Object-Oriented Finite Element Analysis of Material Microstructures

Stephen A. Langer
Mathematical and Computational Sciences Division
Information Technology Laboratory
National Institute of Standards and Technology
## Personnel

<table>
<thead>
<tr>
<th>Name</th>
<th>Affiliation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steve Langer</td>
<td>NIST ITL MCSD</td>
</tr>
<tr>
<td>Andrew Reid*</td>
<td>Drexel University (ITL $)</td>
</tr>
<tr>
<td>Seung-Ill Haan*</td>
<td>U. Md, Baltimore County (CTCMS $)</td>
</tr>
<tr>
<td>Edwin Garcia*</td>
<td>Penn State University (NSF &amp; CTCMS $)</td>
</tr>
<tr>
<td>Andrew Roosen</td>
<td>NIST MSEL</td>
</tr>
<tr>
<td>Eric Ma</td>
<td>Montgomery Blair High School</td>
</tr>
<tr>
<td>Kevin Chang</td>
<td>Math &amp; Science Magnet Program</td>
</tr>
<tr>
<td>Kang-Xing Jin</td>
<td></td>
</tr>
<tr>
<td>Daniel Vlacich</td>
<td></td>
</tr>
<tr>
<td>Kyle Stemen</td>
<td>SURF, Kent State University</td>
</tr>
<tr>
<td>Edwin Fuller</td>
<td>NIST MSEL</td>
</tr>
<tr>
<td>W. Craig Carter</td>
<td>MIT</td>
</tr>
<tr>
<td>Zi-Kui Liu</td>
<td>PSU</td>
</tr>
<tr>
<td>Panos Charalambides</td>
<td>UMBC</td>
</tr>
</tbody>
</table>

* Guest Researchers at the NIST Center for Theoretical and Computational Materials Science
Outline of the talk

- What?
- Examples
- OOF2
  - Why?
  - How?
    - Design Goals
    - Ingredients
What is OOF?

1. Start with a micrograph

2. Assign material properties

3. Perform virtual experiments

4. Visualize and quantify
Why OOF?

- Commercial finite element packages work best with large scale systems with regularly shaped components.
Why OOF?

- Commercial finite element packages work best with large scale systems with regularly shaped components.
- Materials systems are small scale and disordered.
Why OOF?

- Commercial finite element packages work best with large scale systems with regularly shaped components.
- Materials systems are small scale and disordered.
- OOF is designed to answer the questions that materials scientists want to ask.
- OOF is easy to use.
CONCEPTUAL ORGANIZATION

Simulation    Experiment

Microstructure Data (Micrographs)  Fundamental Materials Data  Materials Physics

Object Model Isomorphic to the Microstructure

Finite Element Solver

Virtual Parametric Experiments  Visualization of Microstructural Physics  Effective Macroscopic Properties

Easy-to-use Graphical User Interface
Example Applications:

- Thermal Barrier Coatings
- Residual Stresses in Alumina
- Marble
- Piezoelectrics
- Batteries
Predict Thermal Conductivity $\kappa$ of Ceramic Thermal Barrier Coatings for Turbine Blades

with James Ruud, NS Hari, James Grande, and Antonio Mogro-Campero, GE Corporate R&D
Funded in part by DOE Advanced Turbine Systems Program

- TBC’s allow jet engine blades to operate at higher temperatures.
- Physical measurements of $\kappa$ are difficult, time consuming and expensive. Hardly ever done during quality control.
- OOF could replace measurements during research, development, design, and production.
Residual Stresses in Alumina

Stress invariant 1 \( (\sigma_{11} + \sigma_{22}) \) shown for \( \Delta T = -1500°C \).
Calculations under plane stress and free boundary conditions.
Total number of elements = 117612.

Venkata Vedula, Sandia
Thermal Degradation of Decorative Marbles

bowing of façade claddings (library, Universität Göttingen)

granular disintegration

original

after 6 years

Thomas Weiß and Siegfried Siegesmund, Universität Göttingen, Germany
Microstructural Effects in Polycrystalline Piezoelectrics
Edwin Garcia

Microstructural Design of Rechargeable Batteries

250 µm  52 µm  174 µm

R. Edwin Garcia, Catherine M. Bishop, W. Craig Carter*, Stephen A. Langer†
Pimpa Limthongkul, and Yet-Ming Chiang

* Massachusetts Institute of Technology
† National Institute of Standards and Technology
Thermal OOF1
(elasticity & thermal diffusion)

Classic OOF1 (elasticity)

Why OOF2?

Electrochemical OOF1
(time dependent, nonlinear)

Li concentration in a Li ion battery

Electromechanical OOF1
(adaptive refinement, nonlinear)

OOF2
Why OOF2?

OOF2 reflects lessons learned from OOF1.
☐ More expandable.
☐ More flexible.

Emphases:
☐ *Extensibility* and *maintainability* through proper object-oriented design reflecting the underlying mathematics.
☐ *Generality* by making few assumptions about the problems being solved.
☐ *Usability* with a clear user interface.
☐ *Sanity* with a flexible infrastructure.
<table>
<thead>
<tr>
<th><strong>OOF1</strong></th>
<th><strong>OOF2</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate mesh generation &amp; solver programs</td>
<td>Single program</td>
</tr>
<tr>
<td>C++</td>
<td>C++ &amp; Python</td>
</tr>
<tr>
<td>Unthreaded, single processor</td>
<td>Threaded, parallel processors (soon)</td>
</tr>
<tr>
<td>Extended with difficulty</td>
<td>Easily extendible</td>
</tr>
<tr>
<td>Fixed physics</td>
<td>Arbitrary couplings</td>
</tr>
<tr>
<td>Linear triangular elements</td>
<td>Higher order triangles &amp; quads</td>
</tr>
<tr>
<td></td>
<td>More tools, more outputs, more, more, more (I’m still not satisfied)</td>
</tr>
</tbody>
</table>
Easily extendible to a wide variety of problems

- elasticity, plasticity, thermal conductivity, mass diffusion, electrical polarization, piezoelectricity, ferroelectricity, Darcy’s Law fluid flow, ...

\[ \sigma = \sum_i k_i \nabla \phi_i \quad -\nabla \cdot \sigma = f \]

<table>
<thead>
<tr>
<th>Field Φ</th>
<th>Elasticity</th>
<th>Thermal Cond.</th>
</tr>
</thead>
<tbody>
<tr>
<td>displacement</td>
<td>temperature</td>
<td>?</td>
</tr>
<tr>
<td>Flux σ</td>
<td>force</td>
<td>heat flow</td>
</tr>
<tr>
<td>Modulus k</td>
<td>$C_{ijkl}$</td>
<td>$K_{ij}$</td>
</tr>
<tr>
<td>Force f</td>
<td>force</td>
<td>heat source, sink</td>
</tr>
</tbody>
</table>

Designed for simple addition of new fields, fluxes, and equations.
Why OOF2?

- For example (proper design):
  - Physics and Finite Element class structure more closely tied to the underlying mathematics.
  - Allows more physics and more types of finite elements.

- Properties can be coded completely independently from the element classes.
OOF2 Code Ingredients

- **C++** (core) and **Python** (interface).
- C++/Python glue code generated by **SWIG**.
- **Libraries:**
  - **GTK+** graphics.
  - **PETSc, MPI** parallel solvers.
  - **ImageMagick** image manipulation.
  - **IML++**, **MV++**, **SparseLib++** linear algebra.
- **Threading**
OOF2 Conceptual Ingredients

- image
- materials
  - assembled from lists of properties
- microstructure
  - materials assigned to groups of pixels
- skeleton
  - only the geometry of the finite element mesh
- mesh
  - skeleton + mathematics + physics
- solution
Interface leads users through the tasks
Image Modification tools
Material Construction GUI
Material Construction GUI
Graphics Window

The image shows a computer window with various settings and options. The window is titled "OOF2 Graphics 1". The toolbox is set to "Pixel Selection" and the method is set to "Burn". The local flammability is 0.02, the global flammability is 0.05, and the color space norm is L1. The next nearest option is set to false. The selection size is 770. The window also includes a layer for selection. The image displayed is a black and white pattern with red highlights.
Extensibility via Class Hierarchy

- **Registered classes represent:**
  - Operations on images, meshes, etc.
  - Material properties.
  - Parameters for the above.

- **Registrations** describe how to create objects in the classes

- Menus and GUI components are created *automatically* from Registrations.
Extensibility via Class Hierarchy

**RegisteredClass**

**SkeletonModifier**

**Anneal**

```python
Registration('Anneal', SkeletonModifier, Anneal,
params=[
    RegisteredParameter('targets', FiddleNodesTargets, tip='Which nodes to fiddle.'),
    RegisteredParameter('criterion', skeletonmodifier.SkelModCriterion, tip='Acceptance...'),
    FloatParameter('T', value = 0.0, tip='Effective “temperature” of ...'),
    FloatParameter('delta', value=1.0, tip='Width of the distribution of ...'),
    RegisteredParameter('iteration', IterationManager, tip='Iteration method')
],
tip='Move nodes randomly and accept the ones that meet the acceptance criterion.'
)```
Extensibility via Class Hierarchy

RegisteredClass

SkeletonModifier

Anneal

Registration('Anneal', SkeletonModifier, Anneal,
params=[
    RegisteredParameter('targets', FiddleNodesTargets, tip='Which nodes to fiddle.'),
    RegisteredParameter('criterion', skeletonmodifier.SkelModCriterion, tip='Acceptance...'),
    FloatParameter('T', value=0.0, tip='Effective “temperature” of...'),
    FloatParameter('delta', value=1.0, tip='Width of the distribution of...'),
    RegisteredParameter('iteration', IterationManager, tip='Iteration method')
],
tip='Move nodes randomly and accept the ones that meet the acceptance criterion.'
)

Parameters provide all information needed to construct an object.
Refine Skeleton Modifier Registration ('Anneal', SkeletonModifier, params=[RegisteredParameter('targets', FiddleNodesTargets, tip='Which nodes to fiddle.'), RegisteredParameter('criterion', skeletonmodifier.SkelModCriterion, tip='Acceptance..'), FloatParameter('T', value=0.0, tip='Effective “temperature” of ...'), FloatParameter('delta', value=1.0, tip='Width of the distribution of ...'), RegisteredParameter('iteration', IterationManager, tip='Iteration method')], tip='Move nodes randomly and accept the ones that meet the acceptance criterion.

Extensibility via Class Hierarchy

RegisteredClassFactory is built automatically in the GUI
Task: Solver

Microstructure: small.ppm
Skeleton: skeleton
Mesh: mesh

Solver

LinearSolver

GMRES

- method =
- max_iterations = 1000
- krylov_dimension = 100
- tolerance = 1e-08
- preconditioner =
- inplane = ILU
Extending OOF2 with new Physics

New Fields require just a few lines of Python:

```python
temperature = defineField(ScalarField("Temperature"))
heat_flux = defineFlux(VectorFlux("Heat_Flux"))
heat_eqn = defineEquation(DivergenceEquation("Heat", heat_flux, 1))
planeheatflux_eqn = defineEquation(PlaneFluxEquation("Plane_Heat_Flux", heat_flux, 1)),

displacement = defineField(TwoVectorField("Displacement"))
stress_flux = defineFlux(SymmetricTensorFlux("Stress"))
forcebalance_eqn = defineEquation(DivergenceEquation("Force_Balance", stress_flux, spacedim))
planestress_eqn = defineEquation(PlaneFluxEquation("Plane_Stress", stress_flux, 3))
```

Actually using new Fields in material properties requires a bit more effort…
Finite Elements in 50 Words or Less*

- Divide space into elements.
- Evaluate fields at nodes between elements: $u_{n\nu} = u_n(x_\nu)$
- Interpolate fields in elements via shape functions $N_\nu(x)$

$$u_n(x) = \sum_\nu u_{n\nu}N_\nu(x)$$

- Substitute expansion into equations, multiply by a test function, integrate by parts, and solve the resulting system of linear equations for the unknowns $u_{n\nu}$.

*Pedants will insist upon “fewer” instead of “less” here, but “less” is the colloquial usage.
Adding New Material Properties

- A “Property” is a term in a flux: \[ \sigma = \sum_i k_i \nabla \phi_i \]

- Define \[ \sigma = M \cdot u \]
  - \( u \) is the vector of all field values at all nodes of an element
  - \( M \) is the “flux matrix”

- Developer must provide a routine to compute an element’s contribution to \( M \) at \( x \) for node \( v \).
  - This can be done with no explicit knowledge of the element geometry.
Example: Elasticity

- Displacement component $l$ at point $x$: $u_l(x)$
- Stress component $ij$ at $x$: $\sigma_{ij}(x) = C_{ijkl} \partial_k u_l(x)$
- Expand in shape functions: $u_l(x) = N_\nu(x) u_{l\nu}$

- $u_{l\nu}$ is displacement component $l$ at node $\nu$.

- $\sigma_{ij}(x) = C_{ijkl} \partial_k N_\nu(x) u_{l\nu}$

- Compare to $\sigma_{ij}(x) = M_{ij}^{k\nu} u_{k\nu}$

- Find $M_{ij}^{k\nu}(x) = C_{ijkl} \partial_l N_\nu(x)$
Example: Elasticity

```cpp
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
const ElementFuncNodeIterator &nu,
Flux *flux, const MasterPosition &x) const
{
    if(*flux != *stress_flux) {
        throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
    }

    const Cijkl modulus = cijkl(mesh, element, x);
    double sf = nu.shapefunction(x);
    double dsf0 = nu.dshapefunction(0, x);
    double dsf1 = nu.dshapefunction(1, x);

    for(SymTensorIndex ij; !ij.end(); ++ij) {
        for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
            SymTensorIndex ell0(0, ell.integer());
            SymTensorIndex ell1(1, ell.integer());
            stress_flux->matrix_element(mesh, ij, displacement, ell, nu) +=
                modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
        }
    }
    if(!displacement->in_plane(mesh)) {
        Field *oop = displacement->out_of_plane();
        for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
            stress_flux->matrix_element(mesh, ij, oop, ell, nu) +=
                modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
        }
    }
}
```
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element, const ElementFuncNodeIterator &nu, Flux *flux, const MasterPosition &x) const
{
  if(*flux != *stress_flux) {
    throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
  }
  const Cijkl modulus = cijkl(mesh, element, x);
  double sf = nu.shapefunction(x);
  double dsf0 = nu.dshapefunction(0, x);
  double dsf1 = nu.dshapefunction(1, x);

  for(SymTensorIndex ij; !ij.end(); ++ij) {
    for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
      SymTensorIndex ell0(0, ell.integer());
      SymTensorIndex ell1(1, ell.integer());
      stress_flux->matrix_element(mesh, ij, displacement, ell, nu) +=
        modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
    }
    if(!displacement->in_plane(mesh)) {
      Field *oop = displacement->out_of_plane();
      for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
        stress_flux->matrix_element(mesh, ij, oop, ell, nu)
          += modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
      }
    }
  }
}
Example: Elasticity

```cpp
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
                           const ElementFuncNodeIterator &nu,
                           Flux *flux, const MasterPosition &x) const
{
    if(*flux != *stress_flux) {
        throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
    }

    const Cijkl modulus = cijkl(mesh, element, x);
    double sf = nu.shapefunction(x);
    double dsf0 = nu.dshapefunction(0, x);
    double dsf1 = nu.dshapefunction(1, x);

    for(SymTensorIndex ij; !ij.end(); ++ij) {
        for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
            SymTensorIndex ell0(0, ell.integer());
            SymTensorIndex ell1(1, ell.integer());
            stress_flux->matrix_element(mesh, ij, displacement, ell, nu) +=
                modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
        }
    }
    if(!displacement->in_plane(mesh)) {
        Field *oop = displacement->out_of_plane();
        for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
            stress_flux->matrix_element(mesh, ij, oop, ell, nu)
                += modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
        }
    }
}
```

Sanity Check

Disclaimer: slightly simplified to fit on the screen...
Example: Elasticity

```cpp
Example:

```cpp
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
   const ElementFuncNodeIterator &nu,
   Flux *flux, const MasterPosition &x) const
{
  if(*flux != *stress_flux) {
    throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
  }

  const Cijkl modulus = cijkl(mesh, element, x);
  double sf = nu.shapefunction(x);
  double dsf0 = nu.dshapefunction(0, x);
  double dsf1 = nu.dshapefunction(1, x);

  for(SymTensorIndex ij; !ij.end(); ++ij) {
    for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
      SymTensorIndex ell0(0, ell.integer());
      SymTensorIndex ell1(1, ell.integer());
      stress_flux->matrix_element(mesh, ij, displacement, ell, nu) +=
      modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
    }
  }

  if(!displacement->in_plane(mesh)) {
    Field *oop = displacement->out_of_plane();
    for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
      stress_flux->matrix_element(mesh, ij, oop, ell, nu)
      += modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
    }
  }
}
```
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element, 
    const ElementFuncNodeIterator &nu, 
    Flux *flux, const MasterPosition &x) const 
{
    if(*flux != *stress_flux) {
        throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
    }

    const Cijkl modulus = cijkl(mesh, element, x);
    double sf = nu.shapefunction(x);
    double dsf0 = nu.dshapefunction(0, x);
    double dsf1 = nu.dshapefunction(1, x);

    for(SymTensorIndex ij; !ij.end(); ++ij) {
        for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
            SymTensorIndex ell0(0, ell.integer());
            SymTensorIndex ell1(1, ell.integer());
            stress_flux->matrix_element(mesh, ij, displacement, ell, nu) += 
                modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
        }
    }
    if(!displacement->in_plane(mesh)) {
        Field *oop = displacement->out_of_plane();
        for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
            stress_flux->matrix_element(mesh, ij, oop, ell, nu) += 
                modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
        }
    }
}

Shape function evaluation for node v at point x

Disclaimer: slightly simplified to fit on the screen...
Example: Elasticity

```cpp
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
                            const ElementFuncNodeIterator &nu,
                            Flux *flux, const MasterPosition &x) const {
    if(*flux != *stress_flux) {
        throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
    }

    const Cijkl modulus = cijkl(mesh, element, x);
    double sf = nu.shapefunction(x);
    double dsf0 = nu.dshapefunction(0, x);
    double dsf1 = nu.dshapefunction(1, x);

    for(SymTensorIndex ij; !ij.end(); ++ij) {
        for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
            SymTensorIndex ell0(0, ell.integer());
            SymTensorIndex ell1(1, ell.integer());

            stress_flux->matrix_element(mesh, ij, displacement, ell, nu) +=
                                modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
        }
    }

    if(!displacement->in_plane(mesh)) {
        Field *oop = displacement->out_of_plane();
        for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
            stress_flux->matrix_element(mesh, ij, oop, ell, nu) +=
                                modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
        }
    }
}
```

For all stress components $ij$

For all displacement components $l$

\[ M_{ij}^{lv}(x) = \sum_l C_{ijkl} \partial_k N_v(x) \]

$l,v \Rightarrow$ degree of freedom

$ij \Rightarrow$ stress component
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element, 
    const ElementFuncNodeIterator &nu, 
    Flux *flux, const MasterPosition &x) const 
{
    if(*flux != *stress_flux) {
        throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
    }

    const Cijkl modulus = cijkl(mesh, element, x);
    double sf = nu.shapefunction(x);
    double dsf0 = nu.dshapefunction(0, x);
    double dsf1 = nu.dshapefunction(1, x);

    for(SymTensorIndex ij; !ij.end(); ++ij) {
        for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
            SymTensorIndex ell0(0, ell.integer());
            SymTensorIndex ell1(1, ell.integer());
            stress_flux->matrix_element(mesh, ij, displacement, ell, nu) +=
                modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
        }
    }

    if(!displacement->in_plane(mesh)) {
        Field *oop = displacement->out_of_plane();
        for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
            stress_flux->matrix_element(mesh, ij, oop, ell, nu) +=
                modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
        }
    }
}
Example: Elasticity

```cpp
void Elasticity::fluxmatrix(const FEMesh *mesh, const Element *element,
                const ElementFuncNodeIterator &nu,
                Flux *flux, const MasterPosition &x) const
{
    if(*flux != *stress_flux) {
        throw ErrProgrammingError("Unexpected flux", __FILE__, __LINE__);
    }

    const Cijkl modulus = cijkl(mesh, element, x);
    double sf = nu.shapefunction(x);
    double dsf0 = nu.dshapefunction(0, x);
    double dsf1 = nu.dshapefunction(1, x);

    for(SymTensorIndex ij; !ij.end(); ++ij) {
        for(FieldIterator ell=displacement->iterator(); !ell.end(); ++ell) {
            SymTensorIndex ell0(0, ell.integer());
            SymTensorIndex ell1(1, ell.integer());
            stress_flux->matrix_element(mesh, ij, displacement, ell, nu) +=
                modulus(ij, ell0)*dsf0 + modulus(ij, ell1)*dsf1;
        }

        if(!displacement->in_plane(mesh)) { // Displacement
            Field *oop = displacement->out_of_plane();
            for(FieldIterator ell=oop->iterator(ALL_INDICES); !ell.end(); ++ell) {
                stress_flux->matrix_element(mesh, ij, oop, ell, nu) +=
                    modulus(ij, SymTensorIndex(2,ell.integer())) * sf;
            }
        }
    }
}
```

- **No** explicit dependence on:
  - Element geometry
    - triangle, quadrilateral
  - Element order
    - linear, quadratic...
  - Equation
    - divergence, plane-stress
  - Other material properties
More Infrastructure

- Underlying menu driven structure (in Python):
  - Specify name, callback function, menu, argument parameters.
  - Menu items created explicitly, or implicitly from Registrations.

- Communication between different code components is by means of a “switchboard”
  - Objects send messages to switchboard.
  - Other objects subscribe to messages.
  - Sending object doesn’t have to know who (if anybody) is listening.
  - Allows modular development and use.
OOF2 Control Structure

- Scripts
- Command Line Interface
- Binary Interface
- Menu system
- Graphical User Interface
- Switchboard
- Routines that actually do stuff
- Data Files
GUI, Threading & Parallel Processing

- OOF is meant to be an interactive system in which users can experiment with different scenarios in real time.
  - Need a responsive multithreaded interface.
  - Parallel back-end for quick turnaround.

- Still, lengthy computations need to be performed in batch mode, without a GUI.

- “Worker” classes added to menu system to handle different modes of operation.
  - TextWorker, GUIWorker, ThreadedWorker, etc.
OOF2 Control Structure

- Scripts
- Command Line Interface
- Menu System
- Graphical User Interface
- Switchboard
- Worker
- Routines that actually do stuff

Front End Thread
Back End Thread
http://www.ctcms.nist.gov/oof/

- **OOF1**
  - source code
  - precompiled binaries
  - manuals & tutorials

- **OOF2**
  - source code with built-in tutorials
  - precompiled binaries (soon)
  - manuals (soon)

- **Mailing list**