

```
1: #undef CUTHILL_MCKEE
2: // see: http://llvm.org/docs/CodingStandards.html#include-style
3: #include "vdop.h"
4: #ifdef CUTHILL_MCKEE
5: #include "cuthill_mckee_ordering.h"
6: #endif
7: #include "geom.h"
8:
9: #include <algorithm>
10: #include <array>
11: #include <cassert>
12: #include <cmath>
13: #include <ctime> // contains clock()
14: #include <fstream>
15: #include <iostream>
16: #include <list>
17: #include <string>
18: #include <vector>
19:
20: using namespace std;
21:
22: Mesh::Mesh(int ndim, int nvert_e, int ndof_e, int nedge_e)
23:     : _nelem(0), _nvert_e(nvert_e), _ndof_e(ndof_e), _nnode(0), _ndim(ndim), _ia(0),
  _xc(0),
24:     _bedges(0), _sdedges(0),
25:     _nedge(0), _nedge_e(nedge_e), _edges(0), _ea(), _ebedges(),
26:     _dummy(0)
27: {
28: }
29:
30: Mesh::~Mesh()
31: {}
32:
33: void Mesh::SetValues(std::vector<double> &v, const function<double(double, double)>
&func) const
34: {
35:     int const nnode = Nnodes(); // number of vertices in mesh
36:     assert( nnode == static_cast<int>(v.size()) );
37:     for (int k = 0; k < nnode; ++k)
38:     {
39:         v[k] = func( _xc[2 * k], _xc[2 * k + 1] );
40:     }
41: }
42:
43: void Mesh::SetBoundaryValues(vector<double> &v, const function<double(double, double
)> &func) const
44: {
45:     auto const idx = Index_BoundaryNodes();
46:     for (size_t ik = 0; ik < idx.size(); ++ik)
47:     {
48:         const int k = idx[ik];
49:         v[k] = func( _xc[2 * k], _xc[2 * k + 1] );
50:     }
51: }
52:
53: void Mesh::SetDirichletValues(vector<double> &v, const function<double(double, double
)> &func) const
54: {
55:     auto const idx = Index_DirichletNodes();
56:     for (size_t ik = 0; ik < idx.size(); ++ik)
57:     {
58:         const int k = idx[ik];
59:         v[k] = func( _xc[2 * k], _xc[2 * k + 1] );
60:     }
61: }
62:
63:
64:
```

```

65: void Mesh::Debug() const
66: {
67:     cout << "\n ##### Debug M E S H #####\n";
68:     cout << "\n ..... Coordinates ..... \n";
69:     for (int k = 0; k < _nnode; ++k)
70:     {
71:         cout << k << " : " ;
72:         for (int i = 0; i < _ndof_e; ++i )
73:         {
74:             cout << _xc[2*k+i] << " ";
75:         }
76:         cout << endl;
77:     }
78:     cout << "\n ..... Elements ..... \n";
79:     for (int k = 0; k < _nelem; ++k)
80:     {
81:         cout << k << " : ";
82:         for (int i = 0; i < _ndof_e; ++i )
83:             cout << _ia[_ndof_e * k + i] << " ";
84:         cout << endl;
85:     }
86:     cout << "\n ..... Boundary (vertices) ..... \n";
87:     cout << " _bedges : " << _bedges << endl;
88:     return;
89: }
90:
91: void Mesh::DebugEdgeBased() const
92: {
93:     cout << "\n ##### Debug M E S H (edge based) #####\n";
94:     cout << "\n ..... Coordinates ..... \n";
95:     for (int k = 0; k < _nnode; ++k)
96:     {
97:         cout << k << " : " << _xc[2 * k] << " " << _xc[2 * k + 1] << endl;
98:     }
99:
100:    cout << "\n ..... edges ..... \n";
101:    for (int k = 0; k < _nedge; ++k)
102:    {
103:        cout << k << " : ";
104:        for (int i = 0; i < 2; ++i )
105:            cout << _edges[2 * k + i] << " ";
106:        cout << endl;
107:    }
108:
109:    cout << "\n ..... Elements (edges) ..... \n";
110:    assert(_nedge_e * _nelem == static_cast<int>(_ea.size()) );
111:    for (int k = 0; k < _nelem; ++k)
112:    {
113:        cout << k << " : ";
114:        for (int i = 0; i < _nedge_e; ++i )
115:            cout << _ea[_nedge_e * k + i] << " ";
116:        cout << endl;
117:    }
118:    cout << "\n ..... Boundary (edges) ..... \n";
119:    cout << " _ebedges : " << _ebedges << endl;
120:
121:    return;
122: }
123:
124: void Mesh::Write_ascii_matlab(std::string const &fname, std::vector<double> const &v
) const
125: {
126:     assert(Nnodes() == static_cast<int>(v.size())); // fits vector length to mesh
information?
127:
128:     ofstream fout(fname); // open file ASCII mode
129:     if ( !fout.is_open() )
130:     {

```

```

131:         cout << "\nFile " << fname << " has not been opened.\n\n" ;
132:         assert( fout.is_open() && "File not opened." );
133:     }
134:
135:     string const DELIMITER(" ");    // define the same delimiter as in matlab/ascii_
read*.m
136:     int const    OFFSET(1);        // convert C-indexing to matlab
137:
138:     // Write data: #nodes, #space dimensions, #elements, #vertices per element
139:     fout << Nnodes() << DELIMITER << Ndims() << DELIMITER << Nelems() << DELIMITER <
< NverticesElements() << endl;
140:
141:     // Write coordinates: x_k, y_k    in separate lines
142:     assert( Nnodes()*Ndims() == static_cast<int>(_xc.size()));
143:     for (int k = 0, kj = 0; k < Nnodes(); ++k)
144:     {
145:         for (int j = 0; j < Ndims(); ++j, ++kj)
146:         {
147:             fout << _xc[kj] << DELIMITER;
148:         }
149:         fout << endl;
150:     }
151:
152:     // Write connectivity: ia_k,0, ia_k,1 etc in separate lines
153:     assert( Nelems()*NverticesElements() == static_cast<int>(_ia.size()));
154:     for (int k = 0, kj = 0; k < Nelems(); ++k)
155:     {
156:         for (int j = 0; j < NverticesElements(); ++j, ++kj)
157:         {
158:             fout << _ia[kj] + OFFSET << DELIMITER;    // C to matlab
159:         }
160:         fout << endl;
161:     }
162:
163:     // Write vector
164:     for (int k = 0; k < Nnodes(); ++k)
165:     {
166:         fout << v[k] << endl;
167:     }
168:
169:     fout.close();
170:     return;
171: }
172:
173:
174: void Mesh::Export_scicomp(std::string const &basename) const
175: {
176:     //assert(Nnodes() == static_cast<int>(v.size())); // fits vector length to mes
h information?
177:     string const DELIMITER(" ");    // define the same delimiter as in matlab/ascii_
read*.m
178:     int const    OFFSET(0);
179:     {
180:         // Write coordinates into scicomp-file
181:         string fname(basename + "_coords.txt");
182:         ofstream fout(fname);        // open file ASCII mode
183:         if ( !fout.is_open() )
184:         {
185:             cout << "\nFile " << fname << " has not been opened.\n\n" ;
186:             assert( fout.is_open() && "File not opened." );
187:         }
188:
189:         fout << Nnodes() << endl;
190:         // Write coordinates: x_k, y_k    in separate lines
191:         assert( Nnodes()*Ndims() == static_cast<int>(_xc.size()));
192:         for (int k = 0, kj = 0; k < Nnodes(); ++k)
193:         {
194:             for (int j = 0; j < Ndims(); ++j, ++kj)

```

```

195:         {
196:             fout << _xc[kj] << DELIMETER;
197:         }
198:         fout << endl;
199:     }
200:     fout.close();
201:
202: }
203:
204: {
205:     // Write elements into scicomp-file
206:     string fname(basename + "_elements.txt");
207:     ofstream fout(fname); // open file ASCII mode
208:     if ( !fout.is_open() )
209:     {
210:         cout << "\nFile " << fname << " has not been opened.\n\n" ;
211:         assert( fout.is_open() && "File not opened." );
212:     }
213:
214:     fout << Nelems() << endl;
215:
216:     // Write connectivity: ia_k,0, ia_k,1 etc in separate lines
217:     assert( Nelems()*NverticesElements() == static_cast<int>(_ia.size()));
218:     for (int k = 0, kj = 0; k < Nelems(); ++k)
219:     {
220:         for (int j = 0; j < NverticesElements(); ++j, ++kj)
221:         {
222:             fout << _ia[kj] + OFFSET << DELIMETER; // C to matlab
223:         }
224:         fout << endl;
225:     }
226:     fout.close();
227: }
228:
229: return;
230: }
231:
232: /*
233: manjaro> matlab
234: MATLAB is selecting SOFTWARE_OPENGL rendering.
235: /usr/local/MATLAB/R2019a/bin/glnxa64/MATLAB: error while loading shared libraries:
libcrypt.so.1: cannot open shared object file: No such file or directory
236:
237: SOLUTION: sudo pacman -S libxcrypt-compat + reboot
238: */
239:
240: void Mesh::Visualize(vector<double> const &v) const
241: {
242:     // define external command
243:     const string exec_m("matlab -nosplash < visualize_results.m"); //
/ Matlab
244:     //const string exec_m("octave --no-window-system --no-gui visualize_results.m");
// Octave until version 6.3
245:     //const string exec_m("octave --no-gui --eval visualize_results.m"); // Octave s
ince version 6.4
246:     //const string exec_m("flatpak run org.octave.Octave visualize_results.m");
// Octave (flatpak): desktop GH
247:
248:     const string fname("uv.txt");
249:     Write_ascii_matlab(fname, v);
250:
251:     int ierror = system(exec_m.c_str()); // call ext
ernal command
252:
253:     if (ierror != 0)
254:     {
255:         cout << endl << "Check path to Matlab/octave on your system" << endl;
256:     }

```

```

257:     cout << endl;
258:     return;
259: }
260:
261: std::vector<int> Mesh::Index_DirichletNodes() const
262: {
263:     //vector<int> idx(_bedges); // copy
264:     // 2020-01-08
265:     // copy only the Dirichlet boundary nodes, not all boundary node
s.
266:     vector<int> idx(_bedges.size());
267:     size_t cnt=0;
268:     for (size_t kb = 0; kb < _bedges.size(); kb+=2 )
269:     {
270:         if (_sdedges.at(kb) <0 || _sdedges.at(kb+1) <0) // one neighboring subdomain
is negativ
271:         {
272:             idx[cnt] = _bedges[kb];
273:             ++cnt;
274:             idx[cnt] = _bedges[kb+1];
275:             ++cnt;
276:         }
277:     }
278:     idx.resize(cnt);
279:
280:     sort(idx.begin(), idx.end()); // sort
281:     idx.erase( unique(idx.begin(), idx.end()), idx.end() ); // remove duplicate data
282:
283:     return idx;
284: }
285:
286: // 2020-01-08
287: std::vector<int> Mesh::Index_BoundaryNodes() const
288: {
289:     vector<int> idx(_bedges); // copy
290:
291:     sort(idx.begin(), idx.end()); // sort
292:     idx.erase( unique(idx.begin(), idx.end()), idx.end() ); // remove duplicate data
293:
294:     return idx;
295: }
296:
297: // GH
298: // only correct for simplices
299: void Mesh::DeriveEdgeFromVertexBased_fast_2()
300: {
301:     assert(NedgesElements() == 3);
302:     assert(NverticesElements() == 3); // 3 vertices, 3 edges per element are assum
ed
303:
304:     // Store indices of all elements connected to a vertex
305:     vector<vector<int>> vertex2elems(_nnode, vector<int>(0));
306:     for (int k = 0; k < Nelems(); ++k)
307:     {
308:         for (int i = 0; i < 3; ++i)
309:         {
310:             vertex2elems[_ia[3 * k + i]].push_back(k);
311:         }
312:     }
313:     size_t max_neigh = 0; // maximal number of elements per vertex
314:     for (auto const &v : vertex2elems)
315:     {
316:         max_neigh = max(max_neigh, v.size());
317:     }
318:     //cout << endl << vertex2elems << endl;
319:
320:     // assign edges to elements
321:     _ea.clear(); // old data still in _ea without clear()

```

```

322:     _ea.resize(NedgesElements()*Nelems(), -1);
323:     // Derive the edges
324:     _edges.clear();
325:     _nedge = 0;
326:
327:     // convert also boundary edges
328:     unsigned int mbc(_bedges.size() / 2);      // number of boundary edges
329:     _ebedges.clear();
330:     _ebedges.resize(mbc, -1);
331:     vector<bool> bdir(_nnode, false);           // vector indicating boundary nodes
332:     //for (size_t kb = 0; kb < _bedges.size(); ++kb )
333:     //{
334:         //bdir.at(_bedges[kb]) = true;
335:     //}
336:     // 2020-01-08
337:     // GH ToDo: Selection of Dirichlet edges/nodes is still wrong
338:     //          Problem: _bedges is used externally instead of the local correct bdi
r.
339:     for (size_t kb = 0; kb < _bedges.size(); kb+=2 )
340:     {
341:         bool const booldir = _sdedges.at(kb) <0 || _sdedges.at(kb+1) <0; // one neig
hborning subdomain is negativ
342:         bdir.at(_bedges[kb]) = booldir;
343:         bdir.at(_bedges[kb+1]) = booldir;
344:     }
345:
346:     vector<int> vert_visited;                    // already visisted neighboring vertices o
f k
347:     vert_visited.reserve(max_neigh);            // avoids multiple (re-)allocations
348:     for (int k = 0; k < _nnode; ++k)           // vertex k
349:     {
350:         vert_visited.clear();
351:         auto const &elems = vertex2elems[k];    // element neighborhood
352:         int kedges = static_cast<int>(_edges.size()) / 2; // #edges before vertex k
is investigated
353:         //cout << elems << endl;
354:         // GH: problem, shared edges appear twice.
355:         int nneigh = elems.size();
356:         for (int ne = 0; ne < nneigh; ++ne)     // iterate through neighborhood
357:         {
358:             int e = elems[ne];                  // neighboring element e
359:             //cout << "e = " << e << endl;
360:             for (int i = 3 * e + 0; i < 3 * e + _nvert_e; ++i) // vertices of elem
ent e
361:             {
362:                 int const vert = _ia[i];
363:                 //cout << "vert: " << vert << " " << k << endl;
364:                 if (vert > k)
365:                 {
366:                     int ke = -1;
367:                     auto const iv = find(vert_visited.cbegin(), vert_visited.cend(),
vert);
368:                     if (iv == vert_visited.cend()) // vertex not yet visited
369:                     {
370:                         vert_visited.push_back(vert); // now, vertex vert is visite
d
371:                         _edges.push_back(k);          // add the new edge k->vert
372:                         _edges.push_back(vert);
373:
374:                         ke = _nedge;
375:                         ++_nedge;
376:                         // Is edge ke also a boundary edge?
377:                         if (bdir[k] && bdir[vert])
378:                         {
379:                             size_t kb = 0;
380:                             while (kb < _bedges.size() && (! ( _bedges[kb] == k && _
bedges[kb + 1] == vert) || ( _bedges[kb] == vert && _bedges[kb + 1] == k) )) )
381:                             {

```

```

382:         kb += 2;
383:     }
384:     if (kb < _bedges.size())
385:     {
386:         _ebedges[kb / 2] = ke;
387:     }
388: }
389: }
390: else
391: {
392:     int offset = iv - vert_visited.cbegin();
393:     ke = kedges + offset;
394: }
395: // assign that edge to the edges based connectivity of element e
396: auto ip = find_if(_ea.begin() + 3 * e, _ea.begin() + 3 * (e + 1)
,
397:         [] (int v) -> bool {return v < 0;} );
398: //cout << ip-_ea.begin()+3*e << " " << *ip << endl;
399: assert(ip != _ea.cbegin() + 3 * (e + 1)); // data error !
400: *ip = ke;
401: }
402: }
403: }
404: }
405:
406: assert( Mesh::Check_array_dimensions() );
407: return;
408: }
409: // HG
410:
411: // GH
412: // only correct for simplices
413: void Mesh::DeriveEdgeFromVertexBased_fast()
414: {
415:     assert(NedgesElements() == 3);
416:     assert(NverticesElements() == 3); // 3 vertices, 3 edges per element are assum
ed
417:
418:     // Store indices of all elements connected to a vertex
419:     vector<vector<int>> vertex2elems(_nnode, vector<int>(0));
420:     for (int k = 0; k < Nelems(); ++k)
421:     {
422:         for (int i = 0; i < 3; ++i)
423:         {
424:             vertex2elems[_ia[3 * k + i]].push_back(k);
425:         }
426:     }
427:     size_t max_neigh = 0; // maximal number of elements per vertex
428:     for (auto const &v : vertex2elems)
429:     {
430:         max_neigh = max(max_neigh, v.size());
431:     }
432:     //cout << endl << vertex2elems << endl;
433:
434:     // assign edges to elements
435:     _ea.clear(); // old data still in _ea without clear()
436:     _ea.resize(NedgesElements()*Nelems(), -1);
437:     // Derive the edges
438:     _edges.clear();
439:     _nedge = 0;
440:     vector<int> vert_visited; // already visisted neighboring vertices o
f k
441:     vert_visited.reserve(max_neigh); // avoids multiple (re-)allocations
442:     for (int k = 0; k < _nnode; ++k) // vertex k
443:     {
444:         vert_visited.clear();
445:         auto const &elems = vertex2elems[k]; // element neighborhood
446:         int kedges = static_cast<int>(_edges.size()) / 2; // #edges before vertex k

```

```

is investigated
447:         //cout << elems << endl;
448:         // GH: problem, shared edges appear twice.
449:         int nneigh = elems.size();
450:         for (int ne = 0; ne < nneigh; ++ne)           // iterate through neighborhood
451:         {
452:             int e = elems[ne];                       // neighboring element e
453:             //cout << "e = " << e << endl;
454:             for (int i = 3 * e + 0; i < 3 * e + _nvert_e; ++i) // vertices of elem
ent e
455:             {
456:                 int const vert = _ia[i];
457:                 //cout << "vert: " << vert << " " << k << endl;
458:                 if (vert > k)
459:                 {
460:                     int ke = -1;
461:                     auto const iv = find(vert_visited.cbegin(), vert_visited.cend(),
vert);
462:                     if (iv == vert_visited.cend()) // vertex not yet visited
463:                     {
464:                         vert_visited.push_back(vert); // now, vertex vert is visite
d
465:                         _edges.push_back(k);           // add the new edge k->vert
466:                         _edges.push_back(vert);
467:
468:                         ke = _nedge;
469:                         ++_nedge;
470:                     }
471:                     else
472:                     {
473:                         int offset = iv - vert_visited.cbegin();
474:                         ke = kedges + offset;
475:                     }
476:                     // assign that edge to the edges based connectivity of element e
477:                     auto ip = find_if(_ea.begin() + 3 * e, _ea.begin() + 3 * (e + 1)
,
478:                                     [] (int v) -> bool {return v < 0; } );
479:                     //cout << ip-_ea.begin()+3*e << " " << *ip << endl;
480:                     assert(ip != _ea.cbegin() + 3 * (e + 1)); // data error !
481:                     *ip = ke;
482:                 }
483:             }
484:         }
485:     }
486:
487:     // convert also boundary edges
488:     unsigned int mbc(_bedges.size() / 2);           // number of boundary edges
489:     _ebedges.clear();
490:     _ebedges.resize(mbc, -1);
491:     for (unsigned int kb = 0; kb < mbc; ++kb)
492:     {
493:         int const v1 = min(_bedges[2 * kb], _bedges[2 * kb + 1]); // vertices
494:         int const v2 = max(_bedges[2 * kb], _bedges[2 * kb + 1]);
495:
496:         size_t e = 0;
497:         // ascending vertex indices for each edge e in _edges
498:         while (e < _edges.size() && (_edges[e] != v1 || _edges[e + 1] != v2) )
499:         {
500:             e += 2; // next edge
501:         }
502:         assert(e < _edges.size()); // error: no edge found
503:         _ebedges[kb] = e / 2;      // index of edge
504:     }
505:
506:
507:     assert( Mesh::Check_array_dimensions() );
508:     return;
509: }

```

```

510: // HG
511:
512:
513: #include <utility> // pair
514:
515: void Mesh::DeriveEdgeFromVertexBased_slow()
516: {
517:     assert(NedgesElements() == 3);
518:     assert(NverticesElements() == 3); // 3 vertices, 3 edges per element are assum
ed
519:
520:     _ea.resize(NedgesElements()*Nelems());
521:     vector< pair<int, int> > edges(0);
522:     int nedges = 0;
523:
524:     for (int k = 0; k < Nelems(); ++k)
525:     {
526:         array < int, 3 + 1 > ivert{{ _ia[3 * k], _ia[3 * k + 1], _ia[3 * k + 2], _ia
[3 * k] }};
527:
528:         for (int i = 0; i < 3; ++i)
529:         {
530:             pair<int, int> e2; // this edge
531:             if (ivert[i] < ivert[i + 1]) // guarantee ascending order
532:             {
533:                 e2 = make_pair(ivert[i], ivert[i + 1]);
534:             }
535:             else
536:             {
537:                 e2 = make_pair(ivert[i + 1], ivert[i]);
538:             }
539:
540:             int eki(-1); // global index of this edge
541:             auto ip = find(edges.cbegin(), edges.cend(), e2);
542:             if ( ip == edges.cend() ) // edge not found ==> add that edge
543:             {
544:                 //cout << "found edge\n";
545:                 edges.push_back(e2); // add the new edge
546:                 eki = nedges; // index of this new edge
547:                 ++nedges;
548:             }
549:
550:             else
551:             {
552:                 eki = ip - edges.cbegin(); // index of the edge found
553:             }
554:             _ea[3 * k + i] = eki; // set edge index in edge based connectiv
ity
555:         }
556:     }
557:
558:     assert( nedges == static_cast<int>(edges.size()) );
559:     _nedge = nedges; // set the member variable for number of edg
es
560:     _edges.resize(2 * nedges); // allocate memory for edge storage
561:     for (int k = 0; k < nedges; ++k)
562:     {
563:         _edges[2 * k] = edges[k].first;
564:         _edges[2 * k + 1] = edges[k].second;
565:     }
566:
567:     // convert also boundary edges
568:     unsigned int mbc(_bedges.size() / 2); // number of boundary edges
569:     //cout << "AA " << mbc << endl;
570:     _ebedges.resize(mbc);
571:     for (unsigned int kb = 0; kb < mbc; ++kb)
572:     {
573:         const auto vv1 = make_pair(_bedges[2 * kb], _bedges[2 * kb + 1]); // both

```

```

574:         const auto vv2 = make_pair(_bedges[2 * kb + 1], _bedges[2 * kb  ]); // dire
ctions of edge
575:         auto ip1 = find(edges.cbegin(), edges.cend(), vv1);
576:         if (ip1 == edges.cend())
577:         {
578:             ip1 = find(edges.cbegin(), edges.cend(), vv2);
579:             assert(ip1 != edges.cend()); // stop because inconsistency (bou
ndary edge has to be included in edges)
580:         }
581:         _ebedges[kb] = ip1 - edges.cbegin(); // index of edge
582:     }
583:
584:     assert( Mesh::Check_array_dimensions() );
585:     return;
586: }
587:
588: void Mesh::DeriveVertexFromEdgeBased()
589: {
590:     assert(NedgesElements() == 3);
591:     assert(NverticesElements() == 3); // 3 vertices, 3 edges per element are assum
ed
592:
593:     _ia.resize(NedgesElements()*Nelems()); // NN
594:
595:     for (int k = 0; k < Nelems(); ++k)
596:     {
597:         //vector<int> ivert(6); // indices of vertices
598:         array<int, 6> ivert; // indices of vertices
599:         for (int j = 0; j < 3; ++j) // local edges
600:         {
601:             int const iedg = _ea[3 * k + j]; // index of one edge in triangle
602:             ivert[2 * j  ] = _edges[2 * iedg  ]; // first vertex of edge
603:             ivert[2 * j + 1] = _edges[2 * iedg + 1]; // second vertex of edge
604:         }
605:         sort(ivert.begin(), ivert.end()); // unique indices are needed
606:         auto const ip = unique(ivert.begin(), ivert.end());
607:         assert( ip - ivert.begin() == 3 );
608:         for (int i = 0; i < 3; ++i) // vertex based element connectivity
609:         {
610:             _ia[3 * k + i] = ivert[i];
611:         }
612:     }
613:
614:     // convert also boundary edges
615:     unsigned int mbc(_ebedges.size()); // number of boundary edges
616:     _bedges.resize(2 * mbc);
617:     for (unsigned int k = 0; k < mbc; ++k)
618:     {
619:         const auto ke = _ebedges[k]; // edge index
620:         _bedges[2 * k  ] = _edges[2 * ke  ];
621:         _bedges[2 * k + 1] = _edges[2 * ke + 1];
622:     }
623:
624:
625:     return;
626: }
627:
628: // Member Input: vertices of each element : _ia[_nelem*_nvert_e] stores as 1D array
629: //          number of vertices per element : _nvert_e
630: //          global number of elements: _nelem
631: //          global number of vertices: _nnode
632: vector<vector<int>> Mesh::Node2NodeGraph_2() const
633: {
634:     vector<vector<int>> v2v(_nnode, vector<int>(0)); // stores the vertex to vert
ex connections
635:
636:     ///-----
637:     vector<int> cnt(_nnode, 0);

```

```

638:     for (size_t i = 0; i < _ia.size(); ++i) ++cnt[_ia[i]]; // determine number of e
ntries per vertex
639:     for (size_t k = 0; k < v2v.size(); ++k)
640:     {
641:         v2v[k].resize(_nvert_e * cnt[k]); // and allocate the memory
for that vertex
642:         cnt[k] = 0;
643:     }
644:     ////-----
645:
646:     for (int e = 0; e < _nelem; ++e)
647:     {
648:         int const basis = e * _nvert_e; // start of vertex connecti
vity of element e
649:         for (int k = 0; k < _nvert_e; ++k)
650:         {
651:             int const v = _ia[basis + k];
652:             for (int l = 0; l < _nvert_e; ++l)
653:             {
654:                 v2v[v][cnt[v]] = _ia[basis + l];
655:                 ++cnt[v];
656:             }
657:         }
658:     }
659:     // finally cnt[v]==v2v[v].size() has to hold for all v!
660:
661:     // guarantee unique, ascending sorted entries per vertex
662:     for (size_t v = 0; v < v2v.size(); ++v)
663:     {
664:         sort(v2v[v].begin(), v2v[v].end());
665:         auto ip = unique(v2v[v].begin(), v2v[v].end());
666:         v2v[v].erase(ip, v2v[v].end());
667:         //v2v[v].shrink_to_fit(); // automatically done when copied at return
668:     }
669:
670:     return v2v;
671: }
672:
673: vector<vector<int>> Mesh::Node2NodeGraph_1() const
674: {
675:     vector<vector<int>> v2v(_nnode, vector<int>(0)); // stores the vertex to vert
ex connections
676:
677:     for (int e = 0; e < _nelem; ++e)
678:     {
679:         int const basis = e * _nvert_e; // start of vertex connecti
vity of element e
680:         for (int k = 0; k < _nvert_e; ++k)
681:         {
682:             int const v = _ia[basis + k];
683:             for (int l = 0; l < _nvert_e; ++l)
684:             {
685:                 v2v[v].push_back(_ia[basis + l]);
686:             }
687:         }
688:     }
689:     // guarantee unique, ascending sorted entries per vertex
690:     for (size_t v = 0; v < v2v.size(); ++v)
691:     {
692:         sort(v2v[v].begin(), v2v[v].end());
693:         auto ip = unique(v2v[v].begin(), v2v[v].end());
694:         v2v[v].erase(ip, v2v[v].end());
695:         //v2v[v].shrink_to_fit(); // automatically done when copied at return
696:     }
697:
698:     return v2v;
699: }
700:

```

```

701:
702: Mesh::Mesh(std::string const &fname)
703:     : Mesh(2, 3, 3, 3) // two dimensions, 3 vertices, 3 dofs, 3 edges per element
704: {
705:     ReadVertexBasedMesh(fname);
706:     DeriveEdgeFromVertexBased(); // Generate also the edge based information
707:     //cout << " JJJJJJJJJ\n";
708:     //DeriveEdgeFromVertexBased();
709:     //cout << " KKKKKKKKKKK\n";
710:     ////exit(-1);
711: }
712:
713: void Mesh::ReadVertexBasedMesh(std::string const &fname)
714: {
715:     ifstream ifs(fname);
716:     if (!(ifs.is_open() && ifs.good()))
717:     {
718:         cerr << "Mesh::ReadVertexBasedMesh: Error cannot open file " << fname << endl;
1;
719:         assert(ifs.is_open());
720:     }
721:
722:     int const OFFSET(1); // Matlab to C indexing
723:     cout << "ASCII file " << fname << " opened" << endl;
724:
725:     // Read some mesh constants
726:     int nnode, ndim, nele, nvert_e;
727:     ifs >> nnode >> ndim >> nele >> nvert_e;
728:     cout << nnode << " " << ndim << " " << nele << " " << nvert_e << endl;
729:     assert(ndim == 2 && nvert_e == 3);
730:
731:     // Allocate memory
732:     Resize_Coords(nnode, ndim); // coordinates in 2D [nnode][ndim]
733:     Resize_Connectivity(nele, nvert_e); // connectivity matrix [nele][nvert
]
734:
735:     // Read coordinates
736:     auto &xc = GetCoords();
737:     for (int k = 0; k < nnode * ndim; ++k)
738:     {
739:         ifs >> xc[k];
740:     }
741:
742:     // Read connectivity
743:     auto &ia = GetConnectivity();
744:     for (int k = 0; k < nele * nvert_e; ++k)
745:     {
746:         ifs >> ia[k];
747:         ia[k] -= OFFSET; // Matlab to C indexing
748:     }
749:
750:     // additional read of boundary information (only start/end point)
751:     int nbedges;
752:     ifs >> nbedges;
753:
754:     _bedges.resize(nbedges * 2);
755:     for (int k = 0; k < nbedges * 2; ++k)
756:     {
757:         ifs >> _bedges[k];
758:         _bedges[k] -= OFFSET; // Matlab to C indexing
759:     }
760:     // 2020-01-08 Check
761:     int const DUMMY{-9876};
762:     _sdedges.resize(nbedges * 2, DUMMY);
763:     if (!ifs.eof())
764:     {
765:         cout << nbedges << endl; // change to while-loop
766:         for (int k = 0; k < nbedges * 2; ++k)

```

```

767:         {
768:             ifs >> _sdedges.at(k);
769:             _sdedges[k] -= OFFSET; // Matlab to C indexing
770:         }
771:         //assert(ifs.eof());
772:         assert(count(_sdedges.cbegin(), _sdedges.cend(), DUMMY) == 0);
773:     }
774:     else
775:     {
776:         cout << "\n ##### no subdomain info of edges available! #####\n";
777:     }
778:     cout << "\n End of File read " << _sdedges.at(2*nbedges-2) << " " << _sdedges.at
(2*nbedges-1) << "\n" << endl;
779:
780:     return;
781: }
782:
783: bool Mesh::Check_array_dimensions() const
784: {
785:     bool b_ia = static_cast<int>(_ia.size() / _nvert_e) == _nelem;
786:     if (!b_ia) cerr << "misfit: _nelem vs. _ia" << endl;
787:
788:     bool b_xc = static_cast<int>(_xc.size() / _ndim) == _nnode;
789:     if (!b_xc) cerr << "misfit: _nnode vs. _xc" << endl;
790:
791:     bool b_ea = static_cast<int>(_ea.size() / _nedge_e) == _nelem;
792:     if (!b_ea) cerr << "misfit: _nelem vs. _ea" << endl;
793:
794:     bool b_ed = static_cast<int>(_edges.size() / 2) == _nedge;
795:     if (!b_ed) cerr << "misfit: _nedge vs. _edges" << endl;
796:
797:
798:     return b_ia && b_xc && b_ea && b_ed;
799: }
800:
801: void Mesh::Del_EdgeConnectivity()
802: {
803:     _nedge = 0; //!< number of edges in mesh
804:     _edges.resize(0); //!< edges of mesh (vertices ordered ascending)
805:     _edges.shrink_to_fit();
806:     _ea.resize(0); //!< edge based element connectivity
807:     _ea.shrink_to_fit();
808:     _ebedges.resize(0); //!< boundary edges [nbedges]
809:     _ebedges.shrink_to_fit();
810:     return;
811: }
812:
813:
814:
815: // #####
816:
817: RefinedMesh::RefinedMesh(Mesh const &cmesh, std::vector<bool> const &ibref)
818: //: Mesh(cmesh), _cmesh(cmesh), _ibref(ibref), _nref(0), _vfathers(0)
819: : Mesh(cmesh), _ibref(ibref), _nref(0), _vfathers(0)
820: {
821:     if (_ibref.size() == 0) // refine all elements
822:     {
823:         //
824:         RefineAllElements();
825:     }
826:     else
827:     {
828:         cout << endl << " Adaptive Refinement not implemented yet." << endl;
829:         assert(_ibref.size() != 0);
830:     }
831: }
832:
833: RefinedMesh::~RefinedMesh()

```

```

834: {}
835:
836: Mesh RefinedMesh::RefineElements(std::vector<bool> const & /*ibref*/)
837: {
838:     Mesh new_mesh(_ndim, _nvert_e, _ndof_e, _nedge_e);
839:     cout << " NOT IMPLEMENTED: Mesh::RefineElements" << endl;
840:
841:     //// initialize new coords with the old one
842:     //auto new_coords = new_mesh.GetCoords();
843:     //new_coords = _xc; // copy coordinates from old mesh
844:
845:     //// access vertex connectivite, edge connectivi and edge information of new mesh
846:     //auto new_ia = new_mesh.GetConnectivity();
847:     //auto new_ea = new_mesh.GetEdgeConnectivity();
848:     //auto new_edges = new_mesh.GetEdges();
849:
850:     //// storing the parents of edges and vertices
851:
852:
853:     //assert( new_ia.size() == new_ea.size() );
854:     //new_mesh.SetNnode( new_coords.size() );
855:     //new_mesh.SetNelem( new_ia.size()/3 );
856:     //new_mesh._nedge = new_edges.size()/2;
857:
858:     return new_mesh;
859: }
860:
861: //JF
862: void RefinedMesh::RefineAllElements(int nref)
863: {
864:     cout << "\n##### Refine Mesh " << nref << " times ";
865:     double tstart = clock();
866:     DeriveEdgeFromVertexBased(); // ensure that edge information is availab
le
867:
868:     for (int kr = 0; kr < nref; ++kr)
869:     {
870:         //DeriveEdgeFromVertexBased(); // ensure that edge information is a
available // GH: not needed in each loop
871:
872:         auto old_ea(_ea); // save old edge connectivity
873:         auto old_edges(_edges); // save old edges
874:         auto old_nedges(Nedges());
875:         auto old_nnodes(Nnodes());
876:         auto old_nelems(Nelems());
877:
878:         // the new vertices will be appended to the coordinates in _xc
879:
880:         vector<int> edge_sons(2 * old_nedges); // 2 sons for each edge
881:
882:         // -- Derive the fine edges --
883:         int new_nedge = 2 * old_nedges + 3 * old_nelems; // #edges in new mesh
884:         int new_nelem = 4 * old_nelems; // #elements in new mesh
885:         int new_nnode = old_nnodes + old_nedges; // #nodes in new mesh
886:
887:         _xc.reserve(2 * new_nnode);
888:         // store the 2 fathers of each vertex (equal fathers denote original coarse
vertex)
889:         _vfathers.resize(2 * old_nnodes);
890:         for (int vc = 0; vc < old_nnodes; ++vc)
891:         {
892:             _vfathers[2 * vc] = vc; // equal fathers denote original coarse ver
tex
893:             _vfathers[2 * vc + 1] = vc;
894:         }
895:
896:         _ia.clear();
897:         _ea.clear();

```

```

898:     _ea.resize(new_nelem * 3);
899:     _edges.clear();
900:     _edges.resize(2 * new_nedge);           // vertices of edges [v_0, v_1; v_0, v
_1; ...]
901:     vector<int> e_son(2 * old_nedges); // sons of coarse edges [s_0, s_1; s_0, s
_1; ...]
902:
903:     // split all coarse edges and append the new nodes
904:     int kf = 0;                             // index of edges in fine mesh
905:     int vf = old_nnodes;                     // index of new vertex in fine grid
906:     for (int kc = 0; kc < old_nedges; ++kc) // index of edges in coarse mesh
907:     {
908:         //
909:         int v1 = old_edges[2 * kc];           // vertices of old edge
910:         int v2 = old_edges[2 * kc + 1];
911:         // append coordinates of new vertex
912:         double xf = 0.5 * ( _xc[2 * v1 ] + _xc[2 * v2 ] );
913:         double yf = 0.5 * ( _xc[2 * v1 + 1] + _xc[2 * v2 + 1] );
914:         _xc.push_back(xf);
915:         _xc.push_back(yf);
916:         // fathers of vertex vf
917:         _vfathers.push_back(v1);
918:         _vfathers.push_back(v2);
919:
920:         // split old edge into two edges
921:         _edges[2 * kf ] = v1;                 // coarse vertex 1
922:         _edges[2 * kf + 1] = vf;              // to new fine vertex
923:         e_son[2 * kc ] = kf;                  // son edge
924:         ++kf;
925:         _edges[2 * kf ] = vf;                 // new fine vertex
926:         _edges[2 * kf + 1] = v2;              // to coarse vertex 2
927:         e_son[2 * kc + 1] = kf;              // son edge
928:
929:         ++vf;
930:         ++kf;
931:     }
932:     _xc.shrink_to_fit();
933:     _vfathers.shrink_to_fit();
934:
935:     // -- derive the fine mesh elements --
936:     // creates additional fine edges
937:
938:     for (int kc = 0; kc < old_nelems; ++kc) // index of elements in coarse mes
h
939:     {
940:         array<array<int, 3>, 3 * 2> boundary; // fine scale vertices and edges a
s boundary of old element
941:         //boundary[ ][0], boundary[ ][1] ..vertices boundary[ ][2] edge
942:
943:         for (int j = 0; j < 3; ++j)           // each edge in element
944:         {
945:             int ce = old_ea[3 * kc + j];      // coarse edge number
946:
947:             int s1 = e_son[2 * ce ];           // son edges of that coarse edge
948:             int s2 = e_son[2 * ce + 1];
949:             boundary[2 * j][2] = s1;           //add boundary edge
950:             boundary[2 * j][0] = _edges[2 * s1 + 0];
951:             boundary[2 * j][1] = _edges[2 * s1 + 1];
952:             if (boundary[2 * j][0] > boundary[2 * j][1]) swap(boundary[2 * j][0]
, boundary[2 * j][1]); // fine vertices always in 2nd entry
953:             boundary[2 * j + 1][2] = s2;       //add boundary edge
954:             boundary[2 * j + 1][0] = _edges[2 * s2 + 0];
955:             boundary[2 * j + 1][1] = _edges[2 * s2 + 1];
956:             if (boundary[2 * j + 1][0] > boundary[2 * j + 1][1]) swap(boundary[2
* j + 1][0], boundary[2 * j + 1][1]);
957:         }
958:
959:         sort(boundary.begin(), boundary.end()); // sort -> edges wit

```

h same coarse vertex will be neighbors

```

960:
961:         int interior_1 = 2 * old_nedges + kc * 3;    // add interior edges
962:         int interior_2 = 2 * old_nedges + kc * 3 + 1;
963:         int interior_3 = 2 * old_nedges + kc * 3 + 2;
964:
965:         _edges[interior_1 * 2    ] = boundary[0][1]; // add interior edges
966:         _edges[interior_1 * 2 + 1] = boundary[1][1];
967:
968:         _edges[interior_2 * 2    ] = boundary[2][1];
969:         _edges[interior_2 * 2 + 1] = boundary[3][1];
970:
971:         _edges[interior_3 * 2    ] = boundary[4][1];
972:         _edges[interior_3 * 2 + 1] = boundary[5][1];
973:
974:         _ea[kc * 3 * 4    ] = boundary[0][2];        // add 4 new elements with 3
edges for every old element
975:         _ea[kc * 3 * 4 + 1] = boundary[1][2];
976:         _ea[kc * 3 * 4 + 2] = interior_1;
977:
978:         _ea[kc * 3 * 4 + 3] = boundary[2][2];
979:         _ea[kc * 3 * 4 + 4] = boundary[3][2];
980:         _ea[kc * 3 * 4 + 5] = interior_2;
981:
982:         _ea[kc * 3 * 4 + 6] = boundary[4][2];
983:         _ea[kc * 3 * 4 + 7] = boundary[5][2];
984:         _ea[kc * 3 * 4 + 8] = interior_3;
985:
986:         _ea[kc * 3 * 4 + 9] = interior_1;
987:         _ea[kc * 3 * 4 + 10] = interior_2;
988:         _ea[kc * 3 * 4 + 11] = interior_3;
989:     }
990:
991: // GH: ToDo: _bedges has to updated for the new mesh //!< boundary edges [nbedges]
[2] storing start/end vertex
992: //      Pass the refinement information to the boundary edges (edge based)
993:     auto old_eedges(_eedges);        // save original boundary edges [nbedg
es] (edge based storage)
994:     unsigned int old_nbedges(old_eedges.size());
995:
996:     _eedges.resize(2 * old_nbedges);  // each old boundary edge will be bisec
ted
997:     unsigned int kn = 0;                // index of new boundary edges
998:     for (unsigned int ke = 0; ke < old_nbedges; ++ke) // index of old boundary
edges
999:     {
1000:         const auto kc = old_eedges[ke];
1001:         _eedges[kn] = e_son[2 * kc    ];
1002:         ++kn;
1003:         _eedges[kn] = e_son[2 * kc + 1];
1004:         ++kn;
1005:     }
1006: // HG
1007:     // set new mesh parameters
1008:     SetNelem(new_nelem);
1009:     SetNnode(new_nnode);
1010:     SetNedge(new_nedge);
1011:
1012: #ifdef CUTHILL_MCKEE
1013:     {
1014: // Cuthill-McKee reordering
1015: //      Increases mesh generation time by factor 5 - but solver is faster.
1016:         auto const perm = cuthill_mckee_reordering(_edges);
1017:         PermuteVertices_EdgeBased(perm);
1018:     }
1019: #endif
1020:
1021:     DeriveVertexFromEdgeBased();

```

```

1022:         assert( RefinedMesh::Check_array_dimensions() );
1023:
1024:         ++_nref;                                     // track the number of refinements
1025:     }
1026:
1027:     double duration = (clock() - tstart) / CLOCKS_PER_SEC;
1028:     cout << "finished in " << duration << " sec.      #####\n";
1029:
1030:     return;
1031: }
1032:
1033:
1034: void Mesh::PermuteVertices_EdgeBased(vector<int> const &old2new)
1035: {
1036:     //      permute vertices _edges
1037:     auto const edges_old(_edges);
1038:     for (size_t k = 0; k < _edges.size(); k += 2)
1039:     {
1040:         _edges[k] = old2new[edges_old[k]];
1041:         _edges[k + 1] = old2new[edges_old[k + 1]];
1042:         if (_edges[k] > _edges[k + 1])
1043:             swap(_edges[k], _edges[k + 1]);
1044:     }
1045:     //      permute coordinates
1046:     auto const coord_old(_xc);
1047:     for (size_t k = 0; k < _xc.size() / 2; ++k)
1048:     {
1049:         _xc[2 * old2new[k]] = coord_old[2 * k];
1050:         _xc[2 * old2new[k] + 1] = coord_old[2 * k + 1];
1051:     }
1052:     return;
1053: }
1054:
1055:
1056: void RefinedMesh::PermuteVertices_EdgeBased(vector<int> const &old2new)
1057: {
1058:     Mesh::PermuteVertices_EdgeBased(old2new);
1059:     //      permute fathers of a vertex
1060:     auto const old_fathers(_vfathers);
1061:     for (size_t k = 0; k < _vfathers.size() / 2; ++k)
1062:     {
1063:         _vfathers[2 * old2new[k]] = old_fathers[2 * k];
1064:         _vfathers[2 * old2new[k] + 1] = old_fathers[2 * k + 1];
1065:     }
1066:     return;
1067: }
1068:
1069:
1070: bool RefinedMesh::Check_array_dimensions() const
1071: {
1072:     const bool bp = Mesh::Check_array_dimensions();
1073:
1074:     const bool bvf = (static_cast<int>(_vfathers.size()) / 2 == Nnodes());
1075:
1076:     return bp && bvf;
1077: }
1078: }
1079: // #####
1080:
1081:
1082: gMesh_Hierarchy::gMesh_Hierarchy(Mesh const &cmesh, int const nlevel)
1083: : _gmesh(max(1, nlevel))
1084: {
1085:     _gmesh[0] = make_shared<Mesh>(cmesh);
1086:     for (int lev = 1; lev < nlevel; ++lev)
1087:     {
1088:         _gmesh.at(lev) = make_shared<RefinedMesh>(*_gmesh.at(lev - 1));
1089:         //auto vv=_gmesh[lev]->GetFathersOfVertices();

```

```

1090:         //cout << " :: "<< vv.size() <<endl;
1091:     }
1092:     for (size_t lev = 0; lev < _gmesh.size(); ++lev)
1093:     {
1094:         _gmesh[lev]->Del_EdgeConnectivity();
1095:     }
1096: }
1097:
1098:
1099: // #####
1100: Mesh_2d_3_square::Mesh_2d_3_square(int nx, int ny, int myid, int procx, int procy)
1101:     : Mesh(2, 3, 3, 3), // two dimensions, 3 vertices, 3 dofs, 3 edges per element
1102:     _myid(myid), _procx(procx), _procy(procy), _neigh{{ -1, -1, -1, -1}}, _color(0
),
1103:     _xl(0.0), _xr(1.0), _yb(0.0), _yt(1.0), _nx(nx), _ny(ny)
1104: {
1105:     //void IniGeom(int const myid, int const procx, int const procy, int neigh[], in
t &color)
1106:     int const ky = _myid / _procx;
1107:     int const kx = _myid % _procx; // MOD(myid,procx)
1108:     // Determine the neighbors of domain/rank myid
1109:     _neigh[0] = (ky == 0) ? -1 : _myid - _procx; // South
1110:     _neigh[1] = (kx == _procx - 1) ? -1 : _myid + 1; // East
1111:     _neigh[2] = (ky == _procy - 1) ? -1 : _myid + _procx; // North
1112:     _neigh[3] = (kx == 0) ? -1 : _myid - 1; // West
1113:
1114:     _color = (kx + ky) & 1 ;
1115:
1116:     // subdomain is part of unit square
1117:     double const hx = 1. / _procx;
1118:     double const hy = 1. / _procy;
1119:     _xl = kx * hx; // left
1120:     _xr = (kx + 1) * hx; // right
1121:     _yb = ky * hy; // bottom
1122:     _yt = (ky + 1) * hy; // top
1123:
1124:     // Calculate coordinates
1125:     int const nnode = (_nx + 1) * (_ny + 1); // number of nodes
1126:     Resize_Coords(nnode, 2); // coordinates in 2D [nnode][ndim]
1127:     GetCoordsInRectangle(_nx, _ny, _xl, _xr, _yb, _yt, GetCoords().data());
1128:
1129:     // Calculate element connectivity (linear triangles)
1130:     int const nelelem = 2 * _nx * _ny; // number of elements
1131:     Resize_Connectivity(nelelem, 3); // connectivity matrix [nelem][3]
1132:     GetConnectivityInRectangle(_nx, _ny, GetConnectivity().data());
1133:
1134:     return;
1135: }
1136:
1137: Mesh_2d_3_square::~Mesh_2d_3_square()
1138: {}
1139:
1140:
1141: void Mesh_2d_3_square::SetU(std::vector<double> &u) const
1142: {
1143:     int dx = _nx + 1;
1144:     for (int j = 0; j <= _ny; ++j)
1145:     {
1146:         int k = j * dx;
1147:         for (int i = 0; i <= _nx; ++i, ++k)
1148:         {
1149:             u[k] = 0.0;
1150:         }
1151:     }
1152: }
1153:
1154:
1155: void Mesh_2d_3_square::SetF(std::vector<double> &f) const

```

```

1156: {
1157:     int dx    = _nx + 1;
1158:     for (int j = 0; j <= _ny; ++j)
1159:     {
1160:         int k = j * dx;
1161:         for (int i = 0; i <= _nx; ++i, ++k)
1162:         {
1163:             f[k] = 1.0;
1164:         }
1165:     }
1166:
1167: }
1168:
1169:
1170: std::vector<int> Mesh_2d_3_square::Index_DirichletNodes() const
1171: {
1172:     int const dx = 1,
1173:             dy = _nx + 1,
1174:             bl  = 0,
1175:             br  = _nx,
1176:             tl  = _ny * (_nx + 1),
1177:             tr  = (_ny + 1) * (_nx + 1) - 1;
1178:     int const start[4] = { bl, br, tl, bl};
1179:     int const end[4]   = { br, tr, tr, tl};
1180:     int const step[4]  = { dx, dy, dx, dy};
1181:
1182:     vector<int> idx(0);
1183:     for (int j = 0; j < 4; j++)
1184:     {
1185:         if (_neigh[j] < 0)
1186:         {
1187:             for (int i = start[j]; i <= end[j]; i += step[j])
1188:             {
1189:                 idx.push_back(i);           // node i is Dirichlet node
1190:             }
1191:         }
1192:     }
1193:     // remove multiple elements
1194:     sort(idx.begin(), idx.end());           // sort
1195:     idx.erase( unique(idx.begin(), idx.end()), idx.end() ); // remove duplicate data
1196:
1197:     return idx;
1198: }
1199:
1200: vector<int> Mesh_2d_3_square::Index_BoundaryNodes() const
1201: {
1202:     cerr << "Warning: Funktion not implemented. " << __FILE__ << "." << __LINE__ <<
endl;
1203:     return Index_BoundaryNodes();
1204: }
1205:
1206:
1207:
1208:
1209: void Mesh_2d_3_square::SaveVectorP(std::string const &name, vector<double> const &u)
const
1210: {
1211:     // construct the file name for subdomain myid
1212:     const string tmp( std::to_string(_myid / 100) + to_string((_myid % 100) / 10) +
to_string(_myid % 10) );
1213:
1214:     const string namep = name + "." + tmp;
1215:     ofstream ff(namep.c_str());
1216:     ff.precision(6);
1217:     ff.setf(ios::fixed, ios::floatfield);
1218:
1219:     // assumes tensor product grid in unit square; rowise numbered (as generated in
class constructor)

```

```
1220:      // output is provided for tensor product grid visualization ( similar to Matlab-
surf() )
1221:      auto const &xc = GetCoords();
1222:      int k = 0;
1223:      for (int j = 0; j <= _ny; ++j)
1224:      {
1225:          for (int i = 0; i <= _nx; ++i, ++k)
1226:              ff << xc[2 * k + 0] << " " << xc[2 * k + 1] << " " << u[k] << endl;
1227:          ff << endl;
1228:      }
1229:
1230:      ff.close();
1231:      return;
1232: }
1233:
1234: void Mesh_2d_3_square::GetCoordsInRectangle(int const nx, int const ny,
1235:      double const xl, double const xr, double const yb, double const yt,
1236:      double xc[])
1237: {
1238:     const double hx = (xr - xl) / nx,
1239:                 hy = (yt - yb) / ny;
1240:
1241:     int k = 0;
1242:     for (int j = 0; j <= ny; ++j)
1243:     {
1244:         const double y0 = yb + j * hy;
1245:         for (int i = 0; i <= nx; ++i, k += 2)
1246:         {
1247:             xc[k] = xl + i * hx;
1248:             xc[k + 1] = y0;
1249:         }
1250:     }
1251:
1252:     return;
1253: }
1254:
1255: void Mesh_2d_3_square::GetConnectivityInRectangle(int const nx, int const ny, int ia
[])
1256: {
1257:     const int dx = nx + 1;
1258:     int k = 0;
1259:     int l = 0;
1260:     for (int j = 0; j < ny; ++j, ++k)
1261:     {
1262:         for (int i = 0; i < nx; ++i, ++k)
1263:         {
1264:             ia[l] = k;
1265:             ia[l + 1] = k + 1;
1266:             ia[l + 2] = k + dx + 1;
1267:             l += 3;
1268:             ia[l] = k;
1269:             ia[l + 1] = k + dx;
1270:             ia[l + 2] = k + dx + 1;
1271:             l += 3;
1272:         }
1273:     }
1274:     return;
1275: }
1276:
1277: // #####
1278:
```

```

1: #ifndef GEOM_FILE
2: #define GEOM_FILE
3: #include <array>
4: #include <functional>           // function; C++11
5: #include <iostream>
6: #include <memory>              // shared_ptr
7: #include <string>
8: #include <vector>
9:
10: /**
11:  * Basis class for finite element meshes.
12:  */
13: class Mesh
14: {
15: public:
16:     /**
17:      * Constructor initializing the members with default values.
18:      *
19:      * @param[in] ndim   space dimensions (dimension for coordinates)
20:      * @param[in] nvert_e number of vertices per element (dimension for connectivity)
21:      * @param[in] ndof_e degrees of freedom per element (= @p nvert_e for linear elements)
22:      * @param[in] nedge_e number of edges per element (= @p nvert_e for linear elements in 2D)
23:      */
24:     explicit Mesh(int ndim, int nvert_e = 0, int ndof_e = 0, int nedge_e = 0);
25:
26:     __attribute__((noinline))
27:     Mesh(Mesh const &) = default;
28:
29:     Mesh &operator=(Mesh const &) = delete;
30:
31:     /**
32:      * Destructor.
33:      *
34:      * See clang warning on
35:      * <a href="https://stackoverflow.com/questions/28786473/clang-no-out-of-line-virtual-method-definitions-pure-abstract-c-class/40550578">weak-vtables</a>.
36:      */
37:     virtual ~Mesh();
38:
39:     /**
40:      * Reads mesh data from a binary file.
41:      *
42:      * File format, see ascii_write_mesh.m
43:      *
44:      * @param[in] fname file name
45:      */
46:     explicit Mesh(std::string const &fname);
47:
48:     /**
49:      * Reads mesh data from a binary file.
50:      *
51:      * File format, see ascii_write_mesh.m
52:      *
53:      * @param[in] fname file name
54:      */
55:     void ReadVertexBasedMesh(std::string const &fname);
56:
57:     /**
58:      * Number of finite elements in (sub)domain.
59:      * @return number of elements.
60:      */
61:     int Nelems() const
62:     {
63:         return _nelem;
64:     }

```

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

```

132:     }
133:
134:     /**
135:      * (Re-)Allocates memory for the element connectivity and redefines the appropriate dimensions.
136:      *
137:      * @param[in] nnodes    number of nodes
138:      * @param[in] ndim      space dimension
139:      */
140:     void Resize_Coords(int nnodes, int ndim)
141:     {
142:         SetNnode(nnodes);           // number of nodes
143:         SetNdim(ndim);              // space dimension
144:         _xc.resize(nnodes * ndim);
145:     }
146:
147:     /**
148:      * Read coordinates of vertices (x,y)_i.
149:      * @return coordinates vector [nnodes*2].
150:      */
151:     const std::vector<double> &GetCoords() const
152:     {
153:         return _xc;
154:     }
155:
156:     /**
157:      * Access/Change coordinates of vertices (x,y)_i.
158:      * @return coordinates vector [nnodes*2].
159:      */
160:     std::vector<double> &GetCoords()
161:     {
162:         return _xc;
163:     }
164:
165:     /**
166:      * Calculate values in vector @p v via function @p func(x,y)
167:      * @param[in] v        vector
168:      * @param[in] func      function of (x,y) returning a double value.
169:      */
170:     void SetValues(std::vector<double> &v, const std::function<double(double, double)> &func) const;
171:     void SetBoundaryValues(std::vector<double> &v, const std::function<double(double, double)> &func) const;
172:     void SetDirichletValues(std::vector<double> &v, const std::function<double(double, double)> &func) const;
173:
174:     /**
175:      * Prints the information for a finite element mesh
176:      */
177:     void Debug() const;
178:
179:     /**
180:      * Prints the edge based information for a finite element mesh
181:      */
182:     void DebugEdgeBased() const;
183:
184:     /**
185:      * Determines the indices of those vertices with Dirichlet boundary conditions
186:      * @return index vector.
187:      */
188:     virtual std::vector<int> Index_DirichletNodes() const;
189:     virtual std::vector<int> Index_BoundaryNodes() const;
190:
191:     /**
192:      * Write vector @p v together with its mesh information to an ASCII file @p fnam
193:      *
194:      * The data are written in C-style.

```

```
195:      *
196:      * @param[in] fname    file name
197:      * @param[in] v        vector
198:      */
199: void Write_ascii_matlab(std::string const &fname, std::vector<double> const &v)
const;
200:
201: /**
202:  * Exports the mesh information to ASCII files @p basename + {_coords|_elements
203:  }.txt.
204:  *
205:  * The data are written in C-style.
206:  *
207:  * @param[in] basename    first part of file names
208:  */
209: void Export_scicomp(std::string const &basename) const;
210:
211: /**
212:  * Visualize @p v together with its mesh information via matlab or octave.
213:  *
214:  * Comment/uncomment those code lines in method Mesh::Visualize (geom.cpp)
215:  * that are supported on your system.
216:  *
217:  * @param[in] v            vector
218:  *
219:  * @warning matlab files ascii_read_meshvector.m visualize_results.m
220:  * must be in the executing directory.
221:  */
222: void Visualize(std::vector<double> const &v) const;
223:
224: /**
225:  * Global number of edges.
226:  * @return number of edges in mesh.
227:  */
228: int Nedges() const
229: {
230:     return _nedge;
231: }
232:
233: /**
234:  * Global number of edges for each finite element.
235:  * @return number of edges per element.
236:  */
237: int NedgesElements() const
238: {
239:     return _nedge_e;
240: }
241:
242: /**
243:  * Read edge connectivity information (e1,e2,e3)_i.
244:  * @return edge connectivity vector [nelems*_nedge_e].
245:  */
246: const std::vector<int> &GetEdgeConnectivity() const
247: {
248:     return _ea;
249: }
250:
251: /**
252:  * Access/Change edge connectivity information (e1,e2,e3)_i.
253:  * @return edge connectivity vector [nelems*_nedge_e].
254:  */
255: std::vector<int> &GetEdgeConnectivity()
256: {
257:     return _ea;
258: }
259:
260: /**
261:  * Read edge information (v1,v2)_i.
```

```

261:      * @return edge connectivity vector [_nedge*2].
262:      */
263:      const std::vector<int> &GetEdges() const
264:      {
265:          return _edges;
266:      }
267:
268:      /**
269:       * Access/Change edge information (v1,v2)_i.
270:       * @return edge connectivity vector [_nedge*2].
271:       */
272:      std::vector<int> &GetEdges()
273:      {
274:          return _edges;
275:      }
276:
277:      /**
278:       * Determines all node to node connections from the vertex based mesh.
279:       *
280:       * @return vector[k][] containing all connections of vertex k, including to itse
281:       *
282:       * if.
283:       *
284:       *
285:       *
286:       *
287:       *
288:       *
289:       *
290:       *
291:       *
292:       *
293:       *
294:       *
295:       *
296:       *
297:       *
298:       *
299:       *
300:       *
301:       *
302:       *
303:       *
304:       *
305:       *
306:       *
307:       *
308:       *
309:       *
310:       *
311:       *
312:       *
313:       *
314:       *
315:       *
316:       *
317:       *
318:       *
319:       *
320:       *
321:       *
322:       *
323:       *
324:       *
325:       *
326:       *
327:       *

```

```

328:     void SetNdofsElement(int ndof)
329:     {
330:         _ndof_e = ndof;
331:     }
332:
333:     void SetNnode(int nnode)
334:     {
335:         _nnode = nnode;
336:     }
337:
338:     void SetNdim(int ndim)
339:     {
340:         _ndim = ndim;
341:     }
342:
343:     void SetNedge(int nedge)
344:     {
345:         _nedge = nedge;
346:     }
347:
348:     /**
349:      * Reads vertex based mesh data from a binary file.
350:      *
351:      * File format, see ascii_write_mesh.m
352:      *
353:      * @param[in] fname file name
354:      */
355:     void ReadVertexBasedMesh(std::string const &fname);
356:
357:     /**
358:      * The vertex based mesh data are used to derive the edge based data.
359:      *
360:      * @warning Exactly 3 vertices, 3 edges per element are assumed (linear triangl
e in 2D)
361:      */
362:     void DeriveEdgeFromVertexBased()
363:     {
364:         //DeriveEdgeFromVertexBased_slow();
365:         //DeriveEdgeFromVertexBased_fast();
366:         DeriveEdgeFromVertexBased_fast_2();
367:     }
368:     void DeriveEdgeFromVertexBased_slow();
369:     void DeriveEdgeFromVertexBased_fast();
370:     void DeriveEdgeFromVertexBased_fast_2();
371:
372:
373:
374:     /**
375:      * The edge based mesh data are used to derive the vertex based data.
376:      *
377:      * @warning Exactly 3 vertices, 3 edges per element are assumed (linear triangl
e in 2D)
378:      */
379:     void DeriveVertexFromEdgeBased();
380:
381:     /**
382:      * Determines the indices of those vertices with Dirichlet boundary conditions
383:      * @return index vector.
384:      */
385:     int Nnbedges() const
386:     {
387:         return static_cast<int>(_bedges.size());
388:     }
389:
390:     /**
391:      * Checks whether the array dimensions fit to their appropriate size parameters
392:      * @return index vector.
393:      */

```

```

394:     virtual bool Check_array_dimensions() const;
395:
396:     /**
397:      * Permutes the vertex information in an edge based mesh.
398:      *
399:      * @param[in] old2new    new indices of original vertices.
400:      */
401:     void PermuteVertices_EdgeBased(std::vector<int> const &old2new);
402:
403: private:
404:     /**
405:      * Determines all node to node connections from the vertex based mesh.
406:      *
407:      * @return vector[k][] containing all connections of vertex k, including to itse
lf.
408:      */
409:     std::vector<std::vector<int>> Node2NodeGraph_1() const; // is correct
410:
411:     /**
412:      * Determines all node to node connections from the vertex based mesh.
413:      *
414:      * Faster than @p Node2NodeGraph_1().
415:      *
416:      * @return vector[k][] containing all connections of vertex k, including to itse
lf.
417:      */
418:     std::vector<std::vector<int>> Node2NodeGraph_2() const; // is correct
419:
420:     //private:
421: protected:
422:     int _nelem;           //!< number elements
423:     int _nvert_e;         //!< number of vertices per element
424:     int _ndof_e;          //!< degrees of freedom (d.o.f.) per element
425:     int _nnode;           //!< number nodes/vertices
426:     int _ndim;            //!< space dimension of the problem (1, 2, or 3)
427:     std::vector<int> _ia;  //!< element connectivity
428:     std::vector<double> _xc; //!< coordinates
429:
430: protected:
431:     // B.C.
432:     std::vector<int> _bedges;      //!< boundary edges [nbedges][2] storing start/end
vertex
433: // 2020-01-08
434:     std::vector<int> _sdedges;     //!< boundary edges [nbedges][2] with left/right s
ubdomain number
435:
436:     //private:
437: protected:
438:     // edge based connectivity
439:     int _nedge;              //!< number of edges in mesh
440:     int _nedge_e;            //!< number of edges per element
441:     std::vector<int> _edges;  //!< edges of mesh (vertices ordered ascending)
442:     std::vector<int> _ea;     //!< edge based element connectivity
443:     // B.C.
444:     std::vector<int> _ebedges; //!< boundary edges [nbedges]
445:
446: private:
447:     const std::vector<int> _dummy; //!< empty dummy vector
448:
449: };
450:
451:
452: // *****
453:
454: class RefinedMesh: public Mesh
455: {
456: public:
457:     /**

```

```

458:      * Constructs a refined mesh according to the marked elements in @p ibref.
459:      *
460:      * If the vector @p ibref has size 0 then all elements will be refined.
461:      *
462:      * @param[in] cmesh original mesh for coarsening.
463:      * @param[in] ibref vector containing True/False regarding refinement for each
element
464:      *
465:      */
466:      //explicit RefinedMesh(Mesh const &cmesh, std::vector<bool> const &ibref = std::
vector<bool>(0));
467:      RefinedMesh(Mesh const &cmesh, std::vector<bool> const &ibref);
468:      //RefinedMesh(Mesh const &cmesh, std::vector<bool> const &ibref);
469:
470:      /**
471:       * Constructs a refined mesh by regular refinement of all elements.
472:       *
473:       * @param[in] cmesh original mesh for coarsening.
474:       *
475:       */
476:      explicit RefinedMesh(Mesh const &cmesh)
477:          : RefinedMesh(cmesh, std::vector<bool>(0))
478:      {}
479:
480:
481:      RefinedMesh(RefinedMesh const &) = delete;
482:      //RefinedMesh(RefinedMesh const&&) = delete;
483:
484:      RefinedMesh &operator=(RefinedMesh const &) = delete;
485:      //RefinedMesh& operator=(RefinedMesh const&&) = delete;
486:
487:      /**
488:       * Destructor.
489:       */
490:      virtual ~RefinedMesh() override;
491:
492:      /**
493:       * Refines the mesh according to the marked elements.
494:       *
495:       * @param[in] ibref vector containing True/False regarding refinement for each
element
496:       *
497:       * @return the refined mesh
498:       *
499:       */
500:      Mesh RefineElements(std::vector<bool> const &ibref);
501:
502:      /**
503:       * Refines all elements in the actual mesh.
504:       *
505:       * @param[in] nref number of regular refinements to perform
506:       *
507:       */
508:      void RefineAllElements(int nref = 1);
509:
510:      /**
511:       * Accesses the father-of-nodes relation.
512:       *
513:       * @return father-of-nodes relation [nnodes][2]
514:       *
515:       */
516:      std::vector<int> const &GetFathersOfVertices() const override
517:      {
518:          return _vfathers;
519:      }
520:
521: protected:
522:      /**

```

```

523:      * Checks whether the array dimensions fit to their appropriate size parameters
524:      * @return index vector.
525:      */
526:  bool Check_array_dimensions() const override;
527:
528:  /**
529:   * Permutes the vertex information in an edge based mesh.
530:   *
531:   * @param[in] old2new    new indices of original vertices.
532:   */
533:  void PermuteVertices_EdgeBased(std::vector<int> const &old2new);
534:
535:
536: private:
537:     //Mesh const          &_cmesh; //!< coarse mesh
538:     std::vector<bool> const _ibref; //!< refinement info
539:     int _nref;           //!< number of regular refinements performed
540:     std::vector<int> _vfathers; //!< stores the 2 fathers of each vertex (e
qual fathers denote original coarse vertex)
541:
542: };
543:
544: // *****
545:
546: class gMesh_Hierarchy
547: {
548: public:
549:     /**
550:      * Constructs mesh hierarchy of @p nlevel levels starting with coarse mesh @p cmesh.
551:      * The coarse mesh @p cmesh will be @p nlevel-1 times geometrically refined.
552:      *
553:      * @param[in] cmesh    initial coarse mesh
554:      * @param[in] nlevel   number levels in mesh hierarchy
555:      */
556:     gMesh_Hierarchy(Mesh const &cmesh, int nlevel);
557:
558:
559:     size_t size() const
560:     {
561:         return _gmesh.size();
562:     }
563:
564:     /**
565:      * Access to mesh @p lev from mesh hierarchy.
566:      *
567:      * @return mesh @p lev
568:      * @warning An out_of_range exception might be thrown.
569:      */
570:     Mesh const &operator[](int lev) const
571:     {
572:         return *_gmesh.at(lev);
573:     }
574:
575:
576:     /**
577:      * Access to finest mesh in mesh hierarchy.
578:      *
579:      * @return finest mesh
580:      */
581:     Mesh const &finest() const
582:     {
583:         return *_gmesh.back();
584:     }
585:
586:
587:     /**
588:      * Access to coarsest mesh in mesh hierarchy.

```

```

589:      *
590:      * @return coarsest mesh
591:      *
592:      */
593:      Mesh const &coarsest() const
594:      {
595:          return *_gmesh.front();
596:      }
597:
598: private:
599:      std::vector<std::shared_ptr<Mesh>> _gmesh; ///< mesh hierarchy from coarse ([0])
to fine.
600:
601: };
602:
603:
604:
605: // *****
606: /**
607:  * 2D finite element mesh of the square consisting of linear triangular elements.
608:  */
609: class Mesh_2d_3_square: public Mesh
610: {
611: public:
612:     /**
613:      * Generates the f.e. mesh for the unit square.
614:      *
615:      * @param[in] nx    number of discretization intervals in x-direction
616:      * @param[in] ny    number of discretization intervals in y-direction
617:      * @param[in] myid  my MPI-rank / subdomain
618:      * @param[in] procx number of ranks/subdomains in x-direction
619:      * @param[in] procy number of processes in y-direction
620:      */
621:     Mesh_2d_3_square(int nx, int ny, int myid = 0, int procx = 1, int procy = 1);
622:
623:     /**
624:      * Destructor
625:      */
626:     ~Mesh_2d_3_square() override;
627:
628:     /**
629:      * Set solution vector based on a tensor product grid in the rectangle.
630:      * @param[in] u solution vector
631:      */
632:     void SetU(std::vector<double> &u) const;
633:
634:     /**
635:      * Set right hand side (rhs) vector on a tensor product grid in the rectangle.
636:      * @param[in] f rhs vector
637:      */
638:     void SetF(std::vector<double> &f) const;
639:
640:     /**
641:      * Determines the indices of those vertices with Dirichlet boundary conditions
642:      * @return index vector.
643:      */
644:     std::vector<int> Index_DirichletNodes() const override;
645:     std::vector<int> Index_BoundaryNodes() const override;
646:
647:     /**
648:      * Stores the values of vector @p u of (sub)domain into a file @p name for further
her processing in gnuplot.
649:      * The file stores rowwise the x- and y- coordinates together with the value from
@p u .
650:      * The domain [@p xl, @p xr] x [@p yb, @p yt] is discretized into @p nx x @p ny
intervals.
651:      *
652:      * @param[in] name  basename of file name (file name will be extended by the ra

```

```

nk number)
653:      * @param[in] u      local vector
654:      *
655:      * @warning    Assumes tensor product grid in unit square; rowise numbered
656:      *              (as generated in class constructor).
657:      *              The output is provided for tensor product grid visualization
658:      *              ( similar to Matlab-surf() ).
659:      *
660:      * @see Mesh_2d_3_square
661:      */
662:      void SaveVectorP(std::string const &name, std::vector<double> const &u) const;
663:
664:      // here will still need to implement in the class
665:      // GetBound(), AddBound()
666:      // or better a generalized way with indices and their appropriate ranks for MPI
communication
667:
668: private:
669:     /**
670:      * Determines the coordinates of the discretization nodes of the domain [@p xl,
671:      * @p xr] x [@p yb, @p yt]
672:      * which is discretized into @p nx x @p ny intervals.
673:      * @param[in] nx      number of discretization intervals in x-direction
674:      * @param[in] ny      number of discretization intervals in y-direction
675:      * @param[in] xl      x-coordinate of left boundary
676:      * @param[in] xr      x-coordinate of right boundary
677:      * @param[in] yb      y-coordinate of lower boundary
678:      * @param[in] yt      y-coordinate of upper boundary
679:      * @param[out] xc     coordinate vector of length 2n with x(2*k,2*k+1) as coordin
ates of node k
680:      */
681:      void GetCoordsInRectangle(int nx, int ny, double xl, double xr, double yb, doubl
e yt,
682:                               double xc[]);
683:     /**
684:      * Determines the element connectivity of linear triangular elements of a FEM d
iscretization
685:      * of a rectangle using @p nx x @p ny equidistant intervals for discretization.
686:      * @param[in] nx      number of discretization intervals in x-direction
687:      * @param[in] ny      number of discretization intervals in y-direction
688:      * @param[out] ia     element connectivity matrix with ia(3*s,3*s+1,3*s+2) as nod
e numbers od element s
689:      */
690:      void GetConnectivityInRectangle(int nx, int ny, int ia[]);
691:
692: private:
693:      int _myid;          //!< my MPI rank
694:      int _procx;         //!< number of MPI ranks in x-direction
695:      int _procy;         //!< number of MPI ranks in y-direction
696:      std::array<int, 4> _neigh; //!< MPI ranks of neighbors (negative: no neighbor bu
t b.c.)
697:      int _color;         //!< red/black coloring (checker board) of subdomains
698:
699:      double _xl;         //!< x coordinate of lower left corner of square
700:      double _xr;         //!< x coordinate of lower right corner of square
701:      double _yb;         //!< y coordinate of lower left corner of square
702:      double _yt;         //!< y coordinate of upper right corner of square
703:      int _nx;            //!< number of intervals in x-direction
704:      int _ny;            //!< number of intervals in y-direction
705: };
706:
707: // *****
708:
709:
710:
711:
712: #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 "par_geom.h"
7: #include "vdop.h"
8:
9: #include <cassert>
10: #include <cmath>
11: #include <iostream>
12: #include <mpi.h>          // MPI
13: // #include <omp.h>        // OpenMP, for E9 it is not necessary
14: using namespace std;
15:
16:
17: int main(int argc, char **argv )
18: {
19:     MPI_Init(&argc, &argv);
20:     MPI_Comm const icomm(MPI_COMM_WORLD);
21:     //omp_set_num_threads(1);          // don't use OMP parallelization for
a start
22: //
23:     {
24:         int np;
25:         MPI_Comm_size(icomm, &np);
26:
27:         assert(4 == np);          // example is only provided for 4 MPI proce
sses
28:     }
29: // #####
30: // ---- Read the f.e. mesh and the mapping of elements to MPI processes
31: // Mesh const mesh_c("square_4.txt");    // Files square_4.txt and square_4_sd
.txt are needed
32:     ParMesh const mesh("square", icomm);
33:
34:     int const numprocs = mesh.NumProcs();
35:     int const myrank = mesh.MyRank();
36:     if ( 0 == myrank ) {
37:         cout << "\n There are " << numprocs << " processes running.\n \n";
38:     }
39:
40:     int const check_rank=0;          // choose the MPI process you would like t
o check the mesh
41:     //if ( check_rank == myrank ) mesh.Debug();
42:     //if ( check_rank == myrank ) mesh.DebugEdgeBased();
43:
44: // ---- allocate local vectors and check skalar product and vector accumulation
45:     vector<double> xl(mesh.Nnodes(), 1.0);
46:     mesh.SetValues(xl, [] (double x, double y) -> double {return x * x * std::sin(2.5
* M_PI * y);} );
47:
48:     //E9
49:     if (myrank == check_rank){
50:         cout << "E9" << endl;
51:     }
52:
53:     double ss = mesh.dscapr(xl,xl);
54:     if (myrank == check_rank){
55:         cout << myrank << " : scalar : " << ss << endl << endl;
56:     }
57:
58:     mesh.VecAccu(xl);
59:
60:     //E10
61:     if (myrank == check_rank){
62:         cout << "E10" << endl;
63:     }

```

```
64:     vector<int> xl_int(mesh.Nnodes(), 1);
65:
66:     // check accumulation (by console output)
67:     mesh.VecAccu(xl_int);
68:
69:     vector<double> coords = mesh.GetCoords();
70:     if (check_rank == myrank){
71:         for (size_t i = 0; i < coords.size(); i += 2){
72:             cout << "(" << coords[i] << ", " << coords[i + 1] << "):\t" << xl_int[i/
2] << endl;
73:         }
74:     }
75:
76:     //E11
77:     if (myrank == check_rank){
78:         cout << "E11" << endl;
79:     }
80:     int global_nodes = mesh.GlobalNodes();
81:     if (check_rank == myrank){
82:         cout << "Global nodes: " << global_nodes << endl;
83:     }
84:
85:     //E12
86:     if (myrank == check_rank){
87:         cout << "E12" << endl;
88:     }
89:     vector<double> xl_new(mesh.Nnodes(), 1.0);
90:
91:     mesh.Average(xl_new);
92:
93:
94:
95:
96:
97:
98:
99:
100:
101:
102:
103:
104:     MPI_Finalize();
105:     return 0;
106: }
107:
108:
```

```

1: // see: http://llvm.org/docs/CodingStandards.html#include-style
2: #include "vdop.h"
3: // #include "geom.h"
4: #include "par_geom.h"
5:
6: #include <algorithm>
7: #include <array>
8: #include <cassert>
9: #include <cmath>
10: #include <ctime> // contains clock()
11: #include <fstream>
12: #include <iostream>
13: #include <list>
14: #include <numeric> // accumulate()
15: #include <string>
16: #include <vector>
17:
18: using namespace std;
19:
20:
21: ParMesh::ParMesh(int ndim, int nvert_e, int ndof_e, int nedge_e, MPI_Comm const &icomm)
mm)
22: : Mesh(ndim, nvert_e, ndof_e, nedge_e),
23:   _icomm(icomm), _numprocs(-1), _myrank(-1),
24:   _v_l2g(0), _t_l2g(0), _v_g2l{{{}}}, _t_g2l{{{}}}, _valence(0),
25:   _sendbuf(0), _sendcounts(0), _sdispls(0),
26:   _loc_itf(0), _gloc_itf(0), _buf2loc(0)
27: {
28:   MPI_Comm_size(icomm, &_numprocs);
29:   MPI_Comm_rank(icomm, &_myrank);
30: }
31:
32: ParMesh::~ParMesh()
33: {}
34:
35:
36:
37: ParMesh::ParMesh(std::string const &sname, MPI_Comm const &icomm)
38: : ParMesh(2, 3, 3, 3, icomm) // two dimensions, 3 vertices, 3 dofs, 3 edges per
element
39: {
40:   //const int numprocs = _icomm.GetSize();
41:   const string NS = "_" + to_string(_numprocs);
42:   const string fname = sname + NS + ".txt";
43:   //cout << "##### " << fname << endl;
44:   ReadVertexBasedMesh(fname);
45:   cout << "\n End of sequential File read \n";
46:   // -----
-
47:   // Until this point a l l processes possess a l l mesh info in g l o b a l
numbering
48:   //
49:   // Now, we have to select the data belonging to my_rank
50:   // and we have to create the mapping local to global (l2g) and vice versa (g2l)
51:   // -----
-
52:
53:   // save the global node mesh (maybe we need it later)
54:   DeriveEdgeFromVertexBased(); // and even
more
55:   Mesh global_mesh(*this); // requires a l o t of memory
56:   Del_EdgeConnectivity();
57:
58:   // read the subdomain info
59:   const string dname = sname + NS + "_sd" + ".txt";
60:   vector<int> t2d = ReadElementSubdomains(dname); // global mapping triangle to s
ubdomain for all elements
61:

```

```

62:     //const int myrank = _icomm.Get_rank();
63:     Transform_Local2Global_Vertex(_myrank, t2d);           // Vertex based mesh: now in
1 o c a l indexing
64:
65:     DeriveEdgeFromVertexBased();                           // Generate also the 1 o c a l
edge based information
66:
67:     Generate_VectorAdd();
68:
69:
70:     // Now we have to organize the MPI communication of vertices on the subdomain in
terfaces
71:
72:     return;
73: }
74:
75: vector<int> ParMesh::ReadElementSubdomains(string const &dtype)
76: {
77:     ifstream ifs(dtype);
78:     if (!(ifs.is_open() && ifs.good())) {
79:         cerr << "ParMesh::ReadElementSubdomain: Error cannot open file " << dtype <<
endl;
80:         assert(ifs.is_open());
81:     }
82:
83:     int const OFFSET{1};                                     // Matlab to C indexing
84:     cout << "ASCII file " << dtype << " opened" << endl;
85:
86:     // Read some mesh constants
87:     int nele;
88:     ifs >> nele;
89:     cout << nele << " " << Nelems() << endl;
90:     assert( Nelems() == nele);
91:
92:     // Allocate memory
93:     vector<int> t2d(nele, -1);
94:     // Read element mapping
95:     for (int k = 0; k < nele; ++k) {
96:         int tmp;
97:         ifs >> tmp;
98:         //t2d[k] = tmp - OFFSET;
99:         // 2020-01-08
100:         t2d[k] = min(tmp, NumProcs()) - OFFSET;
101:     }
102:
103:     return t2d;
104: }
105:
106: void ParMesh::Transform_Local2Global_Vertex(int const myrank, vector<int> const &t2d
)
107: {
108:     // number of local elements
109:     const int l_ne = count(t2d.cbegin(), t2d.cend(), myrank);
110:     //cout << myrank << " :: " << l_ne << endl;
111:     vector<int> l_ia(l_ne * NverticesElements(), -1); // local elements still with g
lobal vertex numbers
112:     _t_l2g.resize(l_ne, -1);
113:
114:     int lk = 0;
115:     for (size_t k = 0; k < t2d.size(); ++k) {
116:         if (myrank == t2d[k]) {
117:             //if (0==myrank)
118:             //{
119:             //cout << lk << "   k   " << t2d[k] << endl;
120:             //}
121:             l_ia[3 * lk] = _ia[3 * k];
122:             l_ia[3 * lk + 1] = _ia[3 * k + 1];
123:             l_ia[3 * lk + 2] = _ia[3 * k + 2]; // local elements still with global v

```

ertex numbers

```

124:         _t_l2g[lk] = k;                                // elements: local to global mappi
ng
125:         _t_g2l[k] = lk;                                //                global to local
126:         ++lk;
127:     }
128: }
129: // Checks:
130: assert( count(l_ia.cbegin(), l_ia.cend(), -1) == 0 );
131: assert( count(_t_l2g.cbegin(), _t_l2g.cend(), -1) == 0 );
132:
133: // Vertices: local to global mapping
134: auto tmp = l_ia;
135: sort(tmp.begin(), tmp.end());
136: auto ip = unique(tmp.begin(), tmp.end());
137: tmp.erase(ip, tmp.end());
138: _v_l2g = tmp;                                // Vertices: local to global mappi
ng
139: for (size_t lkv = 0; lkv < _v_l2g.size(); ++lkv) {
140:     _v_g2l[_v_l2g[lkv]] = lkv;                //                global to local
141: }
142:
143: // Boundary edges
144: vector<int> l_bedges;
145: vector<int> l_sdedges;
146: for (size_t b = 0; b < _bedges.size(); b += 2) {
147:     int const v1 = _bedges[b];                // global vertex numbers
148:     int const v2 = _bedges[b + 1];
149:     try {
150:         int const lv1 = _v_g2l.at(v1);        // map[] would add that element
151:         int const lv2 = _v_g2l.at(v2);        //                but at() throws an except
ion
152:         l_bedges.push_back(lv1);
153:         l_bedges.push_back(lv2);                // Boundaries: already in local i
ndexing
154:         // 2020-01-08
155:         l_sdedges.push_back(_sdedges[b]);
156:         l_sdedges.push_back(_sdedges[b+1]);
157:     }
158:     catch (std::out_of_range & err) {
159:         //cerr << ".";
160:     }
161: }
162:
163: // number of local vertices
164: const int l_nn = _v_l2g.size();
165: vector<double> l_xc(Ndims()*l_nn);
166: for (int lkk = 0; lkk < l_nn; ++lkk) {
167:     int k = _v_l2g.at(lkk);
168:     l_xc[2 * lkk] = _xc[2 * k];
169:     l_xc[2 * lkk + 1] = _xc[2 * k + 1];
170: }
171:
172:
173: // Now, we represent the vertex mesh in l o c a l numbering
174: // elements
175:
176: for (size_t i = 0; i < l_ia.size(); ++i) {
177:     l_ia[i] = _v_g2l.at(l_ia[i]);                // element vertices: global to lo
cal
178: }
179: SetNelem(l_ne);
180: _ia = l_ia;
181: // boundary
182: _bedges = l_bedges;
183: _sdedges = l_sdedges;
184: // coordinates
185: SetNnode(l_nn);

```

```

186:     _xc = l_xc;
187:
188:     return;
189: }
190:
191:
192: void ParMesh::Generate_VectorAdd()
193: {
194:     // Some checks
195:     int lnn = Nnodes(); // local number of vertices
196:     assert(static_cast<int>(_v_l2g.size()) == lnn);
197:     int ierr{-12345};
198:
199:     // ---- Determine global largest vertex index
200:     int gidx_max{-1}; // global largest vertex index
201:     int lmax = *max_element(_v_l2g.cbegin(), _v_l2g.cend());
202:     MPI_Allreduce(&lmax, &gidx_max, 1, MPI_INT, MPI_MAX, _icomm);
203:     int gidx_min{-1}; // global smallest vertex index
204:     int lmin = *min_element(_v_l2g.cbegin(), _v_l2g.cend());
205:     MPI_Allreduce(&lmin, &gidx_min, 1, MPI_INT, MPI_MIN, _icomm);
206:     //cout << gidx_min << " " << gidx_max << endl;
207:     assert(0 == gidx_min); // global indices have to start wi
th 0
208:
209:
210:     // ---- Determine for all global vertices the number of subdomains it belongs to
211:     vector<int> global(gidx_max+1, 0); // global scalar array for verti
ces
212:     for (auto const gidx : _v_l2g) global[gidx] = 1;
213:     // https://www.mpi-forum.org/docs/mpi-2.2/mpi22-report/node109.htm
214:     ierr = MPI_Allreduce(MPI_IN_PLACE, global.data(), global.size(), MPI_INT, MPI_SU
M, _icomm);
215:     //if (0 == MyRank()) cout << global << endl;
216:     //MPI_Barrier(_icomm);
217:     //cout << _xc[2*_v_g2l.at(2)] << " , " << _xc[2*_v_g2l.at(2)+1] << endl;
218:     //MPI_Barrier(_icomm);
219:
220:     // now, global[] contains the number of subdomains a global vertex belongs to
221:     if (count(global.cbegin(), global.cend(), 0) > 0 )
222:         cerr << "\n !!! Non-continuous global vertex indexing !!!\n";
223:
224:     // ---- Determine local interface vertices ( <==> global[] > 1 )
225:     // _loc_itf, neigh_itf
226:     //vector<int> loc_itf; // local indices of interface ve
rtices on this MPI process
227:     for (size_t lk = 0; lk < _v_l2g.size(); ++lk) {
228:         int const gk = _v_l2g[lk]; // global index of local vertex lk
229:         if ( global[gk] > 1 ) {
230:             _loc_itf.push_back(lk); // local indices of interface vert
ices on this MPI process
231:         }
232:     }
233:
234:     //MPI_Barrier(_icomm);
235:     //if (0 == MyRank()) cout << "\n..._loc_itf...\n" << _loc_itf << "\n.....\n";
236:     //MPI_Barrier(_icomm);
237:     // ---- global indices of local interface vertices
238:     //auto gloc_itf(_loc_itf);
239:     _gloc_itf=_loc_itf;
240:     for_each(_gloc_itf.begin(), _gloc_itf.end(), [this] (auto & v) -> void { v = _v_
l2g[v]; } );
241:     //MPI_Barrier(_icomm);
242:     //if (0 == MyRank()) cout << "\n..._gloc_itf...\n" << _gloc_itf << "\n.....\n"
;
243:     //DebugVector(_gloc_itf, "_gloc_itf");
244:
245:     // ---- Determine the global length of interfaces
246:     vector<int> vnn(NumProcs(), -1); // number of interface vertices pe

```

```

r MPI rank
247:     int l_itf(_loc_itf.size()); // # local interface vertices
248:     ierr = MPI_Allgather(&l_itf, 1, MPI_INT, vnn.data(), 1, MPI_INT, _icomm);
249:     assert(0 == ierr);
250:     //cout << vnn << endl;
251:
252:     // ---- Now we consider only the interface vertices
253:     int snn = accumulate(vnn.cbegin(), vnn.cend(), 0); // required length of array f
or global interface indices
254:     //cout << snn << " " << gnn << endl;
255:     vector<int> dispnn(NumProcs(), 0); // displacement of interface verti
ces per MPI rank
256:     partial_sum(vnn.cbegin(), vnn.cend() - 1, dispnn.begin() + 1);
257:     //cout << dispnn << endl;
258:
259:     // ---- Get the global indices for all global interfaces
260:     vector<int> g_itf(snn, -1); // collects all global indices of
the global interfaces
261:     // https://www.mpich.org/static/docs/v3.0.x/www3/MPI_Gatherv.html
262:     ierr = MPI_Gatherv( _gloc_itf.data(), _gloc_itf.size(), MPI_INT,
263:         g_itf.data(), vnn.data(), dispnn.data(), MPI_INT, 0, _icomm)
;
264:     assert(0 == ierr);
265:     // https://www.mpich.org/static/docs/v3.1/www3/MPI_Bcast.html
266:     ierr = MPI_Bcast(g_itf.data(), g_itf.size(), MPI_INT, 0, _icomm);
267:     assert(0 == ierr); // Now, each MPI rank has the all
global indices of the global interfaces
268:     //MPI_Barrier(_icomm);
269:     //if (MyRank() == 0) cout << "\n...g_itf...\n" << g_itf << "\n.....\n";
270:     //MPI_Barrier(_icomm);
271:
272:     // ----- Determine all MPI ranks a local interface vertex belongs to
273:     vector<vector<int>> neigh_itf(_loc_itf.size()); // subdomains a local interface v
ertex belongs to
274:     for (size_t lk = 0; lk < _loc_itf.size(); ++lk) {
275:         const int gvert = _gloc_itf[lk]; // global index of local interfac
e node lk
276:         for (int rank = 0; rank < NumProcs(); ++rank) {
277:             auto const startl = g_itf.cbegin() + dispnn[rank];
278:             auto const endl = startl + vnn[rank];
279:             if (find(startl, endl, gvert) != endl) {
280:                 neigh_itf[lk].push_back(rank);
281:             }
282:         }
283:     }
284:
285:     // ---- check the available info in _loc_itf[lk], _gloc_itf[lk], neigh_itf[lk]
286:     //MPI_Barrier(_icomm);
287:     ///if (MyRank() == 0) cout << "\n...neigh_itf ...\n" << neigh_itf << endl;
288:     //if (MyRank() == 0) {
289:         //for (size_t lk = 0; lk < _loc_itf.size(); ++lk) {
290:             //cout << lk << " : local idx " << _loc_itf[lk] << " , global idx " <<
_gloc_itf[lk];
291:             //cout << " with MPI ranks " << neigh_itf[lk] << endl;
292:             //}
293:         //}
294:     //MPI_Barrier(_icomm);
295:
296:     // ---- store the valence (e.g., the number of subdomains it belongs to) of all
local vertices
297:     _valence.resize(Nnodes(), 1);
298:     for (size_t lk = 0; lk < _loc_itf.size(); ++lk)
299:     {
300:         _valence[_loc_itf[lk]] = neigh_itf[lk].size();
301:     }
302:     //DebugVector(_valence, "_valence", _icomm);
303:
304:     // ---- We were going to use MPI_Alltoallv for data exchange on interfaces

```

```

305:      //      https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report/node109.htm#Node109
306:      //      https://www.open-mpi.org/doc/v4.0/man3/MPI\_Alltoallv.3.php
307:      //int MPI_Alltoallv(const void* sendbuf, const int sendcounts[], const int sdisps
ls[], MPI_Datatype sendtype, void* recvbuf, const int recvcounts[], const int rdispls[], MP
I_Datatype recvtype, MPI_Comm comm)
308:      //
309:      // MPI_Alltoallv needs:
310:      //      vector<double> sendbuf          (MPI_IN_PLACE: used also as recvbuf)
311:      //      vector<int> sendcounts          (the same as for recv)
312:      //      vector<int> sdispls             (the same as for recv)
313:      //
314:      // We need to map the interface vertices onto the sendbuffer:
315:      //      vector<int> loc_itf             local index of interface vertex lk
316:      //      vector<int> gloc_itf           global index of interface vertex lk
317:      //      vector<int> buf2loc             local indices of sendbuffer position
s (the same as for recv)
318:
319:      // ---- Determine sendcounts[] and sdispls[] from neigh_itf[]
320:      //vector<int> _sendcounts(NumProcs(), 0);
321:      _sendcounts.resize(NumProcs(), 0);
322:      for (size_t lk = 0; lk < _loc_itf.size(); ++lk) {
323:          auto const &kneigh = neigh_itf[lk];
324:          for (size_t ns = 0; ns < kneigh.size(); ++ns) {
325:              ++_sendcounts[kneigh[ns]];
326:          }
327:      }
328:      //if (MyRank() == 0) cout << "\n..._sendcounts ...\n" << _sendcounts << endl;
329:
330:      //vector<int> _sdispls(NumProcs(), 0);
331:      _sdispls.resize(NumProcs(), 0);
332:      partial_sum(_sendcounts.cbegin(), _sendcounts.cend() - 1, _sdispls.begin() + 1);
333:      //vector<int> _sdispls(NumProcs()+1, 0);
334:      //partial_sum(_sendcounts.cbegin(), _sendcounts.cend(), _sdispls.begin() + 1);
335:      //if (MyRank() == 0) cout << "\n..._sdispls ...\n" << _sdispls << endl;
336:
337:      // ---- Determine size of buffer 'nbuffer' and mapping 'buf2loc'
338:      int const nbuffer = accumulate(_sendcounts.cbegin(), _sendcounts.cend(), 0);
339:      //vector<int> _buf2loc(nbuffer, -1);
340:      _buf2loc.resize(nbuffer, -1);
341:      int buf_idx = 0; // position in buffer
342:      for (int rank = 0; rank < NumProcs(); ++rank) {
343:          assert(buf_idx == _sdispls[rank]);
344:          for (size_t lk = 0; lk < _loc_itf.size(); ++lk) {
345:              auto const &kneigh = neigh_itf[lk];
346:              if (find(kneigh.cbegin(), kneigh.cend(), rank) != kneigh.cend())
347:              {
348:                  _buf2loc[buf_idx] = _loc_itf[lk];
349:                  ++buf_idx;
350:              }
351:          }
352:      }
353:      //if (MyRank() == 0) cout << "\n...buf2loc ...\n" << buf2loc << endl;
354:      //DebugVector(buf2loc, "buf2loc", _icomm);
355:
356:      // ---- Allocate send/recv buffer
357:      //vector<double> _sendbuf(nbuffer, -1.0);
358:      _sendbuf.resize(nbuffer, -1.0);
359:
360:      assert(CheckInterfaceExchange_InPlace());
361:      cout << " Check of data exchange (InPlace) successful!\n";
362:      assert(CheckInterfaceExchange());
363:      cout << " Check of data exchange successful!\n";
364:      assert(CheckInterfaceAdd_InPlace());
365:      cout << " Check of data add successful!\n";
366:      assert(CheckInterfaceAdd());
367:      cout << " Check of data add (InPlace) successful!\n";
368:
369:      vector<double> x(Nnodes(), -1.0);

```

```

370:     VecAccu(x);
371:     cout << " VecAccu (InPlace) successful!\n";
372:
373:
374:     return;
375: }
376:
377: bool ParMesh::CheckInterfaceExchange_InPlace() const
378: {
379:     vector<double> x(Nnodes(), -1.0);
380:     copy(_v_l2g.cbegin(), _v_l2g.cend(), x.begin());           // init x with global verte
x indices
381:
382:     for(size_t ls = 0; ls<_sendbuf.size(); ++ls)
383:     {
384:         _sendbuf[ls] = x[_buf2loc.at(ls)];
385:     }
386:     int ierr = MPI_Alltoallv(MPI_IN_PLACE, _sendcounts.data(), _sdispls.data(), MPI_
DOUBLE,
387:                             _sendbuf.data(), _sendcounts.data(), _sdispls.data(), MPI_
DOUBLE, _icomm);
388:     assert(ierr==0);
389:     //DebugVector(_sendbuf, "_sendbuf", _icomm);
390:
391:     vector<double> y(x);
392:     for(size_t lk = 0; lk<_loc_itf.size(); ++lk) y[_loc_itf.at(lk)] = -1.0;    // onl
y for interface nodes
393:     for(size_t ls = 0; ls<_sendbuf.size(); ++ls)
394:     {
395:         y[_buf2loc.at(ls)] = _sendbuf[ls];
396:     }
397:
398:     double const eps=1e-10;
399:     bool bv = equal(x.cbegin(), x.cend(), y.cbegin(),
400:                    [eps](double a, double b) -> bool
401:                    { return std::abs(a-b)<eps*(1.0+0.5*(std::abs(a)+ std::abs
(b))); });
402:
403:     return bv;
404: }
405:
406: bool ParMesh::CheckInterfaceExchange() const
407: {
408:     vector<double> x(Nnodes(), -1.0);
409:     copy(_v_l2g.cbegin(), _v_l2g.cend(), x.begin());           // init x with global verte
x indices
410:
411:     for(size_t ls = 0; ls<_sendbuf.size(); ++ls)
412:     {
413:         _sendbuf[ls] = x[_buf2loc.at(ls)];
414:     }
415:     vector<double> recvbuf(_sendbuf.size());
416:     int ierr = MPI_Alltoallv(_sendbuf.data(), _sendcounts.data(), _sdispls.data(), M
PI_DOUBLE,
417:                             recvbuf.data(), _sendcounts.data(), _sdispls.data(), M
PI_DOUBLE, _icomm);
418:     //DebugVector(_sendbuf, "_sendbuf", _icomm);
419:     //DebugVector(recvbuf, "recvbuf", _icomm);
420:     assert(ierr==0);
421:
422:     vector<double> y(x);
423:     for(size_t lk = 0; lk<_loc_itf.size(); ++lk) y[_loc_itf.at(lk)] = -1.0;    // onl
y for interface nodes
424:     for(size_t ls = 0; ls<recvbuf.size(); ++ls)
425:     {
426:         y[_buf2loc.at(ls)] = recvbuf[ls];
427:     }
428:     //cout << "WRONG : " << count(y.cbegin(), y.cend(), -1.0) << endl;

```

```

429:
430:     double const eps=1e-10;
431:     bool bv = equal(x.cbegin(),x.cend(),y.cbegin(),
432:                    [eps](double a, double b) -> bool
433:                    { return std::abs(a-b)<eps*(1.0+0.5*(std::abs(a)+ std::abs
(b))))); }
434:         );
435:     return bv;
436: }
437:
438: bool ParMesh::CheckInterfaceAdd_InPlace() const
439: {
440:     vector<double> x(Nnodes(),-1.0);
441:     for (size_t i=0; i<x.size(); ++i)
442:     {
443:         x[i] = _xc[2*i]+_xc[2*i+1]; // init x with coordinate v
444:     }
445:
446:     for(size_t ls = 0; ls<_sendbuf.size(); ++ls)
447:     {
448:         _sendbuf[ls] = x[_buf2loc.at(ls)];
449:     }
450:     int ierr = MPI_Alltoallv(MPI_IN_PLACE, _sendcounts.data(), _sdispls.data(), MPI_
DOUBLE,
451:                             _sendbuf.data(), _sendcounts.data(), _sdispls.data(), MPI_
DOUBLE, _icomm);
452:     assert(ierr==0);
453:     //DebugVector(_sendbuf, "_sendbuf", _icomm);
454:
455:     vector<double> y(x);
456:     for(size_t lk = 0; lk<_loc_itf.size(); ++lk) y[_loc_itf.at(lk)] = 0.0; // only
for interface nodes
457:     for(size_t ls = 0; ls<_sendbuf.size(); ++ls)
458:     {
459:         y[_buf2loc.at(ls)] += _sendbuf[ls];
460:     }
461:     MPI_Barrier(_icomm);
462:     //DebugVector(x, "x", _icomm);
463:     //DebugVector(y, "y", _icomm);
464:     for (size_t i= 0; i<y.size(); ++i) y[i]/=_valence[i]; // divide by valence
465:
466:     double const eps=1e-10;
467:     bool bv = equal(x.cbegin(),x.cend(),y.cbegin(),
468:                    [eps](double a, double b) -> bool
469:                    { return std::abs(a-b)<eps*(1.0+0.5*(std::abs(a)+ std::abs
(b))))); }
470:         );
471:     return bv;
472: }
473:
474: bool ParMesh::CheckInterfaceAdd() const
475: {
476:     vector<double> x(Nnodes(),-1.0);
477:     for (size_t i=0; i<x.size(); ++i)
478:     {
479:         //x[i] = _xc[2*i]+_xc[2*i+1]; // init x with coordinate
values
480:         x[i] = _v_l2g[i];
481:     }
482:
483:     for(size_t ls = 0; ls<_sendbuf.size(); ++ls)
484:     {
485:         _sendbuf[ls] = x[_buf2loc.at(ls)];
486:     }
487:     vector<double> recvbuf(_sendbuf.size());
488:     int ierr = MPI_Alltoallv(_sendbuf.data(), _sendcounts.data(), _sdispls.data(), M
PI_DOUBLE,

```

```

489:                                     recvbuf.data(), _sendcounts.data(), _sdispls.data(), M
PI_DOUBLE, _icomm);
490:     //DebugVector(_sendbuf, "_sendbuf", _icomm);
491:     //DebugVector(recvbuf, "recvbuf", _icomm);
492:     assert(ierr==0);
493:
494:     vector<double> y(x);
495:     for(size_t lk = 0; lk<_loc_itf.size(); ++lk) y[_loc_itf.at(lk)] = 0.0;    // only
for interface nodes
496:     for(size_t ls = 0; ls<recvbuf.size(); ++ls)
497:     {
498:         //if (0==MyRank()) cout << ls << ": " << _buf2loc.at(ls) << " " << y[_buf2l
oc.at(ls)] << " (" << x[_buf2loc.at(ls)] << ") " << " " << recvbuf[ls] << " (" << _sendbuf[
ls] << ") " << endl;
499:         y[_buf2loc.at(ls)] += recvbuf[ls];
500:     }
501:     MPI_Barrier(_icomm);
502:     //DebugVector(x, "x", _icomm);
503:     //DebugVector(y, "y", _icomm);
504:     for (size_t i= 0; i<y.size(); ++i) y[i]/=_valence[i];    // divide by valence
505:
506:     double const eps=1e-10;
507:     bool bv = equal(x.cbegin(), x.cend(), y.cbegin(),
508:                     [eps](double a, double b) -> bool
509:                     { return std::abs(a-b)<eps*(1.0+0.5*(std::abs(a)+ std::abs
(b))))); }
510:                                     );
511:     return bv;
512: }
513:
514:
515: // -----
516:
517: void ParMesh::VecAccu(std::vector<double> &w) const
518: {
519:     for(size_t ls = 0; ls<_sendbuf.size(); ++ls)
520:     {
521:         _sendbuf[ls] = w[_buf2loc.at(ls)];
522:     }
523:     int ierr = MPI_Alltoallv(MPI_IN_PLACE, _sendcounts.data(), _sdispls.data(), MPI_
DOUBLE,
524:                             _sendbuf.data(), _sendcounts.data(), _sdispls.data(), MPI_
DOUBLE, _icomm);
525:     assert(ierr==0);
526:     //DebugVector(_sendbuf, "_sendbuf", _icomm);
527:
528:     for(size_t lk = 0; lk<_loc_itf.size(); ++lk) w[_loc_itf.at(lk)] = 0.0;    // only
for interface nodes
529:     for(size_t ls = 0; ls<_sendbuf.size(); ++ls)
530:     {
531:         w[_buf2loc.at(ls)] += _sendbuf[ls];
532:     }
533:
534:     return;
535: }
536:
537: void ParMesh::VecAccu(std::vector<int> &w) const
538: {
539:     for(size_t ls = 0; ls<_sendbuf.size(); ++ls)
540:     {
541:         _sendbuf[ls] = w[_buf2loc.at(ls)];
542:     }
543:     int ierr = MPI_Alltoallv(MPI_IN_PLACE, _sendcounts.data(), _sdispls.data(), MPI_
DOUBLE,
544:                             _sendbuf.data(), _sendcounts.data(), _sdispls.data(), MPI_
DOUBLE, _icomm);
545:     assert(ierr==0);
546:     //DebugVector(_sendbuf, "_sendbuf", _icomm);

```

OK

```
547:
548:     for(size_t lk = 0; lk<_loc_itf.size(); ++lk) w[_loc_itf.at(lk)] = 0.0;    // only
for interface nodes
549:     for(size_t ls = 0; ls<_sendbuf.size(); ++ls)
550:     {
551:         w[_buf2loc.at(ls)] += _sendbuf[ls];
552:     }
553:
554:     return;
555: }
556:
557: int ParMesh::GlobalNodes() const
558: {
559:     int local_max = *max_element(_v_l2g.begin(), _v_l2g.end());
560:     int global_max = 0;
561:
562:     MPI_Allreduce(&local_max, &global_max, 1, MPI_INT, MPI_MAX, _icomm);
563:
564:     std::vector<int> global_flag(global_max + 1, 0);
565:     for (int gidx : _v_l2g){
566:         global_flag[gidx] = 1;
567:     }
568:
569:     MPI_Allreduce(MPI_IN_PLACE, global_flag.data(), global_flag.size(), MPI_INT, MPI_SUM
, _icomm);
570:
571:     int global_nodes = 0;
572:     for (int v : global_flag){
573:         if (v > 0) ++global_nodes;
574:     }
575:
576:     return global_nodes;
577: }
578:
579:
580: void ParMesh::Average(std::vector<double> &w) const
581: {
582:     VecAccu(w);
583:
584:     for (size_t lk = 0; lk < _loc_itf.size(); ++lk)
585:     {
586:         int const local_idx = _loc_itf[lk];
587:         w[local_idx] /= _valence[local_idx];
588:     }
589: }
590:
```

2
6
✓

```

1: #ifndef PAR_GEOM_FILE
2: #define PAR_GEOM_FILE
3: #include "geom.h"
4: #include "vdop.h"
5: #include <array>
6: #include <functional>           // function; C++11
7: #include <iostream>
8: #include <map>
9: #include <memory>              // shared_ptr
10: #include <mpi.h>               // MPI
11: #include <string>
12: #include <vector>
13:
14: class ParMesh: public Mesh
15: {
16: public:
17:     /**
18:      * Constructor initializing the members with default values.
19:      *
20:      * @param[in] ndim      space dimensions (dimension for coordinates)
21:      * @param[in] nvert_e   number of vertices per element (dimension for connectivity)
22:      * @param[in] ndof_e    degrees of freedom per element (= @p nvert_e for linear elements)
23:      * @param[in] nedge_e   number of edges per element (= @p nvert_e for linear elements in 2D)
24:      * @param[in] icomm     MPI communicator
25:      */
26:     explicit ParMesh(int ndim, int nvert_e = 0, int ndof_e = 0, int nedge_e = 0, MPI_Comm const &icomm = MPI_COMM_WORLD);
27:
28:     ParMesh(ParMesh const &) = default;
29:
30:     ParMesh &operator=(ParMesh const &) = delete;
31:
32:     /**
33:      * Destructor.
34:      *
35:      * See clang warning on
36:      * <a href="https://stackoverflow.com/questions/28786473/clang-no-out-of-line-virtual-method-definitions-pure-abstract-c-class/40550578">weak-vtables</a>.
37:      */
38:     virtual ~ParMesh();
39:
40:     /**
41:      * Reads mesh data from a binary file.
42:      *
43:      * @param[in] sname suffix of file name
44:      * @param[in] icomm MPI communicator
45:      * @see ascii_write_mesh.m for the file format.
46:      */
47:     explicit ParMesh(std::string const &sname, MPI_Comm const &icomm = MPI_COMM_WORLD);
48:
49:     void VecAccu(std::vector<double> &w) const;
50:     void VecAccu(std::vector<int> &x) const;
51:
52:     /** Inner product
53:      * @param[in] x vector
54:      * @param[in] y vector
55:      * @return      resulting Euclidian inner product <x,y>
56:      */
57:     double dscapr(std::vector<double> const &x, std::vector<double> const &y) const
58:     {
59:         return par_scalar(x, y, _icomm);
60:     }
61:
62:

```

```

63:     int GlobalNodes() const;
64:     void Average(std::vector<double> &w) const;
65:
66: private:
67:     /**
68:      * Reads the global triangle to subdomain mapping.
69:      *
70:      * @param[in] dname file name
71:      *
72:      * @see ascii_write_subdomains.m for the file format
73:      */
74:     std::vector<int> ReadElementSubdomains(std::string const &dname);
75:
76:
77:     /**
78:      * Transform
79:      *
80:      * @param[in] myrank MPI rank of this process
81:      * @param[in] t2d      global mapping triangle to subdomain for all elements (
vertex based)
82:      */
83:     void Transform_Local2Global_Vertex(int myrank, std::vector<int> const &t2d);
84:
85:
86:     /**
87:      * Transform
88:      */
89:     void Generate_VectorAdd();
90:
91:     bool CheckInterfaceExchange_InPlace() const;
92:     bool CheckInterfaceExchange() const;
93:     bool CheckInterfaceAdd_InPlace() const;
94:     bool CheckInterfaceAdd() const;
95:
96:
97: public:
98:     /**      MPI rank of the calling process in communication group.
99:      *
100:     * @return      MPI rank of the calling process
101:     */
102:     int MyRank() const
103:     {
104:         return _myrank;
105:     }
106:
107:     /**      Number of MPI processes in communication group.
108:     *
109:     * @return      Number of MPI processes
110:     */
111:     int NumProcs() const
112:     {
113:         return _numprocs;
114:     }
115:
116:     /**      Returns recent
117:     * @return      MPI communicator
118:     */
119:     MPI_Comm GetCommunicator() const
120:     {
121:         return _icomm;
122:     }
123:
124: private:
125:     // Don't use &_icomm ==> Error
126:     MPI_Comm const _icomm;          //!< MPI communicator for the group o
f processes
127:     int _numprocs;                  //!< number of MPI processes
128:     int _myrank;                    //!< my MPI rank

```

```
129:     std::vector<int> _v_l2g;                ///  
ng vertices: local to global mapping  
130:     std::vector<int> _t_l2g;                ///  
ng triangles: local to global mapping  
131:     std::map<int, int> _v_g2l;              ///  
ng vertices: global to local mapping  
132:     std::map<int, int> _t_g2l;              ///  
ng triangles: global to local mapping  
133:  
134:     //std::vector<int> e_l2g;                ///  
ping edges: local to global mapping  
135:  
136:     std::vector<int> _valence;              ///  
number of subdomains they belong to valence of local vertices, i.e.  
137:     // MPI_Alltoallv needs:  
138:     mutable std::vector<double> _sendbuf;    ///  
ffer (MPI_IN_PLACE) send buffer and receiving buffer  
139:     std::vector<int> _sendcounts;            ///  
PI rank (the same as for recv) number of data to send to each MPI rank  
140:     std::vector<int> _sdispls;               ///  
PI rank wrt. _sendbuffer (the same as for recv) offset of data to send to each MPI rank  
141:     //  
142:     // We need to map the interface vertices onto the sendbuffer:  
143:     std::vector<int> _loc_itf;               ///  
lk local index of interface vertex  
144:     std::vector<int> _gloc_itf;             ///  
lk global index of interface vertex  
145:     std::vector<int> _buf2loc;              ///  
tions (the same as for recv) local indices of sendbuffer positions  
146:  
147:  
148: };  
149:  
150:  
151: #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:
15: #pragma omp parallel for
16:     for (size_t k = 0; k < n; ++k)
17:     {
18:         x[k] = y[k] / z[k];
19:     }
20:     return;
21: }
22:
23: //*****
24:
25: void vdaxpy(std::vector<double> & x, std::vector<double> const& y,
26:            double alpha, std::vector<double> const& z )
27: {
28:     assert( x.size()==y.size() && y.size()==z.size() );
29:     size_t n = x.size();
30:
31: #pragma omp parallel for
32:     for (size_t k = 0; k < n; ++k)
33:     {
34:         x[k] = y[k] + alpha * z[k];
35:     }
36:     return;
37: }
38: //*****
39:
40: double dscapr(std::vector<double> const& x, std::vector<double> const& y)
41: {
42:     assert( x.size()==y.size() );
43:     size_t n = x.size();
44:
45:     double s = 0.0;
46: //#pragma omp parallel for reduction(+:s)
47:     for (size_t k = 0; k < n; ++k)
48:     {
49:         s += x[k] * y[k];
50:     }
51:
52:     return s;
53: }
54:
55: //*****
56: //void DebugVector(vector<double> const &v)
57: //{
58:     //cout << "\nVector  (nnode = " << v.size() << ") \n";
59:     //for (size_t j = 0; j < v.size(); ++j)
60:     //{
61:         //cout.setf(ios::right, ios::adjustfield);
62:         //cout << v[j] << " ";
63:     //}
64:     //cout << endl;
65:
66:     //return;
67: //}
68: //*****

```

```

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
(b))))); }
80:     };
81:     if (!bv)
82:     {
83:         assert(static_cast<int>(x.size())==n);
84:         cout << "##### Error: " << "values" << endl;
85:     }
86:     return bn && bv;
87: }
88:
89: //*****
90: double par_scalar(vector<double> const &x, vector<double> const &y, MPI_Comm const &
icomm)
91: {
92:     const double s = dscapr(x,y);
93:     double sg;
94:     MPI_Allreduce(&s,&sg,1,MPI_DOUBLE,MPI_SUM,icomm);
95:
96:     return (sg);
97: }
98:
99: //*****
100: void ExchangeAll(vector<double> const &xin, vector<double> &yout, MPI_Comm const &ic
omm)
101: {
102:     int myrank, numprocs,ierr(-1);
103:     MPI_Comm_rank(icomm, &myrank); // my MPI-rank
104:     MPI_Comm_size(icomm, &numprocs);
105:     int const N=xin.size();
106:     int const sendcount = N/numprocs; // equal sized junks
107:     assert(sendcount*numprocs==N); // really all junk sized
?
108:     assert(xin.size()==yout.size());
109:
110:     auto sendbuf = xin.data();
111:     auto recvbuf = yout.data();
112:     ierr = MPI_Alltoall(sendbuf, sendcount, MPI_DOUBLE,
113:                         recvbuf, sendcount, MPI_DOUBLE, icomm);
114:     assert(0==ierr);
115:
116:     return;
117: }
118:
119: //*****
120: void ExchangeAllInPlace(vector<double> &xin, MPI_Comm const &icomm)
121: {
122:     int myrank, numprocs,ierr(-1);
123:     MPI_Comm_rank(icomm, &myrank); // my MPI-rank
124:     MPI_Comm_size(icomm, &numprocs);
125:     int const N=xin.size();
126:     int const sendcount = N/numprocs; // equal sized junks
127:     assert(sendcount*numprocs==N); // really all junk sized
?
128:
129:     auto sendbuf = xin.data();
130:     ierr = MPI_Alltoall(MPI_IN_PLACE, sendcount, MPI_DOUBLE,

```

./accu_template/vdop.cpp

Mon Jan 12 07:22:36 2026

3

```
131:
132:     assert(0==ierr);
133:
134:     return;
135: }
```

sendbuf, sendcount, MPI_DOUBLE, ict);

```

1: #ifndef VDOP_FILE
2: #define VDOP_FILE
3: #include <iostream>
4: #include <mpi.h> // MPI
5: #include <string>
6: #include <vector>
7:
8: /** @brief Element-wise vector divison  $x_k = y_k/z_k$ .
9:  *
10:  * @param[out] x target vector
11:  * @param[in] y source vector
12:  * @param[in] z source vector
13:  *
14:  */
15: void vddiv(std::vector<double> &x, std::vector<double> const &y,
16:           std::vector<double> const &z);
17:
18: /** @brief Element-wise daxpy operation  $x(k) = y(k) + \alpha*z(k)$ .
19:  *
20:  * @param[out] x target vector
21:  * @param[in] y source vector
22:  * @param[in] alpha scalar
23:  * @param[in] z source vector
24:  *
25:  */
26: void vdaxpy(std::vector<double> &x, std::vector<double> const &y,
27:            double alpha, std::vector<double> const &z );
28:
29:
30: /** @brief Calculates the Euclidean inner product of two vectors.
31:  *
32:  * @param[in] x vector
33:  * @param[in] y vector
34:  * @return Euclidean inner product  $\langle x, y \rangle$ 
35:  *
36:  */
37: double dscapr(std::vector<double> const &x, std::vector<double> const &y);
38:
39:
40: inline
41: double L2_scapr(std::vector<double> const &x, std::vector<double> const &y)
42: {
43:     return dscapr(x, y) / x.size();
44: }
45:
46:
47: /** Parallel inner product
48:  * @param[in] x vector
49:  * @param[in] y vector
50:  * @param[in] icomm MPI communicator
51:  * @return resulting Euclidian inner product  $\langle x, y \rangle$ 
52:  */
53: double par_scalar(std::vector<double> const &x, std::vector<double> const &y,
54:                  MPI_Comm const& icomm=MPI_COMM_WORLD);
55:
56:
57:
58: /* ReadId : Input and broadcast of an integer */
59: inline
60: int ReadIn(std::string const &ss = std::string(), MPI_Comm const &icomm = MPI_COMM_W
ORLD)
61: {
62:     MPI_Barrier(icomm);
63:     int myrank; // my rank number */
64:     MPI_Comm_rank(icomm, &myrank);
65:     int id;
66:
67:     if (myrank == 0) {

```

```

68:         std::cout << "\n\n " << ss << " : Which process do you want to debug ? \n"
;
69:         std::cin >> id;
70:     }
71:     MPI_Bcast(&id, 1, MPI_INT, 0, icomm);
72:
73:     return id;
74: }
75:
76: /**
77:  * Print entries of a vector to standard output.
78:  *
79:  * @param[in]  v      vector values
80:  * @param[in]  ss      string containing the vector name
81:  * @param[in]  icomm   communicator group for MPI
82:  *
83:  */
84: //void DebugVector(std::vector<double> const &v);
85: template <class T>
86: void DebugVector(std::vector<T> const &v, std::string const &ss = std::string(), MPI
_Comm const &icomm = MPI_COMM_WORLD)
87: {
88:     MPI_Barrier(icomm);
89:     int numprocs;                                /* # processes */
90:     MPI_Comm_size(icomm, &numprocs);
91:     int myrank;                                    /* my rank number */
92:     MPI_Comm_rank(icomm, &myrank);
93:
94:     int readid = ReadIn(ss);                        /* Read readid */
95:
96:     while ( (0 <= readid) && (readid < numprocs) ) {
97:         if (myrank == readid) {
98:             std::cout << "\n\n process " << readid;
99:             std::cout << "\n .... " << ss << " (nnode = " << v.size() << ") \n";
100:            for (size_t j = 0; j < v.size(); ++j) {
101:                std::cout.setf(std::ios::right, std::ios::adjustfield);
102:                std::cout << v[j] << " ";
103:            }
104:            std::cout << std::endl;
105:            fflush(stdout);
106:        }
107:
108:        readid = ReadIn(ss, icomm);                    /* Read readid */
109:    }
110:    MPI_Barrier(icomm);
111:    return;
112: }
113:
114: /** @brief Compares an STL vector with POD vector.
115:  *
116:  * The accuracy criteria  $|x_k - y_k| < \epsilon \cdot (1 + |x_k| + |y_k|)$ 
117:  * follows the book by
118:  * <a href="https://www.springer.com/la/book/9783319446592">Stoyan/Baran</a>, p.8.
119:  *
120:  * @param[in]  x      STL vector
121:  * @param[in]  n      length of POD vector
122:  * @param[in]  y      POD vector
123:  * @param[in]  eps    relative accuracy criteria (default := 0.0).
124:  * @return true iff pairwise vector elements are relatively close to each other.
125:  *
126:  */
127: bool CompareVectors(std::vector<double> const &x, int n, double const y[], double co
nst eps = 0.0);
128:
129:
130: /** Output operator for vector
131:  *
132:  * @param[in,out] s      output stream, e.g. @p cout

```

```
132:  * @param[in] v vector
133:  *
134:  * @return output stream
135:  */
136: template <class T>
137: std::ostream &operator<<(std::ostream &s, std::vector<T> const &v)
138: {
139:     for (auto vp : v) {
140:         s << vp << " ";
141:     }
142:     return s;
143: }
144:
145: /** Exchanges equal size portions of vector @p xin with all MPI processes.
146:  * The received data are return in vector @p yout .
147:  *
148:  * @param[in] xin input vector
149:  * @param[out] yout output vector
150:  * @param[in] icommm MPI communicator
151:  *
152:  */
153: void ExchangeAll(std::vector<double> const &xin, std::vector<double> &yout, MPI_Comm
const &icommm = MPI_COMM_WORLD);
154:
155: /** Exchanges equal size portions of vector @p xin with all MPI processes.
156:  * The received data are return in vector @p xin .
157:  *
158:  * @param[in,out] xin input/output vector
159:  * @param[in] icommm MPI communicator
160:  *
161:  */
162: void ExchangeAllInPlace(std::vector<double> &xin, MPI_Comm const &icommm = MPI_COMM_W
ORLD);
163:
164:
165:
166: #endif
```