OOF2 development learning and modification notes

RCL

August 18, 2006

Contents

1 Construction of the stiffness matrix in a distributed environment (06/08/06)) 1

2 Building OOF2 modules as Windows DLLs in Visual C++ (06/08/06) 5

3 Solver performance (03/10/06) 6

4 Adding a PETSc linear solver driver (02/24/06) 7

5 Construction of OOF2 menu commands (02/22/06) 10

6 Adding a simple page and SWIG extension (02/17/06) 11

7 Matrix construction 14

8 Sequence of events after clicking 'Solve' 22

9 Generating Abaqus output 32

1 Construction of the stiffness matrix in a distributed environment (06/08/06))

I start from the following premise: that the parallel skeleton construction/refinement of S. Haan provides the following information regarding each node shared by two or more processes/processors (i.e. the node is at the interface of subdomains):

1. An index (normal skeleton node index in a particular process)
2. List of process numbers (MPI ranks) of the processes, other than the current one, that share this node

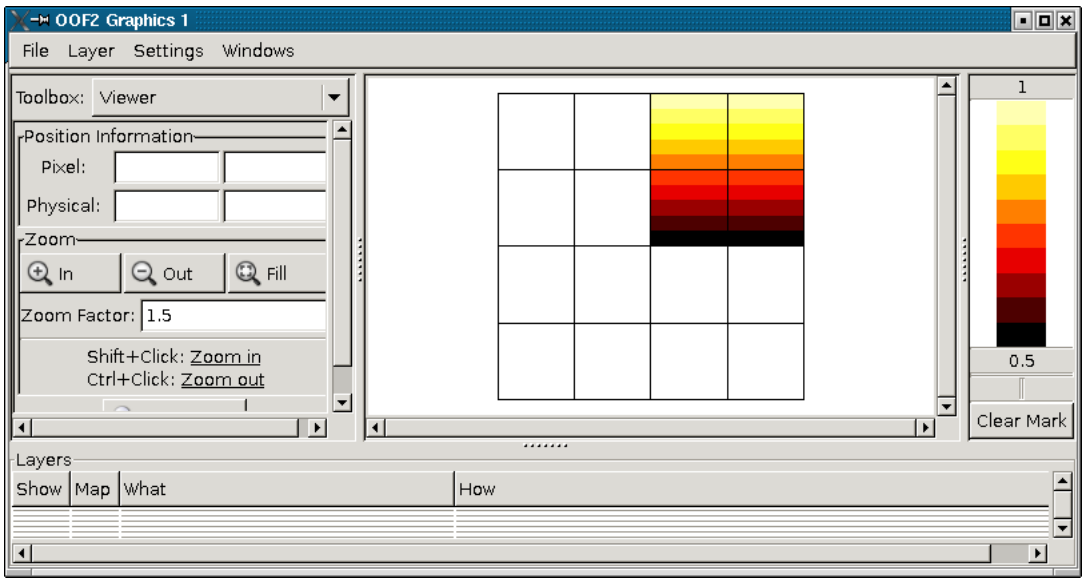

Figure 1: Contour plot for thermal problem solved in parallel. Only the result for process 0 (front end) shows up.

3. List of indices of this node in the other processes

The general outline for the combined stiffness matrix construction, starting from the mesh construction, is as follows:

1. When the funcnodes for the mesh are created, the sharing information from the skeleton nodes is passed on to the funcnodes. (see FEMesh::newFuncNode and FEMesh::newFuncNode_Shares) A shared node is "owned" by the lowest ranked process that contains the node.

2. The sharing information of the equations and degrees of freedom is inherited from the sharing of the funcnodes. When equations (nodal equations) and fields (degrees of freedom) are set, nodal equations and degrees of freedom are added to each funcnode. References to nodal equations and degrees of freedom associated with each funcnode are also maintained by the mesh in a list. For equations and degrees of freedom associated with shared funcnodes, the equations and degrees of freedom are added to a map that links them back to the funcnode (funcnodes already maintain a list of the equations and degrees of freedom, but the map is a faster way to locate the right funcnode). Since fields and equations are added to every funcnode in the same sequence, we know exactly which equation in a funcnode in one process corresponds to the equation in a shared funcnode in another process. (see FuncNode::AddField and FuncNode::AddEquation)

3. The nodal equations in the list maintained by the mesh are mapped into

indices of a (local) symmetric stiffness matrix. The main idea in symmetrizing the matrix is to use the conjugacy of the equation with its degree of freedom. The index of the equation (row) is made identical to the index of the conjugate degree of freedom. (In a very simple scalar field situation, what is happening is that a node in a FE mesh gives rise to an equation, and this node also gives one field or degree of freedom (or an unknown that is to be solved). The equation and the degree of freedom at this node are conjugate.) Another map is maintained that additionally indicates the active equations that are not "owned" by this node. The row for this equation is to be added to the row of the process that does "own" the equation. (see FEMesh::set_equation_mapping)

4. The number of degrees of freedom (should equal the number of equations) in each process is collected/gathered. This yields the size of the combined stiffness matrix as well as offsets for placing the rows and columns of the local stiffness matrix onto the larger combined stiffness matrix. (see FEMesh::set_precombined_equation_mapping)

5. A process that owns shared equations and degrees of freedom sends information about the indices of the equations and degrees of freedom in its local stiffness matrix. The other processes that share these equations and degrees of freedom pick up this information and proceed to add the corresponding equations and degree of freedom on the places indicated by the owning process. (see FEMesh::set_precombined_equation_mapping, PETScSolverDriverCore::precompute_parallel)

The data structure that encodes the node sharing information in the mesh is as follows (see femesh.h)

```
class CNodeShareInfo
{
public:
  CNodeShareInfo(std::vector<int>* procs, std::vector<int>* indices, int index)
  {
    inheritedindex=index;
    localprocrank=Rank();//from mpitools.h
    int minrank=localprocrank;
    for(std::vector<int>::size_type i=0;i<procs->size();i++)
      {
int pi=(*procs)[i];
remoteproclist.push_back(pi);
if(localprocrank>pi)
  minrank=pi;
      }

    if(minrank==localprocrank)
      {
```

```
owns=true;
      }
    else
      {
owns=false;
      }

    for(std::vector<int>::size_type i=0;i<indices->size();i++)
      {
remoteindexlist.push_back((*indices)[i]);
      }
  };
  ~CNodeShareInfo()
  {
    //Not necessary
    remoteproclist.clear();
    remoteindexlist.clear();
    localdofindexlist.clear();
    localeqnindexlist.clear();
    symmatrixdofindexlist.clear();
    symmatrixeqnindexlist.clear();
  };
  int inheritedindex;//index inherited from the skeleton nodes sharing information
  int localprocrank;//the rank of the process
  bool owns;//Does this process own the node? It does if the process is the lowest ranke
  // that share this node.
  std::vector<int> remoteproclist;//list of processes (rank numbers) that share this nod
  std::vector<int> remoteindexlist;//list of indices of this node corresponding to the o
  //The following four vectors should have the same length. These get filled up in set_e
  std::vector<int> localdofindexlist;//list of indices of the free DoFs within this node
  std::vector<int> localeqnindexlist;//list of indices of the independent NodalEqns with
  std::vector<int> symmatrixdofindexlist;//list of indices into the (probably) symmetriz
  std::vector<int> symmatrixeqnindexlist;//list of indices into the (probably) symmetriz
};
```

The shared funcnodes are linked to its sharing information via

```
std::map<FuncNode*,CNodeShareInfo*> m_nodesharemap;
```

The mapping of the elements of the local stiffness matrix onto the elements of the global stiffness matrix are encoded in

```
std::vector<int> m_precombined_freedofmap;
std::vector<int> m_precombined_indepeqnmap;
```

2 Building OOF2 modules as Windows DLLs in Visual C++ (06/08/06)

Classes that are to be exported should be declared as in this example:

```
class __declspec(dllexport) ActiveArea : public CPixelSelection
{
...
};
```

For functions,

```
__declspec(dllexport) ActiveAreaList * areaListFromPixel(const
CMicrostructure*, const ICoord*);
```

For applications to import these functions, the keyword dllexport is replaced by dllimport.

The use of dllexport or dllimport may be handled automatically by defining preprocessor directives (placed in the file oofconfig.h)

```
#ifdef WIN32
#        ifdef _WIN32DLLEXPORTINGCOMMON
#                define WIN32DLLEXPORTCOMMON(x) __declspec(dllexport) x
#                define WIN32DLLEXPORTENGINE(x) x
#        elif defined(_WIN32DLLEXPORTINGENGINE)
#                define WIN32DLLEXPORTCOMMON(x) __declspec(dllimport) x
#                define WIN32DLLEXPORTENGINE(x) __declspec(dllexport) x
#        else
#                define WIN32DLLEXPORTCOMMON(x) __declspec(dllimport) x
#                define WIN32DLLEXPORTENGINE(x) __declspec(dllimport) x
#        endif
#else
#                define WIN32DLLEXPORTCOMMON(x) x
#                define WIN32DLLEXPORTENGINE(x) x
#endif
```

then the examples above become

```
class WIN32DLLEXPORTCOMMON(ActiveArea) : public CPixelSelection
{
...
};
```

and

```
WIN32DLLEXPORTCOMMON(ActiveAreaList *) areaListFromPixel(const
CMicrostructure*, const ICoord*);
```

Notes:

1. The export statements are placed automatically in SWIG generated files by SWIG.

2. The preprocessor directives handle the case when libengine exports libcommon, and when other modules export both libcommon and libengine. A typical module explicitly links with both libcommon and libengine, not just with libengine.

3. A few headers present in unix are not present in WIN32 (those associated with threading, mutexes, certain math constants and advanced functions). To enable the definition of constants like M_PI, in visual C++ 7.1, one may define _USE_MATH_DEFINES. Mutexes might also be replaced by CRITICAL_SECTIONs.

4. The export directives also must be enabled for the parent classes and the members of the class that is exported.

5. There are also a few special compile time options for building in Windows, such as Run-Time Type Information (RTTI), no precompiled headers, etc. that are most easily seen by looking at the Visual C++ project properties.

6. There may be a .def file alternative to using __declspec(dllexport), but it may require knowing the mangled versions of the exported names.

3 Solver performance (03/10/06)

Figure 2 displays a graph of execution times for the solution of a linear system obtained from a simple thermal problem in OOF2, using various solver implementations and grid sizes. The thermal problem is heat conduction, with the left boundary set to $T = 0$, the opposite boundary set to $T = 1$, and the field is in-plane. The grid sizes are 10 (as in 10×10), 20, 30, 40, 100. The execution time is the sum of the time taken by the body of the precompute method of the solverdriver and the time taken by the call to the solve method of the solver. The legend for the solvers used is described as follows:

1. NO PETSC - the solver used is

   ```
   solver=LinearDriver(method=CGSolver(max_iterations=1000,tolerance=1e-13,precondition
   ```

2. PETSC1 - the solver used is

   ```
   solver=PETScLinearDriver(method=PETScKSPCGSolver(max_iterations=1000,relative_tolera
   ```

   This version of the petsc solver is very slow because the CSRmatrix submatrices (A and C) extracted from the master stiffness matrix are copied element-by-element (including the zeros) into petsc matrices.

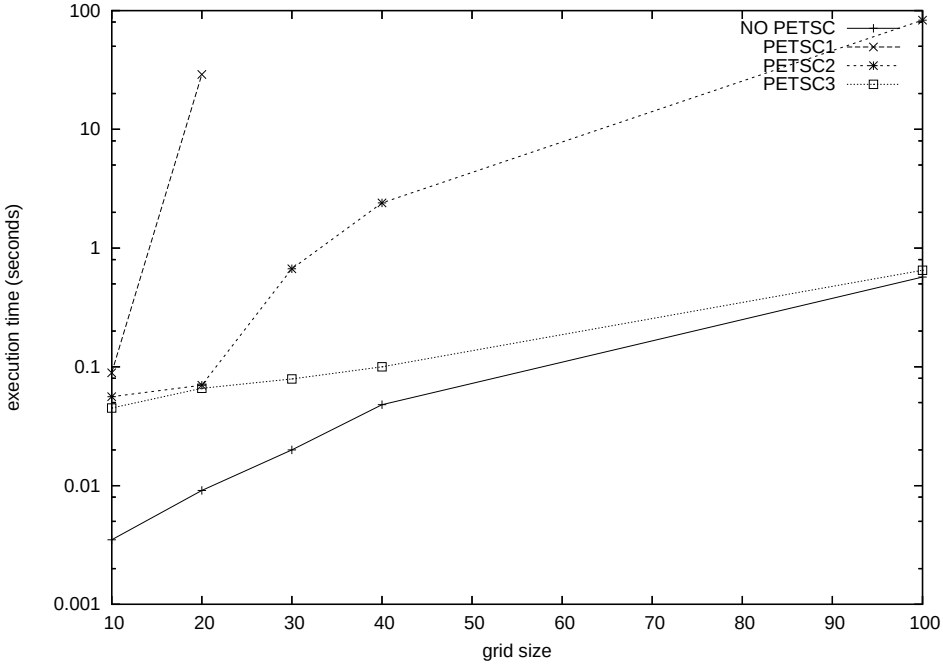

Figure 2: Comparison of solvers for various grid sizes. The execution time is the time taken by $PETScSolverDriverCore :: precompute$ (or $SolverDriverCore :: precompute$) plus the time taken by $solver->solve$.

3. PETSC2 - the solver used is similar to PETSC1, but the submatrices are copied directly to petsc from the master stiffness matrix, without creating a CSRmatrix, and skipping the zero elements (which is the one important reason for the speedup).

4. PETSC3 - similar to PETSC2, but much faster (almost as fast as NO PETSC) because the rows of the petsc matrix are preallocated.

TODO: Run tests in multiple processors. Even if this works, to solve a much bigger problem (say 1000×1000), we would have to do something at the setup stage (e.g. allocating a skeleton).

4 Adding a PETSc linear solver driver (02/24/06)

In this first iteration of the revival of PETSc in OOF2, a PETSc linear driver, providing a default KSP linear solver (figure 3), is added to the OOF2 solver page. The following files are modified (from backend to frontend level):

1. `PETSc/petsc_solver.h, petsc_solver.C`

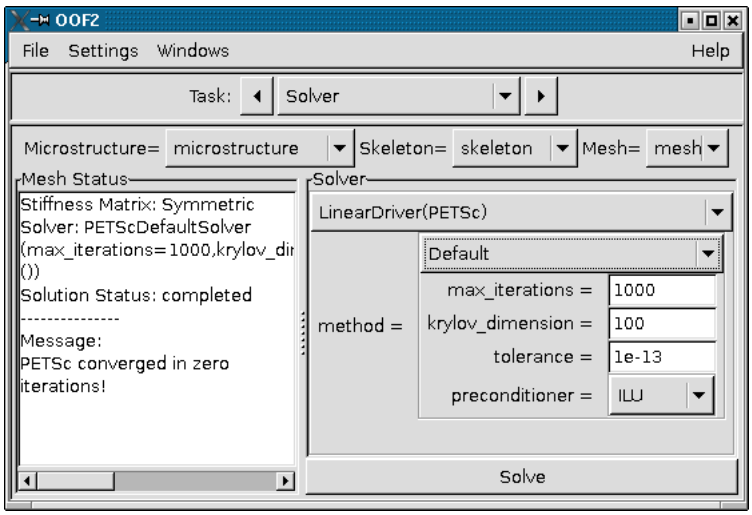

Figure 3: A simple default PETSc solver.

2. `PETSc/petsc_solver.swg`
3. `PETSc/petsc_solver.spy`
4. `solverdriver.spy`
5. `solverPage.py`
6. `PETSc/DIR.py`

In *solverdriver.spy*, added the lines

```
# ****************************
# PETScII
# ****************************
if devel.devel():
    class PETScLinearDriver(Driver):
        def __init__(self, method):
            self.method = method
            self.core = None

        def driver_apply(self, stepper, verbosity):
            self.core = SolverDriverCore(stepper, self.method, self.mesh.getObject())
            self.precompute(verbosity)
            ## precompute assigns properties, creates the mapping to
            ## symmetrizes the linear system of equations,
            ## builds the master stiffness, conductivity, and mass
            ## matrices, sets BCs
```

```
            self.apply_solver(verbosity) ## applies the linear solver. Solution is accep
            self.postcompute (verbosity)
            ## postcompute resets BCs and fixates histo

    from oof2.SWIG.engine.PETSc import petsc_solver
    registeredclass.Registration(
        'LinearDriver(PETSc)',
        Driver, ## superclass
        PETScLinearDriver, ## subclass
        ordering = 1000,
        params = [
        petsc_solver.PETScLinearSolverParameter('method',
                                                tip=parameter.emptyTipString)],
        tip="PETSc KSP linear system solver ",
        discussion=xmlmenudump.loadFile('DISCUSSIONS/engine/reg/linear_driver.xml')
        )
```

In *solverPage.py*, added the lines

```
# ****************************************************************
# PETScII
# ****************************************************************
if devel.devel():
    from oof2.SWIG.engine.PETSc import petsc_solver
    petsc_solver.PETScLinearSolverParameter.makeWidget = _makeLinearSolverWidget
# ****************************************************************
```

In *DIR.py*,

```
dirname = 'PETSc'
clib = 'oof2engine'

cfiles = ['petsc_solver.C']
swigfiles= ['petsc_solver.swg']
swigpyfiles = ['petsc_solver.spy']
hfiles = ['petsc_solver.h']

def set_clib_flags(clib):
    # *************
    # PETScII
    # *************
    clib.externalLibs.append('petsc')
    clib.externalLibs.append('petscmat')
    clib.externalLibs.append('petscvec')
    clib.externalLibs.append('petscdm')
    clib.externalLibs.append('petscksp')
    clib.externalLibDirs.append('/usr/lib/petsc/lib/libO/linux')
```

```
    clib.includeDirs.append('/usr/include/petsc')
    clib.includeDirs.append('/usr/include/mpi')
```

(Thanks to Andrew for figuring out that petscvec is also required, which also hinted at the other link flags.)

The PETSc linear driver is of course patterned after the existing OOF2 linear driver. The PETSc solver is currently written for PETSc version 2.2.0 (the latest debian package in sancho), which seems to be different from the version in the previous PETSc incarnation , although most of the PETSc calls are the same. (BTW, the current PETSc version and the documentation I used is version 2.3.1.) The PETSc solver seems to solve the problem of heat conduction with a linear temperature profile correctly. (While testing the drivers, also found a (relatively old) bug with switching between different drivers, which Andrew remedied quickly.)

The parts of the driver in python are under devel. But in the C code, I did not use the *HAVE_PETSC* preprocessor directives because the compilation can be turned off by just not including a reference to the PETSc directory in the build scripts (e.g. subdirs.append in engine/DIR.py).

TODOs: Very short term: Add preconditioners and integrate PETSc solver results with OOF2 output messages. Longer term: integrate PETSc matrix into OOF2. Currently, copying the stiffness matrix (a CSRmatrix) into a PETSc matrix (via MatSetValues) is not very efficient (although potentially not as bad as before because of an explicit copying of matrix rows only for the processor that owns that portion).

5 Construction of OOF2 menu commands (02/22/06)

The term 'menu' might be confusing because of its strong connotation to something that you click on ('instruction' or 'command' may have been better), although its hierarchical structure does resemble a traditional menu.

Here is an example of how an OOF2 menu command is constructed, in the context of the Solver page. In order to be able to invoke the menu command, (say)

```
OOF.Solver.Solve(mesh=the_mesh, solver=solver_selection)
```

within a python console or OOF2 python code, the items separated by the dots have to be added, as in (in *solvermenu.py*)

```
solvermenu = mainmenu.OOF.addItem(oofmenu.OOFMenuItem(
    'Solver',
    help="Commands for solving Equations on Meshes.",
    cli_only=1))

def _solve(menuitem, mesh, solver):
    solverProgressBar = menuitem.getProgressBar("continuous") ## actually ran by the worl
    meshcontext = oof2.engine.mesh.meshes[mesh] ## a who object
```

```
    meshcontext.reserve()
    timedriver = timedrivers.Null()
    try:
        timedriver.apply(solver, meshcontext)
        ## solved flag is set even if solution did NOT
        ## converge in order to show how far the solution
        ## went.
        meshcontext.solved()

    finally:
        meshcontext.cancel_reservation()
solvermenu.addItem(oofmenu.OOFMenuItem(
    'Solve',
    callback=_solve,
    threadable= oofmenu.THREADABLE,
    params=[whoville.WhoParameter('mesh', oof2.engine.mesh.meshes,
                                  tip=parameter.emptyTipString),
            parameter.RegisteredParameter('solver', solverdriver.Driver,
                                          tip=parameter.emptyTipString)
            ],
    help="Solve a problem",
    discussion=xmlmenudump.loadFile('DISCUSSIONS/engine/menu/solve.xml')
    ))
```

The handler *_solve*, does the actual work of the solve command.

In the *solverPage.py*, the menu command is called when the *Solve* button is clicked:

```
class SolverPage(oofGUI.MainPage):
    def __init__(self):
    ...
        self.solveButton = gtk.Button("Solve")
        self.solveButton.connect("clicked", self.solveCB)
    ...
    def solveCB(self, gtkobj):    # "Solve" button callback.
        the_mesh = self.currentFullMeshName()
        solver_selection = self.driverSelectorFactory.get_value()
        OOF.Solver.Solve(mesh=the_mesh, solver=solver_selection)
```

6 Adding a simple page and SWIG extension (02/17/06)

Figure 4 is a simple task page that computes knot invariants for the gnuplot-style coordinate file entered into the rightmost entry box. In *SRC/EXTENSIONS*, create the following file:

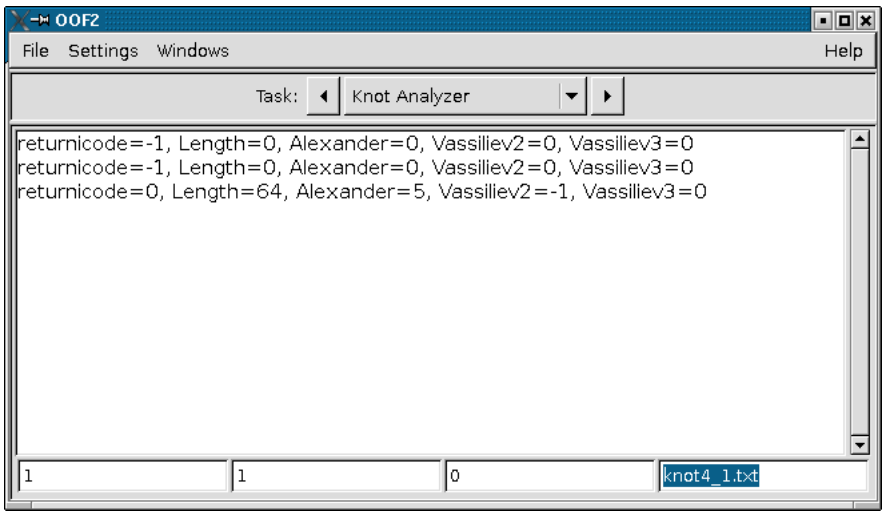

Figure 4: A simple task page.

```
# -*- python -*-
# $RCSfile: KnotPage.py,v $
# $Revision: 1.0 $
# $Author: rlua $
# $Date: 2006/02/16 21:39:31 $

# This software was produced by NIST, an agency of the U.S. government,
# and by statute is not subject to copyright in the United States.
# Recipients of this software assume all responsibilities associated
# with its operation, modification and maintenance. However, to
# facilitate maintenance we ask that before distributing modified
# versions of this software, you first contact the authors at
# oof_manager@ctcms.nist.gov.

from oof2.common.IO.GUI import oofGUI
import gtk
import pango

from oof2.common.IO.words import words

class KnotPage(oofGUI.MainPage):
    def __init__(self):
        oofGUI.MainPage.__init__(self, name="Knot Analyzer", ordering=1000,
                                 tip="Knot Analyzer test page")
        vbox = gtk.VBox()
        self.gtk.add(vbox)
```

```
        scroll = gtk.ScrolledWindow()
        scroll.set_policy(gtk.POLICY_NEVER, gtk.POLICY_ALWAYS)
        vbox.pack_start(scroll, expand=1, fill=1)

        self.textarea = gtk.TextView()
        scroll.set_shadow_type(gtk.SHADOW_IN)
        scroll.add(self.textarea)
        self.textarea.set_editable(0)
        self.textarea.set_cursor_visible(0)
        self.textarea.set_wrap_mode(gtk.WRAP_WORD)
        self.textarea.modify_font(pango.FontDescription("Sans 12"))

        Entrybox = gtk.HBox(homogeneous=1, spacing=2)
        Entrybox.set_border_width(2)
        vbox.pack_start(Entrybox, expand=0, fill=0)

        angles = []
        for i in range(3):
            angles.append(gtk.Entry())
            Entrybox.pack_start(angles[i])
        FileName=gtk.Entry()
        Entrybox.pack_start(FileName)
        FileName.connect("activate",self.FileNameCB,angles)

    def FileNameCB(self,text,angles):
        from oof2.SWIG.EXTENSIONS import KnotAnalyzer
        ka=KnotAnalyzer
        r=ka.KnotProcess(text.get_text(),
float(angles[0].get_text()),float(angles[1].get_text()),float(angles[2].get_text()))
        b=self.textarea.get_buffer()
        t=b.get_text(b.get_start_iter(),b.get_end_iter())
        b.set_text(t+"returncode=%d, Length=%d, Alexander=%d, Vassiliev2=%d, Vassiliev3=
(r,ka.cvar.KnotLength,ka.cvar.Knotdet,ka.cvar.Knotv2,ka.cvar.Knotv3))

KnotPage()
```

This code is patterned after *introPage.py*.

The C++ codes are in the same directory. Modify *DIR.py* to specify the files in the directory for building with OOF2. Any initialization code (executed when OOF2 starts) can be placed in *initialize.py*.

The SWIG file that makes KnotAnalyzer available to python/OOF2 looks like the following: (The %module seems immaterial)

```
%module KnotAnalyzer
%{
extern int KnotProcess(char* szFilename, float Xangle, float Yangle, float Zangle);
```

```
extern int KnotLength;
extern int Knotdet;
extern int Knotv2;
extern int Knotv3;
%}
extern int KnotProcess(char* szFilename, float Xangle, float Yangle, float Zangle);
extern int KnotLength;
extern int Knotdet;
extern int Knotv2;
extern int Knotv3;
```

7 Matrix construction

1. In *femesh.C*:

```
void FEMesh::make_linear_system() {
  Trace("FEMesh::make_linear_system");
  Progress * localbar = getProgress();
  housekeeping();
  // std::cerr << "nodal equations =" << nodaleqn.size() << std::endl;
  // std::cerr << "dofs =" << dof.size() << std::endl;
  /*
    SPATIAL CONTRIBUTIONS START HERE
  */
  if(stiffness) delete stiffness;
  // Computational part of the right-hand-side.
  if(body_force_rhs) delete body_force_rhs;
  stiffness = new StiffnessMatrix(this, nodaleqn.size(), dof.size());
  body_force_rhs = new VECTOR_D(nodaleqn.size(), 0.0);
  /*
    SPATIAL CONTRIBUTIONS END HERE
  */

  /*
    TIME-DEPENDENT CONTRIBUTIONS START HERE
  */
  if(c_matrix)
    {
      delete c_matrix;
    }
  if (kinetics_activated())
    {
      //returns true if ONE equation has kinetics enabled
      c_matrix = new StiffnessMatrix(this, nodaleqn.size(), dof.size());
```

```
    }
  else
    c_matrix = 0;

  if(m_matrix)
    {
      delete m_matrix;
    }
  if(dynamics_activated())
    //returns true if ONE equation has dynamics activated
    m_matrix = new StiffnessMatrix(this, nodaleqn.size(), dof.size());
  else
    m_matrix = 0;

  /*
    TIME-DEPENDENT CONTRIBUTIONS END HERE
  */

  std::string out_mssg;
  for(std::vector<Element*>::size_type i=0;
      i<element.size() && !localbar->query_stop(); i++)
    {
      if (solve_all_state() || element[i]->issubmesh())
{
  // std::cerr << "adding element =" << i << std::endl;
  element[i]->make_linear_system(this, stiffness, c_matrix, m_matrix, *body_force_rh
  localbar->set_progress(float(i+1)/float(element.size()));
  out_mssg = "creating linear system: " + to_string(i+1) + "/" + to_string(element.s
  localbar->set_message(out_mssg);
}
    }
}
...
// Extract the submatrices of the stiffness matrix
// Mapping arrays should be set up by the time you call this.

CSRmatrix FEMesh::Amatrix() const {
//   Trace("FEMesh::Amatrix");
  return stiffness->submatrix(indepeqnmap,freedofmap);
}

CSRmatrix FEMesh::Bmatrix() const {
//   Trace("FEMesh::Bmatrix");
  return stiffness->submatrix(depeqnmap,fixeddofmap);
}
```

```
CSRmatrix FEMesh::Cmatrix() const {
//   Trace("FEMesh::Cmatrix");
  return stiffness->submatrix(indepeqnmap,fixeddofmap);
}

CSRmatrix FEMesh::Cpmatrix() const {
//   Trace("FEMesh::Cpmatrix");
  return stiffness->submatrix(depeqnmap,freedofmap);
}
```

2. Note that the actual work of building the matrix or linear system is passed again to objects called elements. In *element.C*,

```
void Element::make_linear_system(FEMesh *mesh,
 StiffnessMatrix *stiffness,
 StiffnessMatrix *c_matrix,
 StiffnessMatrix *m_matrix,
 VECTOR_D &rhs) const {
//   Trace("Element::make_stiffness for element " + to_string(index_));
  const Material *mat = material();
  if(mat) {
    mat->begin_element(mesh, this);
    mat->make_linear_system(mesh, this, stiffness, c_matrix, m_matrix, rhs);
    mat->end_element(mesh, this);
  }
}
```

3. And in *material.C*,

```
void Material::make_linear_system(FEMesh *mesh,
  const Element *el,
  StiffnessMatrix *stiffness,
  StiffnessMatrix *c_matrix,
  StiffnessMatrix *m_matrix,
  VECTOR_D &rhs) const
{
  //std::cerr << "Material::make_stiffness" << std::endl;
  // loop over all fluxes
  //   Trace("Material::make_stiffness.");
  for(std::vector<FluxPropList>::size_type fluxi=0; fluxi<fluxprop.size();
      fluxi++)
    {
      Flux *flux = Flux::allfluxes()[fluxi];
      // fluxproplist is the list of Properties that contribute to this Flux
      const FluxPropList &fluxproplist = fluxprop[fluxi];
```

```
      // A Flux is active if it is used in one or more active Equations.
      // The fluxproplist is active if any of the Properties that
      // contribute to it are active.  A Property is active if all of
      // the Fields it uses are defined.
      if(mesh->is_active_flux(*flux) && fluxproplist.active(mesh)) {

// Find the integration order for this flux.  Use the maximum
// order required by the relevant Properties.  This could be
// Element dependent, so it can't be precomputed.
int order = 0;
for(FluxPropList::size_type propi=0; propi<fluxproplist.size(); propi++)
  {
    int ord = fluxproplist[propi]->integration_order(mesh, el);
    if(ord > order) order = ord;
  }
order += flux->eqn_integration_order(mesh, el);


// Integrate over the Element
for(GaussPointIterator gpt=el->integrator(order); !gpt.end(); ++gpt) {
  // Trace("Gausspoint " + to_string(gpt.gausspoint()));
  FluxData *fluxdata = flux->initializeData(el);

  // Loop over Properties contributing to this Flux,
  // accumulating their contributions in the flux matrix. First,
  // get the element's funcnode-iterator and pass it in. Do the
  // iteration here, though, to ensure that each property in the
  // fluxproplist gets a crack at each node.

  for(FluxPropList::size_type propi=0; propi<fluxproplist.size();
      propi++)
    {
      Property *prop = fluxproplist[propi];
      if(prop->is_active()) {
//std::cerr << "Material::setting-up flux matrix" << std::endl;
for(ElementFuncNodeIterator eni = el->funcnode_iterator();
    !eni.end(); ++eni) {
  // A flux matrix embodies a numerical representation
  // relating the nodal fluxes with its fields.

  // For elastic systems:
  // For the in-plane components, it is the "gradient", the
  // strain operator, \times the material stiffness,
  // or equivalently the stiffness \times the nodal derivative
  // of the shape function contributions.
  // For the out of plane components (plane stress),
```

```
  // it corresponds to the
  // identity operator, or equivalently, to the stiffness tensor
  // \times the nodal shape function.
  // (NOT the derivative)

  // For non-elastic systems:
  // For in-plane components, it is the "gradient"
  // operator \times the material parameter (e.g., thermal
  // conductivity, mass diffusivity, etc.), or equivalently
  // a proportionality constant \times the derivative of the
  // shape function.
  // For the out of plane components (plane flux) it corresponds
  // to a proportionality constant times the shape function
  // (NOT the derivative)
  fluxproplist[propi]->fluxmatrix(mesh, el, eni, flux, fluxdata,
  gpt.gausspoint());
}
// For the RHS contributions, don't want to iterate over nodes.
fluxproplist[propi]->fluxrhs(mesh, el, flux, fluxdata,
     gpt.gausspoint());
//fluxrhs embodies the body forces for each driving force
//for mechanical systems could be gravity \times density,
//For thermal systems, chemical reactions.
      } //if "property is active" ends here
    } //loop over all properties ends here

  // Have the Flux make its contribution to the stiffness matrix
  flux->make_stiffness(mesh, el, gpt.gausspoint(), fluxdata,
       *stiffness);
  // ...and rhs.
  flux->make_rhs(mesh, el, gpt.gausspoint(), fluxdata, rhs);

  delete fluxdata;
} //loop over gauss points ends here
      } //loop over all fluxproplist ends here
    } //loop over all flux prop ends here

  /*
    TIME DERIVATIVES START HERE
  */
  if (mesh->dynamics_activated() || mesh->kinetics_activated())
    set_time_derivatives(mesh, el, c_matrix, m_matrix);
  //time-dependent properties should be inserted at THIS level
  /*
    TIME DERIVATIVES END HERE
  */
```

```
}// end of member function
```

4. ... in *flux.C*,

```
void Flux::make_stiffness(FEMesh *mesh, const Element *el,
  const GaussPoint &gpt,
  const FluxData *fluxdata,
  StiffnessMatrix &stiffness)
{
  // Loop over all active Equations that use this Flux
  for(std::vector<Equation*>::size_type i=0; i<eqnlist.size(); i++) {
    Equation *eqn = eqnlist[i];
    if(mesh->is_active_equation(*eqn))
      eqn->make_stiffness(mesh, el, gpt, *this, fluxdata, stiffness);
  }
}
```

5. .. in *equation.C*, for plane flux (there is also one for divergence),

```
void PlaneFluxEquation::make_stiffness(FEMesh *mesh, const Element *el,
       const GaussPoint &gpt,
       const Flux &flux,
       const FluxData *fluxdata,
       StiffnessMatrix &stiffness) const
{

  // loop over nodes
  for(ElementFuncNodeIterator node=el->funcnode_iterator();
      !node.end();++node)
    {

      double shapefun = node.shapefunction(gpt);
      // loop over components of equation (ie,  of out-of-plane flux)
      int eqcomp = 0;
      for(IteratorP fluxindex=flux.iterator(OUT_OF_PLANE);
  eqcomp<dim() && !fluxindex.end(); ++eqcomp, ++fluxindex) {

//   std::cerr << "Iterating over flux components." << std::endl;
//    std::cerr << "Flux is " <<  flux.name() << std::endl;
//     const FieldIterator *fldi = fluxindex;
//     const SymTensorIndex *ipsti =
//       dynamic_cast<const SymTensorIndex*>(fldi);

//     std::cerr << "Current flux row is " << ipsti->row() << std::endl;
//     std::cerr << "Current flux col is " << ipsti->col() << std::endl;
```

```
  // row in global matrix/
  // int rowno = nodaleqn(mesh, *node.funcnode(), eqcomp)->index(); //original unsym
// std::cerr << "number of nodal equations=" << node.funcnode()->eqnlist.size() << s
int rowno = mesh->map_forward_index(*this,
    *node.funcnode(),
    eqcomp); //symmetric row component
// This row is an out-of-plane equation.
stiffness.rowgroup[rowno] = node.node()->index();
//std::cerr << "node.node()->index()=" << node.node()->index()<< std::endl;
// loop over degrees of freedom of the element that couple to
// this flux, by looping over all components of the relevant
// *defined* fields at all the nodes.  The out-of-plane fields
// are included in the field list.
const std::vector<const Field*> &fieldlist =
  el->material()->fieldlist(flux);
for(std::vector<const Field*>::size_type f=0; f<fieldlist.size(); f++) {
  const Field *field = fieldlist[f];
  if(mesh->is_defined_field(*field)) {
    // loop over components of active Field at each Node
    for(ElementFuncNodeIterator nj=el->funcnode_iterator();
!nj.end(); ++nj)
      {
for(IteratorP fi=field->iterator(ALL_INDICES);
    !fi.end(); ++fi) {
  DegreeOfFreedom *dof = (*field)(mesh, nj, fi);

  double wgt = gpt.weight();
  double raw = fluxdata->matrix_element(mesh, fluxindex, field,
fi, nj) * shapefun;
  double res = raw * wgt;

//    Trace("PlaneFluxEquation adding " + to_string(res) +
//  " to stiffness matrix (" + to_string(rowno) + ", " +
//  to_string(dof->dofindex()) + "),   Integrand is " +
//  to_string(raw) + ", weight is " + to_string(wgt));

  stiffness(rowno, dof->dofindex()) += res;

  // Mark this column as either in-plane or part of a group.
  stiffness.colgroup[dof->dofindex()] =
    (field->in_plane_part() ? -1 : nj.node()->index() );

//                  stiffness(rowno, dof->dofindex()) +=
//      flux.matrix_element(mesh, fluxindex, field, fi, nj) *
//      shapefun * gpt.weight();
}
```

```
      }
  }
}
      }
    }
}
```

6. Back in *femesh.C*,

```
// Extract the submatrices of the stiffness matrix
// The conceptual subdivision is this:
//
//                 free      fixed
//                          |
//   independent      A     | C
//                          |
//                ----------+---     ---
//     dependent      C'    | B
//


// New version.  May be broken up into steps later, but for
// now duplicates the old functionality with the new calling
// semantics.
//

// Rowmapper and Colmapper.
// Build a map of the rows i of the stiffness matrix for
// which f(i) is true, and return it.  Call with f(i)
// discriminating between independent (known-force) vs.
// dependent (unknown-force) equations for rows, or
// with f(i) discriminating between fixed and free
// DOFs for columns.
//
// Do this by manipulating the pre-existing mapping list.
void FEMesh::rcmapper(bool (FEMesh::*f)(int) const,
      std::vector<int>& rcmap, int len) const {
  //  Trace("FEMesh::rcmapper");
  //  vector<int> rowmap( nrows(), -1);
  // Double-check the size?
  int nrc = 0; // # of rows or columns in the mapped submatrix
  rcmap.resize(len);
  for(int i=0; i<len; i++) {
    if ((this->*f)(i)) {
      rcmap[i] = nrc++;
    }
```

```
    else {
      rcmap[i] = -1; // -1 is default for rows/columns which don't occur.
    }
  }
}

// Run the maps, creating the lists used to partition the stiffness
// matrix.

void FEMesh::mapdofeqs() {
  Trace("FEMesh::mapdofeqs");
  rcmapper(&FEMesh::solve_this_nodal_eqn, indepeqnmap, stiffness->nrows());
  // cerr << "  indepeqnmap= "; print(indepeqnmap, cerr); cerr << endl;
  rcmapper(&FEMesh::dont_solve_this_nodal_eqn, depeqnmap, stiffness->nrows());
  // cerr << "    depeqnmap= "; print(depeqnmap, cerr); cerr << endl;
  rcmapper(&FEMesh::solve_for_this_dof, freedofmap, stiffness->ncols());
  // cerr << "   freedofmap= "; print(freedofmap, cerr); cerr << endl;
  rcmapper(&FEMesh::dont_solve_for_this_dof, fixeddofmap, stiffness->ncols());
  // cerr << "  fixeddofmap= "; print(fixeddofmap, cerr); cerr << endl;
}
```

8 Sequence of events after clicking 'Solve'

Short version (preceded by some undocumented steps):

1. In *solvermenu.py*:

```
def _solve(menuitem, mesh, solver):
    import pdb
    pdb.set_trace()
    solverProgressBar = menuitem.getProgressBar("continuous") ## actually ran by
    meshcontext = oof2.engine.mesh.meshes[mesh] ## a who object
    meshcontext.reserve()
    timedriver = timedrivers.Null()
    try:
        timedriver.apply(solver, meshcontext)
        ## solved flag is set even if solution did NOT
        ## converge in order to show how far the solution
        ## went.
        meshcontext.solved()

    finally:
        meshcontext.cancel_reservation()
```

2. In *timedrivers.py*, when *timedriver.apply* is called:

```
class Null(TimeDriver):
    ...
    def apply(self, solverdriver, meshcontext, verbosity = 1,
              report_success = 1):
        solverdriver.apply(self.steppermethod, meshcontext,
                           verbosity = verbosity)
        self.clean_up(solverdriver, meshcontext, report_success)
        self.redraw()
```

3. In *solverdriver.py/spy*, when *solverdriver.apply* is called:

```
class Driver(registeredclass.RegisteredClass):
    ...
    def apply(self, stepper, meshcontext, verbosity = 1):
        self.mesh = meshcontext
        self.target_time = self.mesh.getObject().current_time() + \
                           stepper.get_dt()
        self.driver_apply(stepper, verbosity)
```

4. In the same file, when *self.driver_apply* is called:

```
class LinearDriver(Driver):
    ...
    def driver_apply(self, stepper, verbosity):
        self.core = SolverDriverCore(stepper, self.method, self.mesh.getObject())
        self.precompute(verbosity)
        ## precompute assigns properties, creates the mapping to
        ## symmetrizes the linear system of equations,
        ## builds the master stiffness, conductivity, and mass
        ## matrices, sets BCs
        self.apply_solver(verbosity) ## applies the linear solver. Solution is accep
        self.postcompute (verbosity)
        ## postcompute resets BCs and fixates histo
```

5. In the same file, when *self.precompute* is called (code not list indicated by ...):

```
class Driver(registeredclass.RegisteredClass):
    ...
    def precompute(self, verbosity = 1):

        ...

        ########################################
        ## MASTER STIFFNESS MATRIX IS CONSTRUCTED HERE
        ########################################
```

```
        self.mesh.make_linear_system()
        ## makes stiffness matrix and rhs
        ## If equations and fields are propertly defined
        ## each material parameter successfully contributes
        ## to the stiffness matrix and RHS
        ########################################
        ## END OF STIFFNESS MATRIX CONSTRUCTION
        ########################################

        ...

        ########################################
        ## ACTUALLY BUILD THE STIFFNESS MATRIX
        ########################################
        ## The next line is the very last thing that must be done
        ## before actually solving the mesh.  It allocates memory and
        ## builds the actual A-matrix, b-vector, x-vector to solve the
        ## system Ax=b
        self.core.precompute()
```

6. In the same file, when *self.apply_solver* is called (from step 4):

```
class Driver(registeredclass.RegisteredClass):
    ...
    def apply_solver(self, verbosity = 1):
        eq_Progress = progressbar.getProgress()
        eq_Progress.reset()    # this may not be the right place to do this
        if eq_Progress.query_stop():
            return
        if verbosity:
            reporter.start_progressbar()
        self.status = self.core.solve()
        if verbosity:
            reporter.progressbar_completed()
            reporter.end_progressbar()
```

7. Several calls - such as *SolverDriverCore* (in step 4), *self.core.precompute* (in step 5), and *self.core.solve* (in step 6) - are already at the interface of Python and C++. The implementations (in file *solverdriver.C*) that are SWIGed are as follows:

```
SolverDriverCore::SolverDriverCore(SteppingScheme *stepsch,
   LinearSolver *solver, FEMesh *femesh)
  : solver(solver),
    femesh(femesh),
    sscheme(stepsch),
```

```
    K(0),
    b(0),
    nrowsA(0),
    ncolsA(0),
    A(0),
    C(0),
    x(0),
    rhs(0),
    xp(0)
{
}

void SolverDriverCore::precompute() {
  //std::cerr << "SolverDriverCore::precomputing"  << std::endl;
  //at this point the solver and the preconditioner are already selected
  // at the UI and passed as arguments into the solver
  RWLock * rwlock = femesh->get_rwlock();
  rwlock->read_acquire(); //acquire read lock
  //std::cerr << "after the lock" << std::endl;
  try{
    int n, m;
    if (A) {
      delete A;
    }
    A = new CSRmatrix (femesh->Amatrix());

    nrowsA = n = A->nrows();

//     std::cerr << "SolverDriverCore::precompute: " << A->nrows() << "x" << A->ncol
    ncolsA = A->ncols();
    if(n != A->ncols())
      throw ErrSetupError("Stiffness matrix is not square! nrows="
  + to_string(n) + ", ncols="
  + to_string(A->ncols()));

    if (C) {
      delete C;
    }
    C = new CSRmatrix (femesh->Cmatrix());
    m = C->ncols();
    if (!x) {
      x = new VECTOR_D (n, 0.0);
    }
    x->clear();
    femesh->get_unknowns(*x, femesh->freedofmap);
    //std::cerr << "x=" << *x << std::endl;
```

```
    if (!rhs)
    {
      ////std::cerr << "allocating memory for rhs" << std::endl;
      rhs = new VECTOR_D (n, 0.0);
    }
    rhs->clear();
    femesh->get_rhs(*rhs, femesh->indepeqnmap);

    if (!xp) {
      //std::cerr << "allocating memory for xp" << std::endl;
      xp = new VECTOR_D (m, 0.0);
    }
    xp->clear();
    //std::cerr << "allocating memory for xp" << std::endl;
    femesh->get_unknowns(*xp, femesh->fixeddofmap);
    *rhs -= (*C) * (*xp);


  } //end of try block
  catch(...) //guarantee that lock will be released if ANY exception is raised
    {
      //std::cerr << "releasing the lock in catch" << std::endl;
      rwlock->read_release();
      throw;
    }
  //std::cerr << "releasing the lock while exiting the function " << std::endl;
  rwlock->read_release();

  //stepper interactions
  sscheme->precompute(femesh, A, rhs, x);//these lines should be outside the locked

  // Get the K matrix and b vector of the linear system, and store it.
  // We don't actually own these matrices, but they're stored as
  // object attributes so they're common to precompute, solve, and
  // postcompute.
  K = sscheme->get_K(femesh, A);
  b = sscheme->get_b(femesh, A, rhs);

}

bool SolverDriverCore::solve(){
  //std::cerr << "entering solverStepperCore::solve" << std::endl;
  solver->set_blocking(femesh);

  solver->precompute(*K);
  //std::cerr << "K = "<< *K << std::endl;
```

```
    bool status = solver->solve(*K, *b, *x);

    solver->postcompute(); // Call possibly-null post-compute hook.
    //postcompute();
    //std::cerr << "leaving solverStepperCore::solve" << std::endl;
    return status;

  }
```

Note that objects that do stepping (stepping scheme or stepper), the solver (method), and femesh, are passed to *SolverDriverCore* and figure prominently in the calculations (both in Python and C++).

The following is a longer version (pdb debugger). Long path names have been truncated:

```
oof2 --script testsolve
--Return--
> /usr/lib/python2.3/pdb.py(992)set_trace()->None
-> Pdb().set_trace()
(Pdb) n
> /oof2/engine/IO/solvermenu.py(83)_solve()
-> solverProgressBar = menuitem.getProgressBar("continuous") ## actually ran bythe worke
(Pdb)
> /oof2/engine/IO/solvermenu.py(84)_solve()
-> meshcontext = oof2.engine.mesh.meshes[mesh] ## a who object
(Pdb)
> /oof2/engine/IO/solvermenu.py(85)_solve()
-> meshcontext.reserve()
(Pdb)
> /oof2/engine/IO/solvermenu.py(86)_solve()
-> timedriver = timedrivers.Null()
(Pdb)
> /oof2/engine/IO/solvermenu.py(87)_solve()
-> try:
(Pdb)
> /oof2/engine/IO/solvermenu.py(88)_solve()
-> timedriver.apply(solver, meshcontext)
(Pdb) s
--Call--
> /oof2/engine/timedrivers.py(38)apply()
-> def apply(self, solverdriver, meshcontext, verbosity = 1,
(Pdb) n
> /oof2/engine/timedrivers.py(40)apply()
-> solverdriver.apply(self.steppermethod, meshcontext,
```

```
(Pdb) s
> /oof2/engine/timedrivers.py(41)apply()
-> verbosity = verbosity)
(Pdb)
--Call--
> /oof2/SWIG/engine/solverdriver.py(207)apply()
-> def apply(self, stepper, meshcontext, verbosity = 1):
(Pdb) n
> /oof2/SWIG/engine/solverdriver.py(208)apply()
-> self.mesh = meshcontext
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(209)apply()
-> self.target_time = self.mesh.getObject().current_time() + \
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(211)apply()
-> self.driver_apply(stepper, verbosity)
(Pdb) s
--Call--
> /oof2/SWIG/engine/solverdriver.py(481)driver_apply()
-> def driver_apply(self, stepper, verbosity):
(Pdb) n
> /oof2/SWIG/engine/solverdriver.py(482)driver_apply()
-> self.core = SolverDriverCore(stepper, self.method, self.mesh.getObject())
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(483)driver_apply()
-> self.precompute(verbosity)
(Pdb) s
--Call--
> /oof2/SWIG/engine/solverdriver.py(213)precompute()
-> def precompute(self, verbosity = 1):
(Pdb) n
> /oof2/SWIG/engine/solverdriver.py(228)precompute()
-> badmatls = self.mesh.check_materials()
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(229)precompute()
-> if badmatls:
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(233)precompute()
-> eq_Progress = progressbar.getProgress()
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(246)precompute()
-> self.mesh.getObject().precompute()
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(249)precompute()
-> if verbosity:
(Pdb)
```

```
> /oof2/SWIG/engine/solverdriver.py(250)precompute()
-> reporter.start_progressbar()
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(260)precompute()
-> if self.mesh.get_symmetry_state()==\
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(264)precompute()
-> conjugate.listofconjugatepairs.find_equation_mapping(
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(271)precompute()
-> self.mesh.make_linear_system()
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(279)precompute()
-> if verbosity:
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(280)precompute()
-> reporter.progressbar_completed()
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(281)precompute()
-> reporter.end_progressbar()
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(284)precompute()
-> if eq_Progress.query_stop() !=0:
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(299)precompute()
-> mtx = self.mesh.getObject().stiffness_matrix()
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(300)precompute()
-> self.mesh.getObject().boundary_rhs_init( mtx.nrows() )
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(304)precompute()
-> self.mesh.getObject().resetdofeqs()
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(336)precompute()
-> while self.mesh.getObject().bcPromote(self.target_time):
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(341)precompute()
-> self.mesh.getObject().fixedBCConsistency(self.target_time)
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(342)precompute()
-> if debug.debug() and verbosity:
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(345)precompute()
-> self.mesh.getObject().floatBCResolve(self.target_time)
(Pdb)
> oof2/SWIG/engine/solverdriver.py(346)precompute()
```

```
-> if debug.debug() and verbosity:
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(349)precompute()
-> self.mesh.getObject().invoke_flux_bcs(self.target_time)
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(350)precompute()
-> if debug.debug() and verbosity:
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(352)precompute()
-> self.mesh.getObject().invoke_fixed_bcs(self.target_time)
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(353)precompute()
-> if debug.debug() and verbosity:
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(355)precompute()
-> self.mesh.getObject().invoke_force_bcs(self.target_time)
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(356)precompute()
-> if debug.debug() and verbosity:
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(359)precompute()
-> self.mesh.getObject().mapdofeqs()      # must come after fixed bcs, before floating o
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(362)precompute()
-> if debug.debug() and verbosity:
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(364)precompute()
-> self.mesh.getObject().invoke_float_bcs(self.target_time)
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(365)precompute()
-> if debug.debug() and verbosity:
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(379)precompute()
-> self.core.precompute()
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(382)precompute()
-> if verbosity:
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(383)precompute()
-> rows_A_matrix = self.core.nrows()
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(384)precompute()
-> cols_A_matrix = self.core.ncols()
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(385)precompute()
-> reporter.report("Made %dx%d stiffness matrix" % (rows_A_matrix,
```

```
(Pdb)
--Return--
> /oof2/SWIG/engine/solverdriver.py(385)precompute()->None
-> reporter.report("Made %dx%d stiffness matrix" % (rows_A_matrix,
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(488)driver_apply()
-> self.apply_solver(verbosity) ## applies the linear solver. Solution is accepted HERE
(Pdb)
> /oof2/SWIG/engine/solverdriver.py(489)driver_apply()
-> self.postcompute (verbosity)
(Pdb)
--Return--
> /oof2/SWIG/engine/solverdriver.py(489)driver_apply()->None
-> self.postcompute (verbosity)
(Pdb)
--Return--
> /oof2/SWIG/engine/solverdriver.py(211)apply()->None
-> self.driver_apply(stepper, verbosity)
(Pdb)
> /oof2/engine/timedrivers.py(42)apply()
-> self.clean_up(solverdriver, meshcontext, report_success)
(Pdb)
> /oof2/engine/timedrivers.py(43)apply()
-> self.redraw()
(Pdb)
--Return--
> /oof2/engine/timedrivers.py(43)apply()->None
-> self.redraw()
(Pdb)
> /oof2/engine/IO/solvermenu.py(92)\_solve()
-> meshcontext.solved()
(Pdb)
> /oof2/engine/IO/solvermenu.py(95)\_solve()
-> meshcontext.cancel_reservation()
(Pdb)
--Return--
> /oof2/engine/IO/solvermenu.py(95)\_solve()->None
-> meshcontext.cancel_reservation()
(Pdb)
> /oof2/common/worker.py(183)run()
-> self.finalize()
...
```

9 Generating Abaqus output

Consult the two methods

```
WriteABAQUSfromSkeleton
WriteABAQUSfromMesh
```

respectively defined in files *skeletonIO.py* and *meshIO.py*. Example graphics outputs are given in figures 5 and 6.

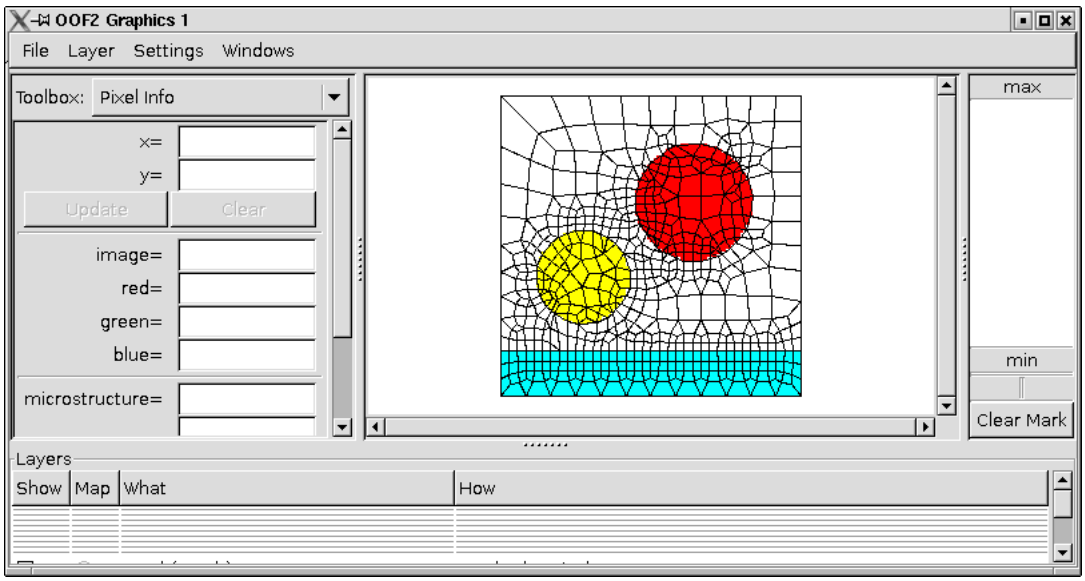

Figure 5: OOF2 graphics window output for two_circles.skeleton

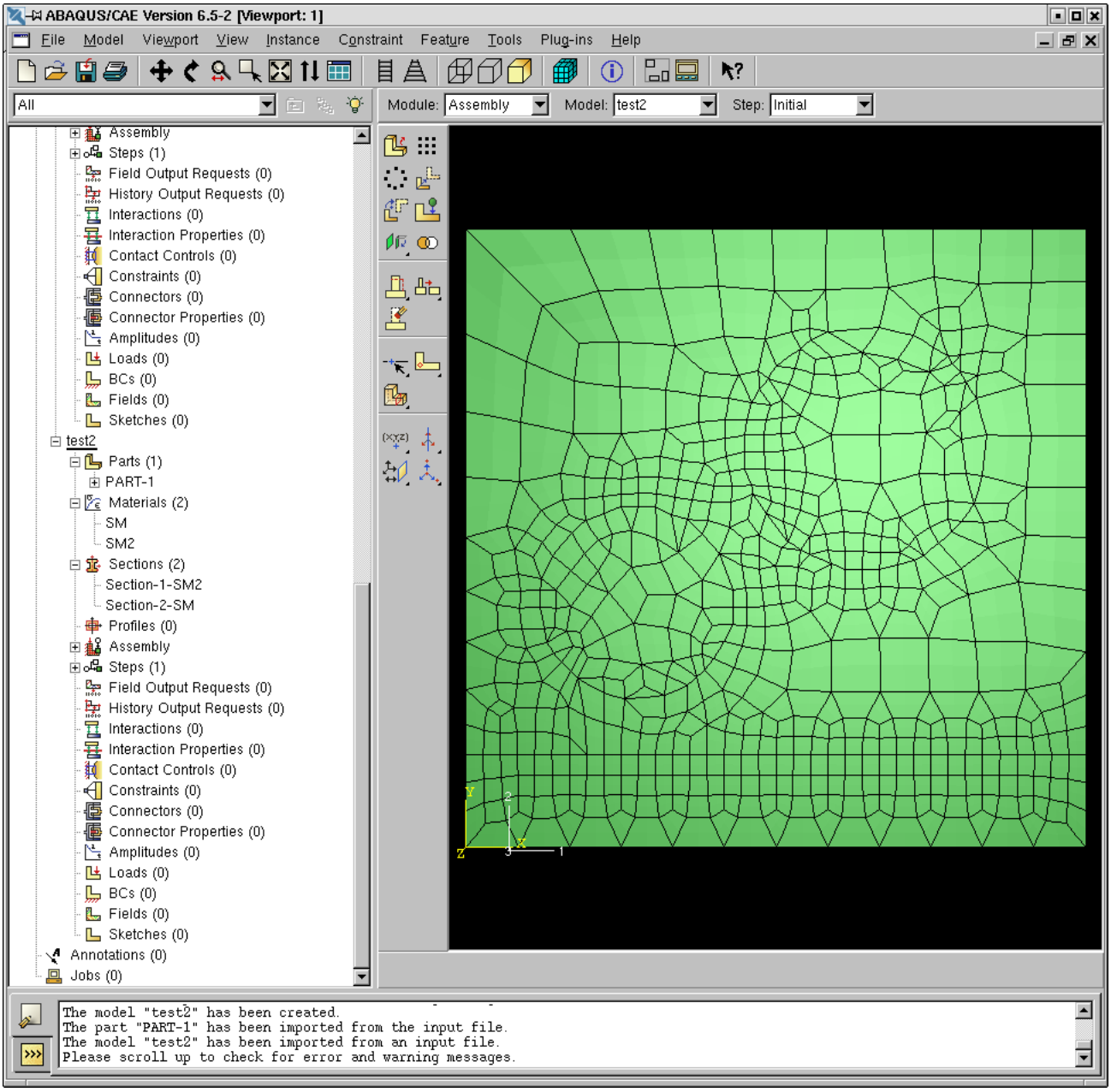

Figure 6: Abaqus graphics output after importing a model (.inp file) for two_circles.skeleton generated from a skeleton or mesh information in OOF2.