Design of Finite Element Software for Modeling Bone Deformation and Failure

D. Bindel

Department of Computer Science
Cornell University

22 Apr 2010
Goal state

What I hope to get across today:

- Show some design choices that help make flexible FE software
  - In high-level solvers and mesh specification
  - In preconditioner construction
  - In element coding (briefly)
- Show some places where I need help from domain experts
Diagnostic toolchain

- Micro-CT (or other) scan data from patient
- Inference of material properties
- Construction of coarse FE model (voxels)
- Simulation under loading
- Output of stress fields, displacements, etc.
Software strategies

Two basic routes:

▶ Discretize microstructure to get giant FE model
  ▶ Prometheus (Mark Adams) – 57M+ elements
  ▶ ParFE (Arbenz and Sala) – 200M unknowns

▶ Approximate microstructure with constitutive model
  ▶ Can do with commercial FEM codes
  ▶ Smaller model, less compute time
  ▶ Less detail required in input?
  ▶ Hard to get the right constitutive model
A little history

BoneFEA started as a consulting gig

- Code for ON Diagnostics (Keaveny and Kopperdahl)
- Developed jointly with P. Papadopoulos
- Meant to replace ABAQUS in overall system
- Initial goal: some basic simulations in under half an hour
- Development work on and off 2006–2008
- More recent revisitings (trying to rebuild)
BoneFEA

- Standard displacement-based finite element code
- Elastic and plastic material models (including anisotropy and asymmetric yield surfaces)
- High-level: incremental load control loop, Newton-Krylov solvers with line search for nonlinear systems
- Library of (fairly simple) preconditioners; default is a two-level geometric multigrid preconditioner
- Input routines read ABAQUS decks (and native format)
- Output routines write requested mesh and element quantities
- Visualization routines write VTK files for use with VisIt
Basic principles

- This sort of programming seems hard (?)
  - How many man-hours went into ABAQUS?
  - Easy to lose sleep to an indexing error
- Want to reduce the *accidental* complexity
  - Express as much as possible at a high level
  - Use C++/Fortran (and libraries) for performance-critical stuff
  - Make trying new things out easy
Enabling technology

Three separate language-based tools:

- Lua-based system for loading conditions, high-level solvers
- Lua-based system for preconditioners, lower-level solver logic
- Matexpr for material model computations

In progress: solver scripting via PyTrilinos (Sandia)
Solver quandries

A simple simulation involves \textit{lots} of choices:

- Load stepping strategy?
- Nonlinear solver strategy?
- Linear solver strategy?
- Preconditioner?
- Subsolvers in multilevel preconditioner?

Want a simple framework for playing with options.
Example analyses

DB: femur.vtk
Example analysis loop

```lua
mesh:rigid(mesh:numnp()-1, {z='min'}, function()
    return 'uuuuuuu', 0, 0, bound_disp
end)

pc = simple_msm_pc(mesh,20)
mesh:set_cg{M=pc, tol=1e-6, max_iter=1000}
for j=1,n do
    bound_disp = 0.2*j
    mesh:step()
    mesh:newton{max_iter=6, Rtol=1e-4}
end
```
Analysis innards

- **rigid** ties a specified part of the mesh to a rigid body (and applies boundary conditions to that rigid body)
- **step** swaps history, updates load, computes predictor
- **newton** does Newton iteration with line search; specify
  - Max iterations
  - Residual tolerance
  - Line search parameters (Armijo constant $\alpha$)
  - What linear solver to use
  - Whether to update the preconditioner
- **Also have mnewton** (modified Newton)
Preconditioning

- Accelerate iterative solver with *preconditioner*
- Often built from simpler blocks
  - Basic iterative solver passes
  - Block solves
  - Coarse grid solves
- Want a simple way to assemble these blocks
function simple_msm_pc(mesh, ncgrid, nsmooth, omega)
  local pcc = form_coarse_pc2(mesh, ncgrid)
  local pc = {}
  local K = mesh.K
  nsmooth = nsmooth or 1
  function pc:solve(x,b) ... end
  function pc:update() pcc:update() end
  function pc:delete() ... end
  return pc
end
function pc:solve(x,b)
    self.r = self.r or QArray:new(x:m(),1)
    self.dx = self.dx or QArray:new(x:m(),1)

    mesh_bgs(mesh.mesh,mesh.K,x,b,nsMOOTH)
    K:apply(x,self.r)
    self.r:sub(b)

    pcc:solve(self.dx,self.r)
    x:sub(self.dx)
    K:apply(x,self.r)
    self.r:sub(b)

    mesh_bgs(mesh.mesh,mesh.K,self.dx,self.r,nsMOOTH)
    x:sub(self.dx)
end
The problem of preconditioning

Standard preconditioners work best for

- Simple geometries
- Constant or smoothly varying coefficients
- Isotropic materials
- Strongly definite problems

Macroscopically, bone breaks almost all of these!
Preconditioning triumphs and failures

![Graph showing relative residual vs. step number for different steps (Step 1, Step 2, Step 3). The x-axis represents the step number (k), and the y-axis represents the relative residual. The graph illustrates the convergence of the residual for each step, with Step 1 converging the fastest, followed by Step 2, and Step 3 showing the slowest convergence.]
We do pretty well with two-level geometric multigrid

- 18 steps, 15 s to solve femur model on my laptop

... up until plasticity starts to kick in

Needed: a better (physics-based) preconditioner

Usual key: physical insight into macroscopic behavior
Material modeling

BoneFEA provides general plastic element framework; specific material model provided by an object. Built-in:

- Isotropic elastic
- Orthotropic elastic
- Simple plastic
- Anisotropic elastic / isotropic plastic
- Isotropic elastic / asymmetric plastic yield surface

How do we make it simplify to code more?
Partial solution: Matexpr

- Relatively straightforward in MATLAB — but slow
- Use Matexpr to translate MATLAB-like code to C
- Supports basic matrix expressions, symbolic differentiation, function definitions.
- Takes advantage of symmetry, sparsity, redundancy to optimize generated code
- Does not provide control flow (that’s left to C)
Matexpr in action

Extract the deviatoric part of the elastic constitutive tensor:

```c++
void ME::compute_Cd(double* Cd)
{
    /* <generator matexpr>
       input symmetric DGelastic(9,9);
       output Cd(9,9);
       m = [1; 1; 1; 0; 0; 0; 0; 0; 0];
       Iv = m*m'/3.0;
       Id = eye(9) - Iv;
       Cd = Id*DGelastic*Id;
    */
}
```
Conclusion

- Initial BoneFEA work for ON Diagnostics is done.
- Currently re-implementing similar functionality in an open package (as part of a more general framework).
- Problems and physical insights both welcome!