

10:02:26 解 CD 方程 (黃楓南)

```
/*$Id: main.c,v 1.7 2000/08/24 22:43:38 bsmith Exp $*/
static char help[] =
"Solves 2d convection-diffusion equation, which is discretized by Galerkin/least-squares finite
elements on triangular grid.\n\
Options:\n\
-show_solution create three files for visualizing: coordinate, meshfile, and solution files.\n\
-show_griddata print the local index sets and local to global mappings \n\
-show_matrix visualize the sparsity structure of the CD matrix.\n\
-show_grid visualize the global and local grids with numbering.\n";

/*
The file appctx.h includes all the data structures used by this code
*/
#include "appctx.h"

#ifdef __FUNC__
#define __FUNC__ "main"
int main(int argc,char **argv)
{
    int          ierr,rank;
    AppCtx      *appctx;    /* contains all the data used by this PDE solver */

    PetscFunctionBegin;

    /* 1. Initialize PETSc */

    ierr = PetscInitialize(&argc,&argv,PETSC_NULL,help); CHKERRQ(ierr);
    ierr = MPI_Comm_rank(PETSC_COMM_WORLD,&rank);CHKERRQ(ierr);

    /* 2. Load the grid database -- in appload.c */
    ierr = AppCtxCreate(PETSC_COMM_WORLD,&appctx); CHKERRQ(ierr);

    /*******
    printf("AppCtxCreate() DONE \n");
    /*******

    /* 3. Setup the linear system and solve it -- in appalgebra.c */
    ierr = AppCtxSolve(appctx); CHKERRQ(ierr);
    /*******
    printf("AppCtxSolve() DONE \n");
    /*******

    /* 4. output data -- in appoutput.c */
    ierr = AppCtxOutput(appctx); CHKERRQ(ierr);
    /*******
    printf("AppCtxOutput() DONE \n");
    /*******

    /* 5. Destroy all datastructures -- in appload.c */
    ierr = AppCtxDestroy(appctx); CHKERRQ(ierr);

    /* 6. Close down PETSc and stop the program */

    ierr = PetscFinalize(); CHKERRQ(ierr);

    PetscFunctionReturn(0);
}
```

```
/*$Id: appctx.h,v 1.0 2007/08/09 Boulder, CO, USA  hwang $*/
/*
    Defines data structures for writing a simple element based PDE code
    for solving scalar PDE problems like the convective-diffusive equation.
*/

#if !defined(__APPCTX_H)
#define __APPCTX_H

#include "petscao.h"          /* allows using the PETSc AODATA-base routines for grid information */
#include "petscksp.h"         /* allows using PETSc linear solvers */
#include "petscpf.h"
#include "petscpc.h"

/*-----

    The AppGrid data structure:
    contains all the information about the grid cells, vertices boundaries etc.
    It is created by Appload() (see appload.c) from the AO database.
*/
typedef struct {
    /****** Data structures for cells *****/
    int cell_n; /* the number of cells on this processor,*/
    int cell_global_n;
    /* ---- coordinates of each of the 4 vertices corresponding to each cell
        cell_coords[0],cell_coords[1] represent x,y of the first cell's first vertice
        cell_coords[0],cell_coords[1] represent x,y of the first cell's second vertice etc, */
    double *cell_coords;

    /* ---- index for each of the 4 vertices of a given cell in the local (per processor) numbering */
    int *cell_vertex;
    double *vertex_coords;
    /****** Data structures for vertices *****/
    int vertex_n; /* number of distinct vertices on local cells, including ghost vertices*/
    int vertex_local_n; /* number of distinct vertices on local cells, excluding ghost vertices */
    PetscInt vertex_global_n;

    /* ---- local to global mapping for vertices, i.e. if you apply ltog to a list of
        vertices in local (per processor) numbering it will return them in global (parallel) numbering */
    ISLocalToGlobalMapping ltog;

    ISLocalToGlobalMapping ltog2; /* new */

    /****** Data structures for the boundary conditions *****/
    IS vertex_boundary; /* local indices of vertices on the boundary */
    IS vertex_bcell;
    int boundary_n; /* number of vertices on boundary (including ghost vertices) */
    int bcell_n; /* number of cells on boundary (including ghost vertices) */
    double *boundary_values; /* work space for the boundary values */
    double *boundary_coords; /* the coordinates of the boundary points */

    /****** Data structures for graphics *****/
    IS iscell; /* cells owned by this processor in global numbering */
    IS isvertex; /* new */

    IS iscell2;
    IS isvertex2;

    PetscTruth geometric_asm_flag; /* Geometric asm */
    PetscTruth geometric_asm_element_flag; /* element based overlapping asm */
    PetscTruth geometric_asm_node_flag; /* node based overlapping asm */
    int geometric_asm_overlap;
}
```

```
/* FIX THIS */ int      *global_cell_vertex;      /* vertices for each local cell in global numbering */

} AppGrid;

/*-----

The AppAlgebra data structure:
contains all the linear algebra objects needed to solve the linear
problem. It is created in appalgebra.c
*/
typedef struct {
    Vec b;          /* Global vector for the rhs */
    Vec x;          /* Global vector for the solution */
    Mat A;          /* Global matrix for the discrete convective-diffusive eq */
} AppAlgebra;

/*-----

The AppView data structure:
contains information about what is to be displayed and where
*/
typedef struct {
    PetscTruth show_solution; /* plots solution in Matlab (using bscript.m) */
    PetscTruth show_matrix;   /* displays sparsity pattern of matrix */
    PetscTruth show_griddata; /* dumps grid database, orderings etc to screen */
    PetscTruth show_grid;     /* plots the grid with numbering of cells, vertices etc. */
    PetscTruth summary;
    FILE *summaryfile;

    PetscDraw drawlocal; /* graphics window for drawing local per processor part of
global grid */
    PetscDraw drawglobal; /* graphics window for drawing global (parallel grid) */
} AppView;

/* new geometrical based additive schwarz */

typedef struct {

    PetscTruth pc_geo_asm_flag;
    int pc_geo_asm_overlap;
} AppEquations;

/*-----

The AppElement data structure:
contains information about the finite element basis functions on the
REFERENCE ELEMENT and then work space used to contain results in computing
the local element and element load.
*/
typedef struct {
    /* ***** same for all elements, i.e. for the reference element***** */
    double RefVal[3][3]; /* values of the reference interpolating functions at the Gauss pts */
    double RefDx[3][3];
    double RefDy[3][3];

    double weights[3]; /* quadrature weights */

    /* *****computed for each element while computing the stiffness ***** */

    double dx[3][3], dy[3][3]; /* values of the local interpolating functions at the Gauss pts */
    double detDh[3];

    double xy[6]; /* the images of the Gauss pts in the local element */
}
```

```
-
double rhsresult[3]; /* results of local integrations */
double localelementresult[3][3];

PetscTruth kappa_flag;

double kappa; /* diffusivity */
double *coords; /* pointer to coords of current cell */
double tau[3]; /* stabilization parameter on Gaussian points */
PF      rhs;
double a1[3],a2[3]; /* velocity components */

} AppElement;

/*-----
AppCtx:
    entire application context; any data in the computation can be access
    through this.
*/
typedef struct {
    MPI_Comm    comm;
    AOData      aodata;
    AppGrid     grid;
    AppAlgebra  algebra;
    AppView     view;
    AppElement  element;
    PF          bc;
    AppEquations equations;
} AppCtx;

/*-----*/
/* function declarations */

extern int AppCtxCreate(MPI_Comm,AppCtx **);
extern int AppCtxDestroy(AppCtx *);

extern int AppCtxOutput(AppCtx*);

extern int AppCtxSetLocal(AppCtx *);
extern int AppCtxSolve(AppCtx*);

extern int AppCtxCreateRhs(AppCtx*);
extern int AppCtxCreateMatrix(AppCtx*);
extern int AppCtxSetMatrix(AppCtx*);
extern int AppCtxSetRhs(AppCtx*);
extern int SetBoundaryConditions(AppCtx*);
extern int SetMatrixBoundaryConditions(AppCtx *appctx);

extern int ComputerHSElement(AppElement *phi);
extern int ComputeLocalElement(AppElement *phi);

extern int ComputeTau(AppElement *phi);

extern int SetReferenceElement(AppCtx* appctx);
extern int SetLocalElement(AppElement *phi );
extern int MonitorFunction(KSP ksp, int its, double norm, void *mctx);

double u(double x, double y); /* 1st velocity component */
double v(double x, double y); /* 2nd velocity component */

-/extern int AppCtxEfield(AppCtx*);

#endif
-
```

```
/*$Id: appelement.c,v 1.3 2000/08/24 22:43:38 bsmith Exp $*/
#include "appctx.h"

/*
    Performs the numerical integration and setup at the element level
*/

/* A -
    Returns the value of the shape function or its xi or eta
    derivative at any point in the REFERENCE element. xi and
    eta are the coordinates in the reference element.
*/

#undef __FUNC__
#define __FUNC__ "InterpolatingFunctionsElement"
static int InterpolatingFunctionsElement(int partial,int node,double xi,double eta, double *value)
{
    /* 4 node bilinear interpolation functions */
    PetscFunctionBegin;
    switch (partial) {
    case 0: /* function itself */
        switch (node) {
            case 0: *value = 1.-xi-eta; break;
            case 1: *value = xi; break;
            case 2: *value = eta; break;
            /* case 3: *value = 0.25 * (1-xi) * (1+eta); break;*/
        } break;
    case 1: /* d() / d(xi) */
        switch (node) {
            case 0: *value = -1.; break;
            case 1: *value = 1.; break;
            case 2: *value = 0.; break;
            /*case 3: *value = 0.25 * (-1) * (1+eta); break;*/
        } break;
    case 2: /* d() / d(eta) */
        switch (node) {
            case 0: *value = -1.; break;
            case 1: *value = 0.; break;
            case 2: *value = 1.; break;
            /* case 3: *value = 0.25 * (1-xi) * (1); break;*/
        } break;
    }
    PetscFunctionReturn(0);
}

#undef __FUNC__
#define __FUNC__ "SetReferenceElement"
/*
    Computes the numerical integration (Gauss) points and evaluates the basis
    funtions at these points. This is done ONCE for the element, the information
    is stored in the AppElement data structure and then used repeatedly to compute
    each element load and element stiffness. Uses InterpolatingFunctionsElement().
*/
int SetReferenceElement(AppCtx* appctx)
{
    int i,j;
    int bn = 3; /* number of basis functions*/
    int qn = 3; /* number of quadrature points */
    double gx[3],gy[3]; /* Gauss points: */
    AppElement *phi = &appctx->element;

    PetscFunctionBegin;
}
```

```
/* set Gauss points */
gx[0] = 1./6.; gx[1] = 2./3.;
gx[2] = 1./6.;

gy[0] = 1./6.; gy[1] = 1./6.;
gy[2] = 2./3.;

/* set quadrature weights */
phi->weights[0] = 1./6.; phi->weights[1] = 1./6.;
phi->weights[2] = 1./6.;

/* Set the reference values,
   i.e., the values of the basis functions at the Gauss points */
for(i=0;i<bn;i++){ /* loop over functions*/
    for(j=0;j<qn;j++){/* loop over Gauss points */
        InterpolatingFunctionsElement(0,i,gx[j],gy[j],&(appctx->element.RefVal[i][j]));
        InterpolatingFunctionsElement(1,i,gx[j],gy[j],&(appctx->element.RefDx[i][j]));
        InterpolatingFunctionsElement(2,i,gx[j],gy[j],&(appctx->element.RefDy[i][j]));
    }
}
PetscFunctionReturn(0);
}

/*-----
   B - Computes derivative information for each real element (this is
        called once per element. This data is used in C) and D) to compute
        the element load and stiffness.
*/
#undef __FUNC__
#define __FUNC__ "SetLocalElement"
int SetLocalElement(AppElement *phi)
{
    /* the coordinates array consists of pairs (x[0],y[0],...,x[2],y[2]) representing
       the images of the support points for the 3 basis functions */
    int i,j;
    int bn = 3,qn = 3; /* number of basis functions, number of quadrature points */
    double Dh[3][2][2],Dhinv[3][2][2]; /* the Jacobian and inverse of the Jacobian */

    PetscFunctionBegin;
    /* The function h takes the reference element to the local (true) element.
        $h(x,y) = \sum(i) \text{ of } \alpha_i \phi_i(x,y),$ 
       where  $\alpha_i$  is the image of the support point of the ith basis function */

    /*Values, i.e., (x(xi, eta), y(xi, eta)),
       the images of the Gauss points in the local element */
    for(i=0;i<qn;i++){ /* loop over the Gauss points */
        phi->xy[2*i] = 0; phi->xy[2*i+1] = 0;
        for(j=0;j<bn;j++){/*loop over the basis functions, and support points */
            phi->xy[2*i] += phi->coords[2*j]*phi->RefVal[j][i];
            phi->xy[2*i+1] += phi->coords[2*j+1]*phi->RefVal[j][i];
        }
    }

    /* Jacobian */
    for(i=0;i<qn;i++){ /* loop over Gauss points */
        Dh[i][0][0] = 0; Dh[i][0][1] = 0; Dh[i][1][0] = 0; Dh[i][1][1] = 0;
        for(j=0;j<bn;j++){/* loop over functions */
            Dh[i][0][0] += phi->coords[2*j]*phi->RefDx[j][i];
            Dh[i][0][1] += phi->coords[2*j]*phi->RefDy[j][i];
            Dh[i][1][0] += phi->coords[2*j+1]*phi->RefDx[j][i];
            Dh[i][1][1] += phi->coords[2*j+1]*phi->RefDy[j][i];
        }
    }
}
```

```

/* Determinant of the Jacobian */
* for(i=0; i<qn; i++){ /* loop over Gauss points */
    phi->detDh[i] = Dh[i][0][0]*Dh[i][1][1] - Dh[i][0][1]*Dh[i][1][0];
}

/* Inverse of the Jacobian */
for(i=0; i<qn; i++){ /* loop over Gauss points */
    Dhinv[i][0][0] = Dh[i][1][1]/phi->detDh[i];
    Dhinv[i][0][1] = -Dh[i][0][1]/phi->detDh[i];
    Dhinv[i][1][0] = -Dh[i][1][0]/phi->detDh[i];
    Dhinv[i][1][1] = Dh[i][0][0]/phi->detDh[i];
}

/* Notice that phi~ = phi(h), so Dphi~ = Dphi*Dh, (chain rule)
   so Dphi~ = Dphi*(Dh)inv (multiply by (Dh)inv */
/* partial of phi at h(gauss pt) times Dhinv */
/* loop over Gauss, the basis functions, then d/dx or d/dy */
for(i=0; i<qn; i++){ /* loop over Gauss points */
    for(j=0; j<bn; j++){ /* loop over basis functions */
        phi->dx[j][i] = phi->RefDx[j][i]*Dhinv[i][0][0] + phi->RefDy[j][i]*Dhinv[i][1][0];
        phi->dy[j][i] = phi->RefDx[j][i]*Dhinv[i][0][1] + phi->RefDy[j][i]*Dhinv[i][1][1];
    }
}

PetscFunctionReturn(0);
}
/*-----
   C - Computes an element load
*/
#undef __FUNC__
#define __FUNC__ "ComputeRHS"
int ComputeRHSElement(AppElement *phi)
{
    int i,k,ierr;
    int bn = 3,qn = 3; /* number of basis functions, number of quadrature points */
    PetscReal f;

    PetscFunctionBegin;

    for (i=0; i<bn; i++) { /* loop over basis functions */
        phi->rhsresult[i] = 0.0;
        for (k=0; k<qn; k++) { /* loop over Gauss points */

            /* evaluate right hand side function */
            ierr = PFApply(phi->rhs,1,&phi->xy[2*k],&f);CHKERRQ(ierr);
            phi->rhsresult[k]+=phi->weights[k]*f*PetscAbsReal(phi->detDh[k])*

(phi->RefVal[i][k]-phi->tau[k]*(phi->a1[k]*phi->dx[i][k]+phi->a2[k]*phi->dy[i][k])
        );
    }
}
PetscFunctionReturn(0);
}
/* -----

D - ComputeLocalElement: computes integrals of convective, diffusive and GLS stabilized terms

The formula used is, for example for diffusive term

integral (given element) of <grad phi_j', grad phi_i'> =
    integral over (ref element) of
    <(grad phi_j composed with h)*(grad h)^-1,

```

```
(grad phi_i composed with h)*(grad h)^-1>*det(grad h).
*   this is evaluated by quadrature:
    = sum over Gauss points, above evaluated at Gauss pts
*/
#undef __FUNC__
#define __FUNC__ "ComputeLocalElement"
int ComputeLocalElement(AppElement *phi)
{
    int i,j,k;
    int bn = 3,qn = 3; /* number of basis functions, number of Gauss points */

    PetscFunctionBegin;
    /* could even do half as many by exploiting symmetry */
    for(i=0;i<bn;i++){ /* loop over first basis function */
        for(j=0;j<bn;j++){ /* loop over second basis function */
            phi->localelementresult[i][j] = 0.;
        }
    }

    /* Now Integral. 1. diffusive term is kappa (<d(phi)/dx ,d(phi)/dx>+ <d(phi)/dy ,d(phi)/dy>)
                    2. convective term is <a_1 d(phi)/dx+a_2 d(phi)/dy,phi>
                    3. stabilized term is <a_1 d(phi)/dx+a_2 d(phi)/dy,a_1 d(phi)/dx+a_2 d(phi)/dy> */

    for (i=0; i<bn; i++) { /* loop over first basis function */
        for (j=0; j<bn; j++) { /* loop over second basis function*/
            for (k=0; k<qn; k++) { /* loop over Gauss points */
                phi->localelementresult[i][j]
                += phi->weights[k]*PetscAbsReal(phi->detDh[k])*
                (phi->kappa*(phi->dx[i][k]*phi->dx[j][k]+phi->dy[i][k]*phi->dy[j][k]) /*diffusive term */
                +(phi->a1[k]*phi->dx[j][k]+phi->a2[k]*phi->dy[j][k])*phi->RefVal[i][k] /*convective term */
                +(phi->a1[k]*phi->dx[j][k]+phi->a2[k]*phi->dy[j][k])
                *(phi->a1[k]*phi->dx[i][k]+ phi->a2[k]*phi->dy[i][k])*phi->tau[k]); /* GLS term */
            }
        }
    }
    PetscFunctionReturn(0);
}

#undef __FUNC__
#define __FUNC__ "ComputeTau"
int ComputeTau(AppElement *phi)
{
    int i,k;
    int bn=3, qn=3; /* basis count, quadrature count */
    double alpha[3],Peh[3];
    double edge0,edge1,edge2;
    double x[3],y[3],hK;

    /* mK=1/3 and lambda=1.0 */

    /* a) Compute the element diameter */

    for (i=0;i<bn;i++) {
        x[i]=phi->coords[2*i];
        y[i]=phi->coords[2*i+1];
    }
    edge0=sqrt((x[1]-x[0])*(x[1]-x[0])+(y[1]-y[0])*(y[1]-y[0]));
    edge1=sqrt((x[2]-x[1])*(x[2]-x[1])+(y[2]-y[1])*(y[2]-y[1]));
    edge2=sqrt((x[2]-x[0])*(x[2]-x[0])+(y[2]-y[0])*(y[2]-y[0]));
    //printf("edge0 is %f \t edge1 is %f \t edge2 is %f \n",edge0,edge1,edge2);
    hK=edge0;
```

```
    if (edge1>edge0) { hK=edge1;}
    if (edge2>hK) {hK=edge2;}

    for(k=0;k<qn;k++){ /* gauss points */

/* b) Compute the velocity and its magnitude alpha */

    alpha[k]=0.0;
    phi->a1[k]=0.0; phi->a2[k]=0.0;

    phi->a1[k]=u(phi->xy[2*k],phi->xy[2*k+1]);
    phi->a2[k]=v(phi->xy[2*k],phi->xy[2*k+1]);

    alpha[k]=sqrt(phi->a1[k]*phi->a1[k]+phi->a2[k]*phi->a2[k]);

/* c) Compute the element Peclet number */

    Peh[k]=alpha[k]*hK/3.0/phi->kappa;

/* d) Compute the stability parameters tau */

    if (Peh[k]>=1)
    { /* Convective dominant */
        phi->tau[k]=hK/(2.0*alpha[k]);
    }
    else { /* diffusive dominant */
        phi->tau[k]=hK*hK/(6.0*phi->kappa);
    }
}

PetscFunctionReturn(0);

}
```

```
/*$Id: appalgebra.c,v 1.0 2000/08/24 22:43:38 bsmith Exp $*/
#include "appctx.h"
#include "math.h"

/*
Sample right hand side and boundary conditions
*/
#undef __FUNC__
#define __FUNC__ "pde_rhs"
int pde_rhs(void *dummy,int n,double *xx,double *f)
{
double x = xx[0], y = xx[1];
PetscFunctionBegin;
// *f = 1. ;
*f = 0.;
PetscFunctionReturn(0);
}

#undef __FUNC__
#define __FUNC__ "pde_bc"
int pde_bc(void *dummy,int n,double *xx,double *f)
{
double x = xx[0], y = xx[1];
PetscFunctionBegin;

// skew 45
//*****
/*
if ( fabs(x) == 0. && fabs(y) == 0. ) { *f = 1. ; }
if ( fabs(x) == 0. && fabs(y) == 1. ) { *f = 1. ; }
if ( fabs(x) == 1. && fabs(y) == 0. ) { *f = 0. ; }
if ( fabs(x) == 1. && fabs(y) == 1. ) { *f = 0. ; }
if ( fabs(x) == 0. && fabs(y) <= 1.0 ) { *f = 1 ; }
if ( fabs(x) == 1.0 && fabs(y) <= 1.0 ) { *f = 0 ; }
*/
/*
{ *f = 0.; }
if ( fabs(x) < 1.e-4 ) { *f = 1.; }
if ( fabs(x-0.25) < 0.26 && fabs(y) < 1.e-4 ) { *f = 1.; }
*/

if ( fabs(x) == 0. && fabs(y) <= 1.0 ) *f = 1 ;
else if ( fabs(x) < 0.5 && fabs(y) == 0 ) *f = 1 ;
else if ( fabs(x) == 0.5 && fabs(y) == 0 ) *f = 0 ;
else if ( fabs(x) == 1.0 && fabs(y) == 1.0 ) *f = 0 ;
else *f = 0 ;

//*****
// Test_case2 60 degree
//*****

// *f = 0;
//if ( fabs ( x ) <= 0.5 && fabs(y) == 0 ) *f = 1;
//if ( fabs ( x ) == 0 && fabs ( y ) <= 1.0 ) *f = 1;
//if ( fabs( x - 0.75 ) <= 0.25 + 1.e-4 && fabs(y) <= 1.e-4 ) *f = 0 ;
//boundary 1
//else if ( fabs(x - 1.0 ) <= 1.e-4 && fabs(y-sqrt(3.0)/4.0)<= sqrt(3.0)/4.0+1.e-4 ) *f = 0 ;
//boundary 2
//else if ( fabs(x - 1.0 ) <= 1.e-4 && fabs(y-1.0) <= 1.e-4 ) *f = 0 ;
//boundary 3
//else *f = 1 ;

//if ( fabs(x - 0.75 ) <= 0.25 + 1.e-4 && fabs(y) <= 1.e-4 ) *f = 0 ; //boundary 1
//else if ( fabs(x-1.)<= 1.e-4 && fabs( y - sqrt(3.0)/4.0 ) <= (sqrt(3.0)/4.0)+ 1.e-4 ) *f = 0 ;
```

```
//else *f = 1.0;

//*****/

// Thermal Boundary Problem
//*****
/*
{*f = 1 ;}          // boundary 4
if ( fabs( x - 1.0 ) <= 1.e-4 && fabs( y ) <= 0.5 + 1.e-4 ) *f = 2.0*y ; // boundary 1
if ( fabs( x ) <= 1.0 + 1.e-4 && fabs(y) <= 1.e-4 ) *f = 0 ; // boundary 2
if ( fabs( x ) <= 1.e-4 && fabs( y ) <= 0.5 + 1.e-4 ) *f = 1 ; // boundary 3
*/
//*****/

// Bubble Ramp Problem
//*****

//{*f = 0. ;}
//if ( fabs(x-0.5) <= 1.e-4 && fabs(y-0.5) <= 1.e-4 ) *f = 0 ; // boundary 1
//else if ( fabs(x-0.5) <= 1.e-4 && fabs(y-0.5) <= 1.e-4 ) *f = 0 ; // boundary 2
//else if ( fabs(x-1.0) <= 0.5+1.e-4 && fabs(y) <= 1.e-4 ) *f = 0 ; // boundary 3
//else if ( fabs(x) <= 1.5+1.e-4 && fabs(y-1) <= 1.e-4 ) *f = 0 ; // boundary 4
//else if ( fabs(x-1.5) <= 1.e-4 && fabs(y) <= 1.0+1.e-4 ) *f = 0 ; // boundary 5
//else if ( fabs(x) <= 1.e-4 && fabs(y-0.75) <= 0.25+1.e-4 ) *f = 0 ; // boundary 6

//*****/

PetscFunctionReturn(0);
}

#undef __FUNC__
#define __FUNC__ "u"
double u(double x,double y)
{
//return 1.0;
// Test_case1
return 1.0/sqrt(2.0);
// Test_case2
//return 1.0 / 2.0 ;
// Test_case3
//return (2.0*y) ;
// Test_case4
//return 1.0;
}

#undef __FUNC__
#define __FUNC__ "v"
double v(double x, double y)
{
//return 0.;
// Test_Cae1
return 1.0/sqrt(2.0);
// Test_cade2
//return sqrt(3.0)/2.0 ;
//Test_case3
//return 0.;
//Test_case4
//return 0;
}

/*
Sets up the linear system associated with the PDE and solves it
*/
#undef __FUNC__
```

```
#define __FUNC__ "AppCtxSolve"
int AppCtxSolve(AppCtx* appctx)
{
    AppAlgebra *algebra = &appctx->algebra;
    AppGrid *grid = &appctx->grid;
    MPI_Comm comm = appctx->comm;
    KSP ksp;
    int ierr, N1, one=1, its;
    PC pc; /* new */
    double tmp;
    PetscTruth flg;
    PetscFunctionBegin;

    /* Set the functions to use for the right hand side and Dirichlet boundary */
    ierr = PFCreate(comm, 2, 1, &appctx->element.rhs); CHKERRQ(ierr);
    ierr = PFSetOptionsPrefix(appctx->element.rhs, "rhs_"); CHKERRQ(ierr);
    ierr = PFSetType(appctx->element.rhs, PFQUICK, (void*)pde_rhs); CHKERRQ(ierr);
    ierr = PFSetFromOptions(appctx->element.rhs); CHKERRQ(ierr);

    ierr = PFCreate(comm, 2, 1, &appctx->bc); CHKERRQ(ierr);
    ierr = PFSetOptionsPrefix(appctx->bc, "bc_"); CHKERRQ(ierr);
    ierr = PFSetType(appctx->bc, PFQUICK, (void*)pde_bc); CHKERRQ(ierr);
    ierr = PFSetFromOptions(appctx->bc); CHKERRQ(ierr);

    /* A) Set the quadrature values for the reference element */
    ierr = SetReferenceElement(appctx); CHKERRQ(ierr);

    /* 1) Create vector to contain load and various work vectors */
    ierr = AppCtxCreateRhs(appctx); CHKERRQ(ierr);

    /* 2) Create the sparse matrix with correct nonzero pattern */
    ierr = AppCtxCreateMatrix(appctx); CHKERRQ(ierr);

    /* 3) Set the right hand side values into the load vector */
    ierr = AppCtxSetRhs(appctx); CHKERRQ(ierr);

    /* 4) Set the matrix entries */
    ierr = AppCtxSetMatrix(appctx); CHKERRQ(ierr);

    /* 5) Set the rhs boundary conditions -
       this also creates initial guess that satisfies boundary conditions */
    ierr = SetBoundaryConditions(appctx); CHKERRQ(ierr);

    /* view sparsity structure of the matrix before bc applied */
    if (appctx->view.show_matrix) {
        ierr = PetscPrintf(PETSC_COMM_WORLD, "The discrete CD matrix, before bc applied\n"); CHKERRQ(ierr);
        ierr = MatView(appctx->algebra.A, PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
    }

    /* 6) Set the matrix boundary conditions */
    ierr = SetMatrixBoundaryConditions(appctx); CHKERRQ(ierr);

    /* view sparsity structure of the matrix after bc applied */
    if (appctx->view.show_matrix) {
        ierr = PetscPrintf(PETSC_COMM_WORLD, "The stiffness matrix, after bc applied\n"); CHKERRQ(ierr);
        ierr = MatView(appctx->algebra.A, PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
    }

    PreLoadBegin(PETSC_TRUE, "Solver setup and solve");
}
```

```
/* KSP 1) Create a linear solver context (required) */
ierr = KSPCreate(comm,&ksp);CHKERRQ(ierr);

/* Ksp 2) Set an operator. Here the matrix defines the linear
system also serves the preconditionig matrix (required) */
ierr = KSPSetOperators(ksp,algebra->A,algebra->A,DIFFERENT_NONZERO_PATTERN);CHKERRQ(ierr);

/* KSP 3a) Set linear solver defaults for this problem (Optional) */
ierr = KSPGetPC(ksp,&pc); CHKERRQ(ierr);
ierr = PCSetType(pc,PCJACOBI); CHKERRQ(ierr);
ierr = KSPSetTolerances(ksp,1.e-7,PETSC_DEFAULT,
    PETSC_DEFAULT,PETSC_DEFAULT); CHKERRQ(ierr);

/* KSP 3b) Set runtime options, e.g.
-ksp_type -pc_type -ksp_rtol <rtol>
This option will override those specified as long as
KSPSetFromOptions as called after any customization
routine */
ierr = KSPSetFromOptions(ksp);CHKERRQ(ierr);
ierr = KSPSetUp(ksp);CHKERRQ(ierr);

/* KSP 3c) Set a one-level additive Schwarz preconditioner */
if(appctx->grid.geometric_asm_flag){
    ierr = PCSetType(pc,PCASM); CHKERRQ(ierr);
    ierr = PCASMSetType(pc,PC_ASM_BASIC); CHKERRQ(ierr);
    ierr = PCASMSetLocalSubdomains(pc,one,&grid->isvertex2);CHKERRQ(ierr);
}

if (appctx->view.summary) {
    ierr = PetscFOpen(PETSC_COMM_WORLD,"summary.m",
        "a",&(appctx->view.summaryfile));CHKERRQ(ierr);

    ierr = PetscFPrintf(PETSC_COMM_WORLD,appctx->view.summaryfile,
        "\n NEW \n\n"); CHKERRQ(ierr);
    ierr = KSPSetMonitor(ksp,MonitorFunction,(void*)appctx,0);CHKERRQ(ierr);
}
/*
PreloadStage("Solve");
*/
/* KSP 4) Solve the linear system */
ierr = KSPSolve(ksp,algebra->b,algebra->x); CHKERRQ(ierr);

/* Summary */
ierr=VecGetSize(algebra->b,&N1); CHKERRQ(ierr);
ierr=PetscPrintf(PETSC_COMM_WORLD,"Number of unknowns = %d\n",N1);
    CHKERRQ(ierr);
ierr = KSPGetIterationNumber(ksp,&its); CHKERRQ(ierr);
ierr=PetscPrintf(PETSC_COMM_WORLD,"Number of linear iterations= %d\n",its);
    CHKERRQ(ierr);
if (appctx->view.summary) {
    ierr = PetscFPrintf(PETSC_COMM_WORLD,appctx->view.summaryfile,"SUMMARY\n");
}
```

```
        CHKERRQ(ierr);
    ierr = VecGetSize(appctx->algebra.b,&N1); CHKERRQ(ierr);
    ierr = PetscFPrintf(PETSC_COMM_WORLD,appctx->view.summaryfile,"Number of unknowns = %d\n",N1);
        CHKERRQ(ierr);
    ierr = PetscFPrintf(PETSC_COMM_WORLD,appctx->view.summaryfile,
        "Number of linear iterations =%d\n",its); CHKERRQ(ierr);
    ierr = PetscOptionsGetReal(PETSC_NULL,"-ksp_atol",&tmp,&flg);CHKERRQ(ierr);
    ierr = PetscFPrintf(PETSC_COMM_WORLD,appctx->view.summaryfile,"SLES atol=%g\n",tmp);
        CHKERRQ(ierr);

    ierr = PetscOptionsGetReal(PETSC_NULL,"-ksp_rtol",&tmp,&flg);CHKERRQ(ierr);
    ierr = PetscFPrintf(PETSC_COMM_WORLD,appctx->view.summaryfile,"SLES rtol=%g\n",tmp);
        CHKERRQ(ierr);
}

/* KSP 5) Free the solver data structures */
    ierr = KSPDestroy(ksp);CHKERRQ(ierr);

    PreLoadEnd();

    ierr = PFDestroy(appctx->bc);CHKERRQ(ierr);
    ierr = PFDestroy(appctx->element.rhs);CHKERRQ(ierr);

    PetscFunctionReturn(0);
}

/*-----
1 - Generates the "global" parallel vectors to contain the
right hand side and solution.
*/
#undef __FUNC__
#define __FUNC__ "AppCtxCreateRhs"
int AppCtxCreateRhs(AppCtx *appctx)
{
    AppGrid      *grid = &appctx->grid;
    AppAlgebra    *algebra = &appctx->algebra;
    MPI_Comm      comm = appctx->comm;
    int           ierr;

    PetscFunctionBegin;
    /* Create vector to contain load, local size should be number of vertices on this proc. */
    ierr = VecCreateMPI(comm,grid->vertex_local_n,PETSC_DETERMINE,&algebra->b);CHKERRQ(ierr);

    /* This allows one to set entries into the vector using the LOCAL numbering: via VecSetValuesLocal() */
    ierr = VecSetLocalToGlobalMapping(algebra->b,grid->ltog);CHKERRQ(ierr);

    /* Generate the vector to contain the solution */
    ierr = VecDuplicate(algebra->b,&algebra->x);CHKERRQ(ierr);

    PetscFunctionReturn(0);
}

/*-----
2 - Generates the "global" parallel matrix
*/
#undef __FUNC__
#define __FUNC__ "AppCtxCreateMatrix"
int AppCtxCreateMatrix(AppCtx* appctx)
{
    AppAlgebra    *algebra = &appctx->algebra;
```

```
AppGrid      *grid      = &appctx->grid;
MPI_Comm      comm = appctx->comm;
int           ierr;
PetscTruth    flg;

PetscFunctionBegin;
ierr = PetscOptionsHasName(PETSC_NULL, "-mat_type", &flg); CHKERRQ(ierr);
if (!flg) {
    /* use very rough estimate for nonzeros on and off the diagonal */
    ierr = MatCreateMPIAIJ(comm, grid->vertex_local_n, grid->vertex_local_n,
        PETSC_DETERMINE, PETSC_DETERMINE, 9, 0, 3, 0, &algebra->A); CHKERRQ(ierr);
} else {
    /* ierr = MatCreate(comm, grid->vertex_local_n, grid->vertex_local_n,
        PETSC_DETERMINE, PETSC_DETERMINE, &algebra->A); CHKERRQ(ierr); */
    /*
        ierr = MatSetTypeFromOptions(algebra->A); CHKERRQ(ierr);
        ierr = MatSetOption(algebra->A); CHKERRQ(ierr); */
}

/* Allows one to set values into the matrix using the LOCAL numbering, via MatSetValuesLocal() */
ierr = MatSetLocalToGlobalMapping(algebra->A, grid->ltag); CHKERRQ(ierr);

PetscFunctionReturn(0);
}

/*-----
3 - Computes the entries in the right hand side and sets them into the parallel vector
Uses B and C
*/
#undef __FUNC__
#define __FUNC__ "AppCtxSetRhs"
int AppCtxSetRhs(AppCtx* appctx)
{
    /*----- Context informatrion -----*/
    AppGrid      *grid = &appctx->grid;
    AppAlgebra    *algebra = &appctx->algebra;
    AppElement    *phi = &appctx->element;

    /*----- Local Variables -----*/
    int           ierr, i;
    int           *vertex_ptr;
    int           bn = 3; /* number of basis functions */
    int           vertexn = 3; /* number of degrees of freedom */

    PetscFunctionBegin;
    /* loop over local cells */
    for(i=0; i<grid->cell_n; i++){

        /* coords_ptr points to the coordinates of the current cell */
        phi->coords = grid->cell_coords + 2*bn*i; /*number of cell coords */

        /* compute the values of basis functions on this element */
        ierr = SetLocalElement(phi); CHKERRQ(ierr);

        /* Compute the stabilization parameter tau and local velocity fields a1 and a2 */
        ierr = ComputeTau(phi); CHKERRQ(ierr);

        /* compute the element load (integral of f with the 4 basis elements) */
        /* values get put into phi->rhsresult */
        ierr = ComputeRHSElement(phi); CHKERRQ(ierr);

        /*----- Set Values -----*/
        /* vertex_ptr points to place in the vector to set the values */
    }
}
```

```

    vertex_ptr = grid->cell_vertex + vertexn*i;

    ierr = VecSetValuesLocal(algebra->b,bn,vertex_ptr,phi->rhsresult,ADD_VALUES);CHKERRQ(ierr);
}
ierr = VecAssemblyBegin(algebra->b);CHKERRQ(ierr);
ierr = VecAssemblyEnd(algebra->b);CHKERRQ(ierr);
PetscFunctionReturn(0);
}

/*-----
    4 - Computes the element CD matrices and stick into
    global CD matrix. Uses B and D.
*/
#undef __FUNC__
#define __FUNC__ "AppCtxSetMatrix"
int AppCtxSetMatrix(AppCtx* appctx)
{
    /****** Context information *****/
    AppAlgebra *algebra = &appctx->algebra;
    AppGrid *grid = &appctx->grid;
    AppElement *phi = &appctx->element;

    /****** Local Variables *****/
    int i,ierr;
    int *vertex_ptr;
    int bn = 3; /* number of basis functions */
    int vertexn = 3; /* number of degrees of freedom */

    PetscFunctionBegin;

    /* loop over local cells */
    for(i=0;i<grid->cell_n;i++){

        /* coords_ptr points to the coordinates of the current cell */
        phi->coords = grid->cell_coords + 2*bn*i; /*number of cell coords */
        /* compute the values of basis functions on this element */
        ierr = SetLocalElement(phi);CHKERRQ(ierr);

        /* Compute the stabilization parameter tau and local velocity fields a1 and a2 */

        ierr = ComputeTau(phi); CHKERRQ(ierr);

        /* Compute the local element results, which is returned in phi->elementresult */

        ierr = ComputeLocalElement(phi);CHKERRQ(ierr);

        /****** Set Values *****/
        /* vertex_ptr points to place in the matrix to set the values */
        vertex_ptr = grid->cell_vertex + vertexn*i;

        ierr = MatSetValuesLocal(algebra->A,vertexn,vertex_ptr,vertexn,vertex_ptr,
            (double*)phi->localelementresult,ADD_VALUES);CHKERRQ(ierr);
    }
    ierr = MatAssemblyBegin(algebra->A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
    ierr = MatAssemblyEnd(algebra->A,MAT_FINAL_ASSEMBLY);CHKERRQ(ierr);
    PetscFunctionReturn(0);
}

/*-----
    5 - Apply the Dirichlet boundary conditions (see 6 also).
    This places the Dirichlet function value on the right hand side
    and 6 sticks a row of the identity matrix on the left side

```

```
    thus forcing the solution at the given points to match the
    Dirichlet function.
*/
#undef __FUNC__
#define __FUNC__ "SetBoundaryConditions"
int SetBoundaryConditions(AppCtx *appctx)
{
    /***** Context informatrion *****/
    AppAlgebra *algebra = &appctx->algebra;
    AppGrid *grid = &appctx->grid;

    /***** Local Variables *****/
    int ierr,i;
    int *vertex_ptr;
    PetscReal zero = 0.0;

    PetscFunctionBegin;

    /* -----
       Apply Dirichlet boundary conditions
       ----- */

    /* need to set the points on RHS corresponding to vertices on the boundary to
       the desired value. */

    /* get list of vertices on the bounday */

    ierr = ISGetIndices(grid->vertex_boundary,&vertex_ptr);CHKERRQ(ierr);
    for(i=0;i<grid->boundary_n;i++)
    {
        /* evaluate boundary condition function at point */
        ierr =
PFApply(appctx->bc,1,&grid->boundary_coords[2*i],&grid->boundary_values[i]);CHKERRQ(ierr);
    }

    if (grid->boundary_n!=0)
    {
        /* set the right hand side values at those points */
        ierr =
VecSetValuesLocal(algebra->b,grid->boundary_n,vertex_ptr,grid->boundary_values,INSERT_VALUES);CHKERRQ(ierr);
    }

    /* set initial guess satisfying boundary conditions */
    ierr = VecSet(algebra->x,zero);CHKERRQ(ierr);
    if (grid->boundary_n!=0)
    {
        ierr =
VecSetValuesLocal(algebra->x,grid->boundary_n,vertex_ptr,grid->boundary_values,INSERT_VALUES);CHKERRQ(ierr);
    }

    ierr = VecAssemblyBegin(algebra->x);CHKERRQ(ierr);
    ierr = VecAssemblyEnd(algebra->x);CHKERRQ(ierr);

    ierr = ISRestoreIndices(grid->vertex_boundary,&vertex_ptr);CHKERRQ(ierr);

    ierr = VecAssemblyBegin(algebra->b);CHKERRQ(ierr);
    ierr = VecAssemblyEnd(algebra->b);CHKERRQ(ierr);

    PetscFunctionReturn(0);
}
```

```
}`
/*-----
    6 - Set the matrix boundary conditions (see also 5). Replace the corresponding
        rows in the matrix with the identity.
*/
#undef __FUNC__
#define __FUNC__ "SetMatrixBoundaryConditions"
int SetMatrixBoundaryConditions(AppCtx *appctx)
{
    /****** Context informatrion *****/
    AppAlgebra *algebra = &appctx->algebra;
    AppGrid *grid = &appctx->grid;

    /****** Local Variables *****/
    double one = 1.0;
    int ierr;

    PetscFunctionBegin;
    ierr = MatZeroRowsLocalIS(algebra->A, grid->vertex_boundary, one); CHKERRQ(ierr);
    PetscFunctionReturn(0);
}

#undef __FUNC__
#define __FUNC__ "MonitorFunction"
int MonitorFunction(KSP ksp, int its, double norm, void *dappctx)
{
    AppCtx *appctx = (AppCtx *)dappctx;
    int ierr;
    int rank;

    PetscFunctionBegin;
    ierr=MPI_Comm_rank(PETSC_COMM_WORLD,&rank); CHKERRQ(ierr);
    ierr=PetscFPrintf(PETSC_COMM_WORLD,appctx->view.summaryfile,
        "Sles(%d)=%20.15e;\n",its+1,norm); CHKERRQ(ierr);
    PetscFunctionReturn(0);
}
```

```
/*$Id: appload.c,v 1.0 2007/08/ 20:58:18 bsmith Exp $*/
#include "appctx.h"

/*
    Loads the triangular grid database from a file and sets up the local
    data structures.
*/

#undef __FUNC__
#define __FUNC__ "AppCtxCreate"
/*
    AppCtxCreate - Fills in the data structures using the grid information from a AOData file.
*/
int AppCtxCreate(MPI_Comm comm,AppCtx **appctx)
{
    int ierr;
    PetscTruth flag;
    PetscViewer binary;
    char filename[256];
    AppView *view;
    AppGrid *grid;
    AppElement *phi;

    PetscFunctionBegin;
    /* (*appctx) = (AppCtx*)PetscMalloc(sizeof(AppCtx));CHKPTRQ(*appctx); */
    ierr = PetscMalloc(sizeof(AppCtx),appctx); CHKERRQ(ierr);
    (*appctx)->comm = comm;

    /*-----
    - - Load in the grid database
    -----*/
    /* user indicates grid database with -f; defaults to "gridfile" if not give */
    ierr = PetscOptionsGetString(0,"-f",filename,256,&flag);CHKERRQ(ierr);
    if (!flag) {ierr = PetscStrcpy(filename,"gridfile");CHKERRQ(ierr);}

    /* Open the database and read in grid (each processor gets a portion of the grid data)*/
    ierr = PetscViewerBinaryOpen((*appctx)->comm,filename,FILE_MODE_READ,&binary);CHKERRQ(ierr);

    ierr = AODataCreateBasic((*appctx)->comm, &(*appctx)->aodata); CHKERRQ(ierr)
    ierr = AODataLoadBasic(binary,&(*appctx)->aodata);CHKERRQ(ierr);
    ierr = PetscViewerDestroy(binary);CHKERRQ(ierr);

    /* allows "values" key or segment in data base to be refered to as "coords" */
    ierr = AODataAliasAdd((*appctx)->aodata,"coords","values");CHKERRQ(ierr);

    /*-----
    setup viewing (graphics) options
    -----*/
    view = &(*appctx)->view;
    ierr = PetscOptionsHasName(PETSC_NULL,"-show_solution",&view->show_solution);CHKERRQ(ierr);
    ierr = PetscOptionsHasName(PETSC_NULL,"-show_matrix",&view->show_matrix);CHKERRQ(ierr);
    ierr = PetscOptionsHasName(PETSC_NULL,"-show_griddata",&view->show_griddata);CHKERRQ(ierr);
    ierr = PetscOptionsHasName(PETSC_NULL,"-show_grid",&view->show_grid);CHKERRQ(ierr);
    ierr = PetscOptionsHasName(PETSC_NULL,"-summary",&view->summary);CHKERRQ(ierr);
    /* setup geometric asm options */

    grid = &(*appctx)->grid;
    ierr = PetscOptionsHasName(0,"-geometric_asm",&grid->geometric_asm_flag); CHKERRQ(ierr);
    ierr = PetscOptionsGetInt(0,"-geometric_asm_overlap", &grid->geometric_asm_overlap,&flag);
    CHKERRQ(ierr);

    PetscPrintf(PETSC_COMM_WORLD,"geometric_asm_overlap: %d\n",grid->geometric_asm_overlap);
    PetscPrintf(PETSC_COMM_WORLD,"flag: %d\n",flag);
}
```

```
.
ierr = PetscOptionsHasName(0, "-geometric_asm_element", &grid->geometric_asm_element_flag);
ierr = PetscOptionsHasName(0, "-geometric_asm_node", &grid->geometric_asm_node_flag); CHKERRQ(ierr);

PetscPrintf(PETSC_COMM_WORLD, "geometric_asm_element_flag: %d\n", grid->geometric_asm_element_flag);
PetscPrintf(PETSC_COMM_WORLD, "geometric_asm_node_flag: %d\n", grid->geometric_asm_node_flag);

phi = &(*appctx)->element;
phi->kappa_flag = PETSC_FALSE;
ierr = PetscOptionsGetReal(PETSC_NULL, "-kappa", &phi->kappa, &phi->kappa_flag);
if (!phi->kappa_flag) {
    phi->kappa = 1.e-4;
}

/*-----
   Setup the local data structures; this generates a local numbering of cells and vertices
   -----*/
ierr = AppCtxSetLocal(*appctx); CHKERRQ(ierr);

PetscFunctionReturn(0);
}

#undef __FUNC__
#define __FUNC__ "AppCtxSetLocal"
/*
   AppCtxSetLocal - Sets the local numbering data structures for the grid.

   Here the basic local data structures are set up. It was found that the code was simplified by
   making all data structures and computations as cell-based as possible. The one exception was the
   boundary condtions, and so the code for applying boundary conditions is kept separate for clarity.

   This is by far the most complicated part of the program, it uses the "magic" of the AODATA object
   to organize all the grid data to make it suitable for a finite element computation.

   Each processor is assigned a certain number of cells. These are numbered 0 to cell_n[0]-1,
   cell_n[0] to cell_n[1]-1, cell_n[1] to cell_n[2]-1, etc. So each processor has cell_n[rank] cells.
   The vertices are then divided up among the processors.
*/
int AppCtxSetLocal(AppCtx *appctx)
{
    AODATA    ao = appctx->aodata;
    AppGrid   *grid = &appctx->grid;
    /* IS      isvertex; */

    int       *vertex_ptr, i, ierr, rank;

    PetscTruth flag;

    PetscFunctionBegin;
    ierr = MPI_Comm_rank(PETSC_COMM_WORLD, &rank); CHKERRQ(ierr);

    /*-----
   /* Set up cell related data structures
   /*-----
   /*
   Get the local cells and vertices, and the local to global mapping for vertices
   - Partitions the cells across the processors, then
   - Partitions the vertices, subservient to the elements

   grid->iscell    - the cells owned by this processor (in global numbering of cells)
   grid->isvertex  - the vertices owned by this processor (in global numbering of vertices)
```

```

    *      grid->ltog      - the mapping from local numbering of vertices to global numbering of
vertices
    */

/* algebraic */

ierr = AODataView(ao,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);

ierr = AODataPartitionAndSetupLocal(ao,"cell","df",
    &grid->iscell,&grid->isvertex,&grid->ltog);CHKERRQ(ierr);

/* Geometric ASM */
if(appctx->grid.geometric_asm_flag){

    ierr=ISDuplicate(grid->iscell,&grid->iscell2); CHKERRQ(ierr);
    ierr=ISDuplicate(grid->isvertex,&grid->isvertex2); CHKERRQ(ierr);

    if (appctx->grid.geometric_asm_element_flag){

        for( i = 0; i < grid->geometric_asm_overlap-1; i++ ){
            ierr =AODataKeyGetNeighborsIS(ao,"cell",grid->iscell2,
                &grid->iscell2);CHKERRQ(ierr); }

        ierr=AODataSegmentGetReducedIS(ao,"cell","df",
            grid->iscell2,&grid->isvertex2); CHKERRQ(ierr);
    } else {
        for( i = 0; i < grid->geometric_asm_overlap-1; i++ ){
            ierr =AODataKeyGetNeighborsIS(ao,"df",
                grid->isvertex2,&grid->isvertex2); CHKERRQ(ierr);
- ./      ierr =AODataKeyGetNeighborsIS(ao,"vertex",
//          grid->isvertex2,&grid->isvertex2); CHKERRQ(ierr);
        }
    }
}

if (appctx->view.show_griddata) {
    ierr = PetscPrintf(PETSC_COMM_WORLD,"The Application Ordering Database (AOData:\n");CHKERRQ(ierr);
    ierr = AODataView(ao,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
    ierr = PetscPrintf(PETSC_COMM_WORLD,"\nAODataPartitionAndSetupLocal generates \n grid->iscell:\n");
CHKERRQ(ierr);
    ierr = ISView(grid->iscell,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);

    ierr = PetscPrintf(PETSC_COMM_WORLD,"\n grid->isvertex:\n"); CHKERRQ(ierr);
    ierr = ISView(grid->isvertex,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);

    ierr = PetscPrintf(PETSC_COMM_WORLD,"\n grid->ltog:\n"); CHKERRQ(ierr);
    ierr = ISLocalToGlobalMappingView(grid->ltog,PETSC_VIEWER_STDOUT_WORLD); CHKERRQ(ierr);
}

/*      Get the number of cells on processor */
ierr = ISGetLocalSize(grid->iscell,&grid->cell_n);CHKERRQ(ierr);

/* the total number of vertices which are belong to some cell on this processor, */
ierr = ISGetLocalSize(grid->isvertex,&grid->vertex_n);CHKERRQ(ierr);

/* the number of vertices which were partionned onto (storage actually belongs on) this processor */
ierr =
AODataKeyGetInfo(ao,"df",&grid->vertex_global_n,&grid->vertex_local_n,PETSC_NULL,PETSC_NULL);CHKERRQ(ierr);
- ./      ierr =
AODataKeyGetInfo(ao,"vertex",&grid->vertex_global_n,&grid->vertex_local_n,PETSC_NULL,PETSC_NULL);CHKERR
- -

```

```
Q(ierr);
.
/* get the vertex coordinates */
ierr = AODataSegmentGetIS(ao,"df","coords",grid->isvertex,
    (void *)&grid->vertex_coords);CHKERRQ(ierr);
// ierr = AODataSegmentGetIS(ao,"vertex","coords",grid->isvertex,
//    (void *)&grid->vertex_coords);CHKERRQ(ierr);

if (appctx->view.show_griddata) {
    ierr = PetscSynchronizedPrintf(PETSC_COMM_WORLD,"%d] grid->cell_n %d grid->vertex_n %d
grid->vertex_local_n %d\n",
rank,grid->cell_n,grid->vertex_n,grid->vertex_local_n);CHKERRQ(ierr);
    ierr = PetscSynchronizedPrintf(PETSC_COMM_WORLD,"%d] AODataSegmentGetIS generated vertex
coordinates\n",rank);CHKERRQ(ierr);
    ierr = PetscRealView(2*grid->vertex_n,grid->vertex_coords,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
}

/* Get the vertex local numbering for each local cell */
ierr = AODataSegmentGetLocalIS(ao,"cell","df",grid->iscell,(void *)&grid->cell_vertex);CHKERRQ(ierr);
if (appctx->view.show_griddata) {
    ierr = PetscSynchronizedPrintf(PETSC_COMM_WORLD,"%d] AODataSegmentGetLocalIS generated vertex
local numbering for each cell\n",rank);CHKERRQ(ierr);
    ierr = PetscIntView(3*grid->cell_n,grid->cell_vertex,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
}

/* Get the global vertex number for each cell, for graphics only */

ierr = PetscMalloc(3*grid->cell_n*sizeof(int),&grid->global_cell_vertex); CHKERRQ(ierr);
ierr =
ISLocalToGlobalMappingApply(grid->ltag,3*grid->cell_n,grid->cell_vertex,grid->global_cell_vertex);CHKERR
Q(ierr);

/* Get the coordinates of the cell vertices */
ierr = AODataSegmentExists(ao,"cell","df",&flag);CHKERRQ(ierr);
//if (!flag) {
ierr = AODataSegmentGetIS(ao,"cell","df",grid->iscell,(void *)&grid->cell_coords);CHKERRQ(ierr);
//} else {
ierr = PetscMalloc(2*3*grid->cell_n*sizeof(double),&grid->cell_coords); CHKERRQ(ierr);
    for (i=0; i<3*grid->cell_n; i++) {
        grid->cell_coords[2*i] = grid->vertex_coords[2*grid->cell_vertex[i]];
        grid->cell_coords[2*i+1] = grid->vertex_coords[2*grid->cell_vertex[i]+1];
    }
//}

/*****
/* Set up boundary related data structures */
*****/

/* Generate a list of local vertices that are on the boundary */
ierr =
AODataKeyGetActiveLocalIS(ao,"df","NS1",grid->isvertex,0,&grid->vertex_boundary);CHKERRQ(ierr);
// ierr =
AODataKeyGetActiveLocalIS(ao,"df","boundary",grid->isvertex,0,&grid->vertex_boundary);CHKERRQ(ierr);
// ierr =
AODataKeyGetActiveLocalIS(ao,"vertex","boundary",grid->isvertex,0,&grid->vertex_boundary);CHKERRQ(ierr);
;

/* get the number of local boundary vertices */
ierr = ISGetLocalSize(grid->vertex_boundary,&grid->boundary_n);CHKERRQ(ierr);
ierr = PetscMalloc((grid->boundary_n+1)*sizeof(double),&grid->boundary_values); CHKERRQ(ierr);
ierr = PetscMalloc(2*(grid->boundary_n+1)*sizeof(double),&grid->boundary_coords); CHKERRQ(ierr);
```

```
/* now extract the needed vertex coordinates for the boundary ones */
ierr = ISGetIndices(grid->vertex_boundary,&vertex_ptr);CHKERRQ(ierr);
for(i = 0; i<grid->boundary_n; i++){
    grid->boundary_coords[2*i] = grid->vertex_coords[2*vertex_ptr[i]];
    grid->boundary_coords[2*i+1] = grid->vertex_coords[2*vertex_ptr[i]+1];
}
ierr = ISRestoreIndices(grid->vertex_boundary,&vertex_ptr);CHKERRQ(ierr);

if (appctx->view.show_griddata) {
    ierr = PetscSynchronizedPrintf(PETSC_COMM_WORLD,"%d] AODataKeyGetActiveLocalIS generated
grid->vertex_boundary\n",rank);CHKERRQ(ierr);
    ierr = ISView(grid->vertex_boundary,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr);
}

//*****
printf("AppCtxSetLocal() DONE \n");
//*****

PetscFunctionReturn(0);
}

#undef __FUNC__
#define __FUNC__ "AppCtxDestroy"
/*
    Frees the all the data structures in the program
*/
int AppCtxDestroy(AppCtx *appctx)
{
    int      ierr;
    AOData   ao = appctx->aodata;
    AppGrid  *grid = &appctx->grid;

    PetscFunctionBegin;
    /*
        Free the database, first we must "restore" any data we have accessed from the AODatabase
    */
    /*
        ierr = AODataSegmentRestoreLocalIS(ao,"cell","df",PETSC_NULL,(void
**)&grid->cell_vertex);CHKERRQ(ierr);
    */
    /*
        ierr = AODataSegmentRestoreIS(ao,"cell","coords",0,(void **)&grid->cell_coords);CHKERRQ(ierr);
        ierr = AODataSegmentRestoreIS(ao,"df","coords",0,(void **)&grid->vertex_coords);CHKERRQ(ierr);
        ierr = AODataDestroy(ao);CHKERRQ(ierr);
    /*
        ierr = AODataSegmentRestoreIS(ao,"vertex","coords",grid->isvertex,(void
**)&grid->vertex_coords);CHKERRQ(ierr); */
    /*
        Free the grid data
    */
    ierr = ISLocalToGlobalMappingDestroy(appctx->grid.ltog);CHKERRQ(ierr);
    ierr = PetscFree(grid->boundary_values);CHKERRQ(ierr);
    ierr = PetscFree(grid->boundary_coords);CHKERRQ(ierr);
    ierr = ISDestroy(grid->vertex_boundary);CHKERRQ(ierr);
    ierr = PetscFree(grid->global_cell_vertex);CHKERRQ(ierr);
    ierr = ISDestroy(grid->iscell);CHKERRQ(ierr);
    ierr = ISDestroy(grid->isvertex);CHKERRQ(ierr); /* NEW */

    /*
        Free the algebra
    */
    ierr = MatDestroy(appctx->algebra.A);CHKERRQ(ierr);
    ierr = VecDestroy(appctx->algebra.b);CHKERRQ(ierr);
    ierr = VecDestroy(appctx->algebra.x);CHKERRQ(ierr);
}
```

```
    ierr = PetscFree(appctx);CHKERRQ(ierr);  
    PetscFunctionReturn(0);  
}
```

```
/*$Id: appview.c,v 1.5 2000/01/17 00:09:20 bsmith Exp $*/
#include "appctx.h"

#ifdef __FUNC__
#define __FUNC__ "AppCtxOutput"
int AppCtxOutput(AppCtx* appctx)
{
    int    ierr,i,j;
    FILE    *fp;

    PetscInt *Node_indices,*element_indices,*global_cell_vertex;
    PetscScalar *global_vertex_coords;
    PetscViewer file;

    PetscFunctionBegin;
    /*----get the global incoord output
    file-----*/

    ierr = PetscMalloc(appctx->grid.vertex_global_n*sizeof(int),&Node_indices); CHKERRQ(ierr);
    for (i=0; i<appctx->grid.vertex_global_n; i++) {Node_indices[i]=i;}
    ierr = AODataSegmentGet(appctx->aodata,"df","coords",appctx->grid.vertex_global_n,Node_indices,(void
    **)&global_vertex_coords);
    /* ierr =
    AODataSegmentGet(appctx->aodata,"vertex","coords",appctx->grid.vertex_global_n,Node_indices,(void
    **)&global_vertex_coords);
    /* ierr =
    PetscRealView(2*grid->vertex_global_n,global_vertex_coords,PETSC_VIEWER_STDOUT_WORLD);CHKERRQ(ierr); */

    PetscViewerASCIIOpen(PETSC_COMM_WORLD,"CoordinatesFile1",&file);
    PetscViewerSetFormat(file,PETSC_VIEWER_ASCII_COMMON);
    PetscViewerASCIIPrintf(file,"%d\t %d\n",appctx->grid.vertex_global_n,2);
    for (i=0; i<appctx->grid.vertex_global_n; i++ )
    {
        /* PetscViewerASCIIPrintf(file,"%d\t",i); */
        for (j=0; j<2; j++)
        {
            PetscViewerASCIIPrintf(file,"%f\t",global_vertex_coords[j+2*i]);
        }
        PetscViewerASCIIPrintf(file,"\n");
    }
    PetscViewerDestroy(file);

    /*----get the global node output
    file-----*/
    ierr =
    AODataKeyGetInfo(appctx->aodata,"cell",&appctx->grid.cell_global_n,PETSC_NULL,PETSC_NULL,PETSC_NULL);CH
    KERRQ(ierr);
    ierr = PetscMalloc(appctx->grid.cell_global_n*sizeof(int),&element_indices); CHKERRQ(ierr);
    for (i=0; i<appctx->grid.cell_global_n; i++) {element_indices[i]=i;}
    ierr = AODataSegmentGet(appctx->aodata,"cell","df",appctx->grid.cell_global_n,element_indices,(void
    **)&global_cell_vertex);
    /* ierr =
    AODataSegmentGet(appctx->aodata,"cell","vertex",appctx->grid.cell_global_n,element_indices,(void
    **)&global_cell_vertex);
    PetscViewerASCIIOpen(PETSC_COMM_WORLD,"MeshFile1",&file);
    PetscViewerSetFormat(file,PETSC_VIEWER_ASCII_COMMON);
    PetscViewerASCIIPrintf(file,"%d\t %d\n",appctx->grid.cell_global_n,1);
    for (i=0; i<appctx->grid.cell_global_n; i++ )
    {
        /* PetscViewerASCIIPrintf(file,"%d\t",i); */
        for (j=0; j<3; j++)
        {
```

```
        PetscViewerASCIIPrintf(file,"%d\t",global_cell_vertex[j+3*i]);
    }
    PetscViewerASCIIPrintf(file,"\n");
}
PetscViewerDestroy(file);
/*-----*/
-----*/
PetscViewerASCIIOpen(PETSC_COMM_WORLD,"Solution.dat",&file);
PetscViewerSetFormat(file,PETSC_VIEWER_ASCII_COMMON);
VecView(appctx->algebra.x,file);
PetscViewerDestroy(file);

/*-----*/
-----*/

PetscFunctionReturn(0);
}
```