob << " x " << NYglob << endl;
49:
50: // ##### STL #####
51:     {
52:         Mesh_2d_3_square const mesh(nx, ny);
53:         //mesh.Debug();
54:
55:         CRS_Matrix SK(mesh);          // CRS matrix
56:         //SK.Debug();
57:         vector<double> uv(SK.Nrows(), 0.0); // temperature
58:         vector<double> fv(SK.Nrows(), 0.0); // r.h.s.
59:
60:         SK.CalculateLaplace(fv);
61:         //SK.Debug();
62:
63:         //mesh.SetU(uv);          // deprecated
64:         //mesh.SetF(fv);          // deprecated
65:         // Two ways to initialize the vector
66:         //mesh.SetValues(uv, f_zero); // functional
67:         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: }
126:
127:
128:     return 0;
129: }
130: }
131:

```

132:

```
1: #include "useriset.h"
2: #include <cmath>
3:
4:
5: double FunctF(double const x, double const y)
6: {
7: // return std::sin(3.14159*1*x)*std::sin(3.14159*1*y);
8: // return 16.0*1024. ;
9: // return (double)1.0 ;
10:     return x * x * std::sin(2.5 * 3.14159 * y);
11: }
12:
13: double FunctU(const double /* x */, double const /* y */)
14: {
15:     return 1.0 ;
16: }
```

```
1: #ifndef USERSET_FILE
2: #define USERSET_FILE
3: #include <cmath>
4: /**
5:  * User function: f(@p x,@p y)
6:  * @param[in] x      x-coordinate of discretization point
7:  * @param[in] y      y-coordinate of discretization point
8:  * @return  value for right hand side f(@p x,@p y)
9:  */
10: double FunctF(double const x, double const y);
11:
12: /**
13:  * User function: u(@p x,@p y)
14:  * @param[in] x      x-coordinate of discretization point
15:  * @param[in] y      y-coordinate of discretization point
16:  * @return  value for solution vector u(@p x,@p y)
17:  */
18: double FunctU(double const x, double const y);
19:
20:
21: /**
22:  * User function: f(@p x,@p y) = @f$ x^2 \sin(2.5\pi y)@f$.
23:  * @param[in] x      x-coordinate of discretization point
24:  * @param[in] y      y-coordinate of discretization point
25:  * @return  value f(@p x,@p y)
26:  */
27: inline double fNice(double const x, double const y)
28: {
29:     return x * x * std::sin(2.5 * 3.14159 * y);
30: }
31:
32: /**
33:  * User function: f(@p x,@p y) = 0$.
34:  * @param[in] x      x-coordinate of discretization point
35:  * @param[in] y      y-coordinate of discretization point
36:  * @return  value 0
37:  */
38: inline double f_zero(double const x, double const y)
39: //double f_zero(double const /*x*/, double const /*y*/)
40: {
41:     return 0.0 + 0.0*(x+y);
42: }
43:
44: #endif
```



```

1: #include "vdop.h"
2: #include <cassert> // assert()
3: #include <cmath>
4: #include <iostream>
5: #include <vector>
6: using namespace std;
7:
8:
9: void vddiv(vector<double> &x, vector<double> const &y,
10:           vector<double> const &z)
11: {
12:     assert( x.size() == y.size() && y.size() == z.size() );
13:     size_t n = x.size();
14:     #pragma omp parallel for
15:     for (size_t k = 0; k < n; ++k)
16:     {
17:         x[k] = y[k] / z[k];
18:     }
19:     return;
20: }
21:
22: //*****
23:
24: void vdaxpy(std::vector<double> &x, std::vector<double> const &y,
25:            double alpha, std::vector<double> const &z )
26: {
27:     assert( x.size() == y.size() && y.size() == z.size() );
28:     size_t n = x.size();
29:     #pragma omp parallel for
30:     for (size_t k = 0; k < n; ++k)
31:     {
32:         x[k] = y[k] + alpha * z[k];
33:     }
34:     return;
35: }
36: //*****
37:
38: double dscapr(std::vector<double> const &x, std::vector<double> const &y)
39: {
40:     assert( x.size() == y.size() );
41:     size_t n = x.size();
42:
43:     double s = 0.0;
44:     #pragma omp parallel for reduction(+:s)
45:     for (size_t k = 0; k < n; ++k)
46:     {
47:         s += x[k] * y[k];
48:     }
49:
50:     return s;
51: }
52:
53: //*****
54: void DebugVector(vector<double> const &v)
55: {
56:     cout << "\nVector (nnode = " << v.size() << ")\n";
57:     for (size_t j = 0; j < v.size(); ++j)
58:     {
59:         cout.setf(ios::right, ios::adjustfield);
60:         cout << v[j] << " ";
61:     }
62:     cout << endl;
63:
64:     return;
65: }
66: //*****
67: bool CompareVectors(std::vector<double> const &x, int const n, double const y[], double const eps)

```

```
68: {
69:     bool bn = (static_cast<int>(x.size()) == n);
70:     if (!bn)
71:     {
72:         cout << "##### Error: " << "number of elements" << endl;
73:     }
74:     //bool bv = equal(x.cbegin(), x.cend(), y);
75:     bool bv = equal(x.cbegin(), x.cend(), y,
76:         [eps](double a, double b) -> bool
77:         { return std::abs(a - b) < eps * (1.0 + 0.5 * (std::abs(a) + std::abs(b))); }
78:         );
79:     if (!bv)
80:     {
81:         assert(static_cast<int>(x.size()) == n);
82:         cout << "##### Error: " << "values" << endl;
83:     }
84:     return bn && bv;
85: }
```

```

1: #ifndef VDOP_FILE
2: #define VDOP_FILE
3: #include <vector>
4:
5: /** @brief Element-wise vector division  $x_k = y_k/z_k$ .
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  $\langle x, y \rangle$ 
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  $|x_k - y_k| < \epsilon \left( (1 + 0.5(|x_k| + |y_k|)) \right)$ 
46:  * follows the book by
47:  * Stoyan/Baran, 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

```