```
    1: #include "vdop.h"
    2: #include "geom.h"
    3: #include "getmatrix.h"
    4: #include "jacsolve.h"
    5: #include "userset.h"
    6:
    7: #include <cassert>
    8: #include <cmath>
    9: #include <iostream>
   10: #include <vector>
   11: using namespace std;
   12:
   13: // ########################################################################
   14: //      ParMesh const & mesh,
   15: void JacobiSolve(CRS_Matrix const &SK, vector<double> const &f, vector<double> &u)
   16: {
   17:     const double omega   = 1.0;
   18:     const int    maxiter = 300;              //LISA: lowered the maxiter
   19:     const double tol  = 1e-4,                // tolerance //LISA: lowered the toller
ance
   20:                  tol2 = tol * tol;           // tolerance^2
   21:
   22:     int nrows = SK.Nrows();                  // number of rows == number of columns
   23:     assert( nrows == static_cast<int>(f.size()) && f.size() == u.size() );
   24:
   25:     cout << endl << " Start Jacobi solver for " << nrows << " d.o.f.s"  << endl;
   26:     //  Choose initial guess
   27:     for (int k = 0; k < nrows; ++k) {
   28:         u[k] = 0.0;                          //  u := 0
   29:     }
   30:
   31:     vector<double> dd(nrows);                // matrix diagonal
   32:     vector<double>  r(nrows);                // residual
   33:     vector<double>  w(nrows);                // correction
   34:
   35:     SK.GetDiag(dd);                          //  dd := diag(K)
   36:     ////DebugVector(dd);{int ijk; cin >> ijk;}
   37:
   38:     //  Initial sweep
   39:     SK.Defect(r, f, u);                      //  r := f - K*u
   40:
   41:     vddiv(w, r, dd);                         //  w := D^{-1}*r
   42:     const double sigma0 = dscapr(w, r);      // s0 := <w,r>
   43:
   44:     // Iteration sweeps
   45:     int iter  = 0;
   46:     double sigma = sigma0;
   47:     while ( sigma > tol2 * sigma0 && maxiter > iter)  // relative error
   48:         //while ( sigma > tol2 && maxiter > iter)         // absolute error
   49:     {
   50:         ++iter;
   51:         vdaxpy(u, u, omega, w );             //  u := u + om*w
   52:         SK.Defect(r, f, u);                  //  r := f - K*u
   53:         vddiv(w, r, dd);                     //  w := D^{-1}*r
   54:         sigma = dscapr(w, r);                // s0 := <w,r>
   55: //              cout << "Iteration " << iter << " : " << sqrt(sigma/sigma0) << endl;
   56:     }
   57:     cout << "aver. Jacobi rate :  " << exp(log(sqrt(sigma / sigma0)) / iter) << "  (
" << iter << " iter)" << endl;
   58:     cout << "final error: " << sqrt(sigma / sigma0) << " (rel)    " << sqrt(sigma) <<
 " (abs)\n";
   59:
   60:     return;
   61: }
   62:
   63:
   64:
   65: void JacobiSmoother(Matrix const &SK, std::vector<double> const &f, std::vector<doub
```

```
le> &u,
   66:                     std::vector<double> &r, int nsmooth, double const omega, bool ze
ro)
   67: {
   68:     // ToDO: ensure compatible dimensions
   69:
   70:     int const nnodes = static_cast<int>(u.size());
   71:     if (zero) {             // assumes initial solution is zero
   72:         DiagPrecond(SK, f, u, omega);
   73:         --nsmooth;                              // first smoothing sweep done
   74:     }
   75:
   76:     auto const &D = SK.GetDiag();               // accumulated diagonal of matrix @p SK
.
   77:     for (int ns = 1; ns <= nsmooth; ++ns) {
   78:         SK.Defect(r, f, u);                     //  r := f - K*u
   79: #pragma omp parallel for
   80:         for (int k = 0; k < nnodes; ++k) {
   81:             // u := u + om*D^{-1}*r
   82:             u[k] = u[k] + omega * r[k] / D[k]; // MPI: distributed to accumulated ve
ctor needed
   83:         }
   84:     }
   85:
   86:     return;
   87: }
   88:
   89: void DiagPrecond(Matrix const &SK, std::vector<double> const &r, std::vector<double>
 &w,
   90:                  double const omega)
   91: {
   92:     // ToDO: ensure compatible dimensions
   93:     auto const &D = SK.GetDiag();         // accumulated diagonal of matrix @p SK.
   94:     int const nnodes = static_cast<int>(w.size());
   95: #pragma omp parallel for
   96:     for (int k = 0; k < nnodes; ++k) {
   97:         w[k] = omega * r[k] / D[k];       // MPI: distributed to accumulated vector n
eeded
   98:     }
   99:
  100:     return;
  101: }
  102:
  103:
  104: Multigrid::Multigrid(Mesh const &cmesh, int const nlevel)
  105:     : _meshes(cmesh, nlevel),
  106:       _SK(), _u(_meshes.size()), _f(_meshes.size()), _d(_meshes.size()), _w(_meshes.
size()),
  107:       _Pc2f()
  108: {
  109:     cout << "\n.....................  in Multigrid::Multigrid  ................
\n";
  110:     // Allocate Memory for matrices/vectors on all levels
  111:     for (size_t lev = 0; lev < Nlevels(); ++lev) {
  112:         _SK.push_back( FEM_Matrix(_meshes[lev]) );  // CRS matrix
  113:         const auto nn = _SK[lev].Nrows();
  114:         _u[lev].resize(nn);
  115:         _f[lev].resize(nn);
  116:         _d[lev].resize(nn);
  117:         _w[lev].resize(nn);
  118:         auto vv = _meshes[lev].GetFathersOfVertices();
  119:         cout << vv.size() << endl;
  120:     }
  121:     // Intergrid transfer operators
  122:     //cout << "\n.....................  in Multigrid::Multigrid  Prolongation ...
.............\n";
  123:     //_Pc2f.push_back( BisectInterpolation(vector<int>(0)) ); // no prolongation to
coarsest grid
```

```
  124:     _Pc2f.push_back( BisectIntDirichlet() ); // no prolongation to coarsest grid
  125:     for (size_t lev = 1; lev < Nlevels(); ++lev) {
  126:                     //cout << lev << endl;
  127:                     //cout << _meshes[lev].GetFathersOfVertices () << endl;
  128:         _Pc2f.push_back( BisectIntDirichlet( _meshes[lev].GetFathersOfVertices (), _
meshes[lev-1].Index_DirichletNodes ()  )  );
  129:                     //cout << _Pc2f.back().Nrows() << "  " << _Pc2f.back().Ncols() <
< endl;
  130:     }
  131:     cout << "\n.....................................\n";
  132: }
  133:
  134: Multigrid::~Multigrid()
  135: {}
  136:
  137: void Multigrid::DefineOperators()
  138: {
  139:     for (size_t lev = 0; lev < Nlevels(); ++lev) {
  140:         DefineOperator(lev);
  141:     }
  142:     return;
  143: }
  144:
  145: // GH: Hack
  146: void Multigrid::DefineOperator(size_t lev)
  147: {
  148:     _SK[lev].CalculateLaplace(_f[lev]);  // fNice()  in userset.h
  149:
  150:     if (lev == Nlevels() - 1) {                 // fine mesh
  151:         _meshes[lev].SetValues(_u[lev], [](double x, double y) -> double
  152:         { return x *x * std::sin(2.5 * M_PI * y); }
  153:                                );
  154:     }
  155:     else {
  156:         _meshes[lev].SetValues(_u[lev], f_zero);
  157:     }
  158:
  159:     _SK[lev].ApplyDirichletBC(_u[lev], _f[lev]);
  160:
  161:     return;
  162: }
  163:
  164: void Multigrid::JacobiSolve(size_t lev)
  165: {
  166:     assert(lev < Nlevels());
  167:     ::JacobiSolve(_SK[lev], _f[lev], _u[lev]);
  168: }
  169:
  170: void Multigrid::MG_Step(size_t lev, int const pre_smooth, bool const bzero, int nu)
  171: {
  172:     assert(lev < Nlevels());
  173:     int const post_smooth = pre_smooth;
  174:
  175:     if (lev == 0) { // coarse level
  176:         JacobiSmoother(_SK[lev], _f[lev], _u[lev], _d[lev],  100, 1.0, false);
  177:     }
  178:     else {
  179:         JacobiSmoother(_SK[lev], _f[lev], _u[lev], _d[lev],  pre_smooth, 0.85, bzero
);
  180:
  181:         if (nu > 0) {
  182:
  183:             _SK[lev].Defect(_d[lev], _f[lev], _u[lev]);   //   d := f - K*u
  184:             _Pc2f[lev].MultT(_d[lev], _f[lev - 1]);       // f_H := R*d
  185:             //DefectRestrict(_SK[lev], _Pc2f[lev], _f[lev - 1], _f[lev], _u[lev]); /
/ f_H := R*(f - K*u)
  186:
  187:                     //_meshes[lev-1].Visualize(_f[lev - 1]);        // GH: Visualize
```

```
: f_H should be 0 on Dirichlet B.C.
  188:
  189:             MG_Step(lev - 1, pre_smooth, true, nu);       // solve  K_H * u_H =f_H
with u_H:=0
  190:             for (int k = 1; k < nu; ++k) {
  191:                 // W-cycle
  192:                 MG_Step(lev - 1, pre_smooth, false, nu);  // solve  K_H * u_H =f_H
  193:             }
  194:
  195:             _Pc2f[lev].Mult(_w[lev], _u[lev - 1]);        // w := P*u_H
  196:
  197:             vdaxpy(_u[lev], _u[lev], 1.0, _w[lev] );      // u := u + tau*w
  198:         }
  199:
  200:         JacobiSmoother(_SK[lev], _f[lev], _u[lev], _d[lev],  post_smooth, 0.85, fals
e);
  201:
  202:     }
  203:
  204:     return;
  205: }
  206:
  207: void Multigrid::MG_Solve(int pre_smooth, double eps, int nu)
  208: {
  209:     size_t lev=Nlevels()-1;                 // fine level
  210:
  211:     // start with zero guess
  212:     DiagPrecond(_SK[lev], _f[lev], _w[lev], 1.0);  // w   := D^{-1}*f
  213:     //double s0 = L2_scapr(_f[lev],_w[lev]);          // s_0 := <f,w>
  214:     double s0 = dscapr(_f[lev],_w[lev]);          // s_0 := <f,w>
  215:     double si;
  216:
  217:     bool bzero = true;                          // start with zero guess
  218:     int  iter  = 0;
  219:     do
  220:     {
  221:         MG_Step(lev, pre_smooth, bzero, nu);
  222:         bzero=false;
  223:         _SK[lev].Defect(_d[lev], _f[lev], _u[lev]);    //   d := f - K*u
  224:         DiagPrecond(_SK[lev], _d[lev], _w[lev], 1.0);  // w   := D^{-1}*d
  225:         //si = L2_scapr(_d[lev],_w[lev]);                 // s_i := <d,w>
  226:         si = dscapr(_d[lev],_w[lev]);                // s_i := <d,w>
  227:         ++iter;
  228:     } while (si>s0*eps*eps);
  229:
  230:
  231:     cout << "\nrel. error: " << sqrt(si/s0) << "  ( " << iter << " iter.)" << endl;
  232:     return;
  233: }
  234:
  235:
  236: void JacobiSolveMPI(ParMesh const &mesh, CRS_Matrix const &SK, vector<double> const
&f, vector<double> &u)
  237: {
  238:     const double omega   = 1.0;
  239:     const int    maxiter = 300;
  240:     const double tol  = 1e-4,
  241:                  tol2 = tol * tol;
  242:
  243:     int nrows = SK.Nrows();
  244:     assert(nrows == static_cast<int>(f.size()) && f.size() == u.size());
  245:
  246:     vector<double> dd(nrows);
  247:     vector<double> r(nrows);
  248:     vector<double> w(nrows);
  249:
  250:     SK.GetDiag(dd);     <-- accu (dd)
  251:     SK.Defect(r, f, u);
```

```
252:     vddiv(w, r, dd);
253:
254:     double sigma0 = dscapr(w, r);
255:     double sigma  = sigma0;
256:
257:     int iter = 0;
258:     while (sigma > tol2 * sigma0 && iter < maxiter)
259:     {
260:         ++iter;
261:         vddiv(w, r, dd);        // w = D^-1 * r
262:         mesh.VecAccu(w);        // make w consistent across processes
263:         vdaxpy(u, u, omega, w); // u := u + omega * w
264:         SK.Defect(r, f, u);
265:         sigma = dscapr(w, r);
266:     }
267:
268:     if (mesh.MyRank() == 0)
269:     {
270:         cout << "aver. Jacobi rate : "
271:              << exp(log(sqrt(sigma / sigma0)) / iter)
272:              << "   (" << iter << " iter)" << endl;
273:
274:         cout << "final error: "
275:              << sqrt(sigma / sigma0) << " (rel)    "
276:              << sqrt(sigma) << " (abs)\n";
277:     }
278:
279: }
280:
281:
```