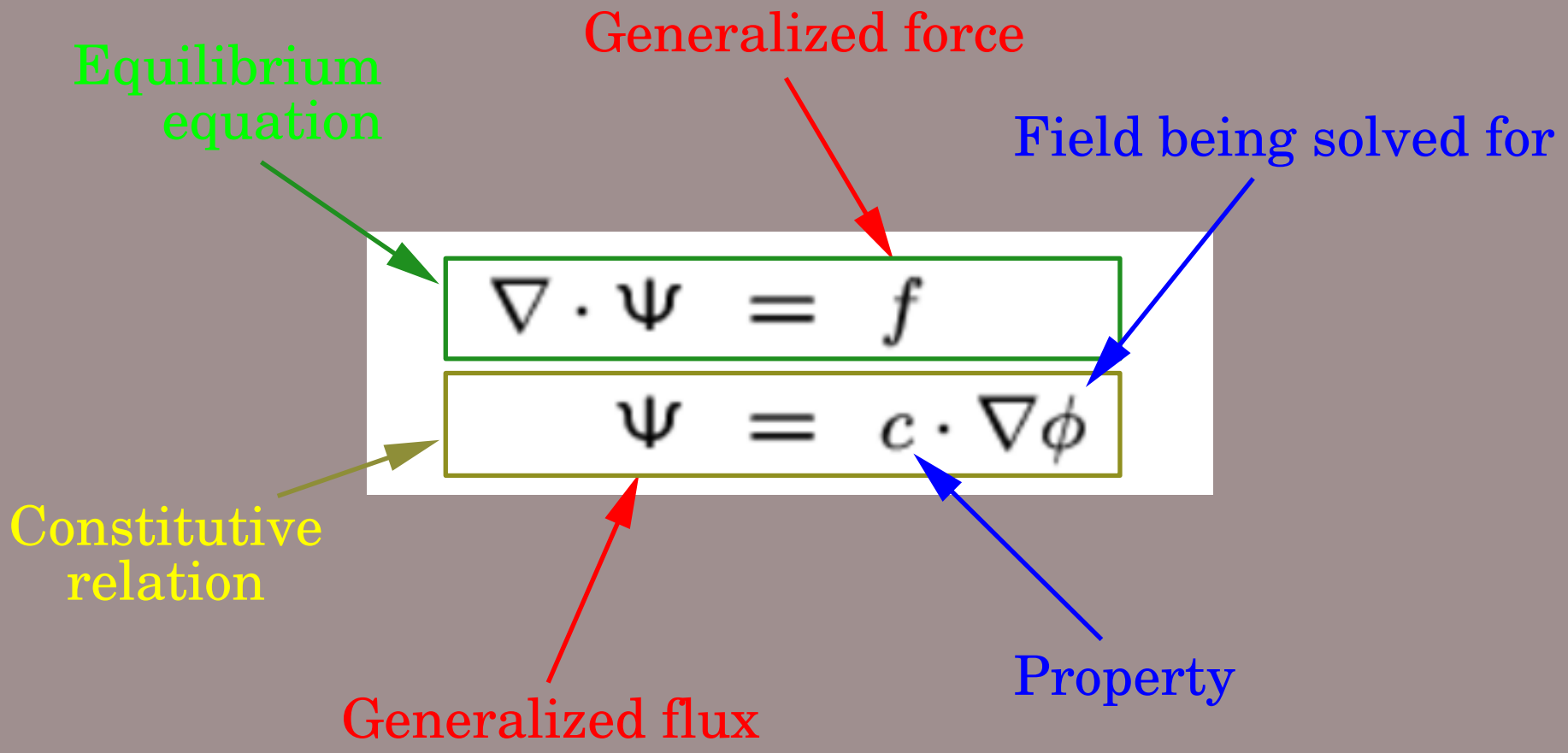
OOF2: Progress and Plans

Andrew Reid
MIT/NIST

June 21, 2001

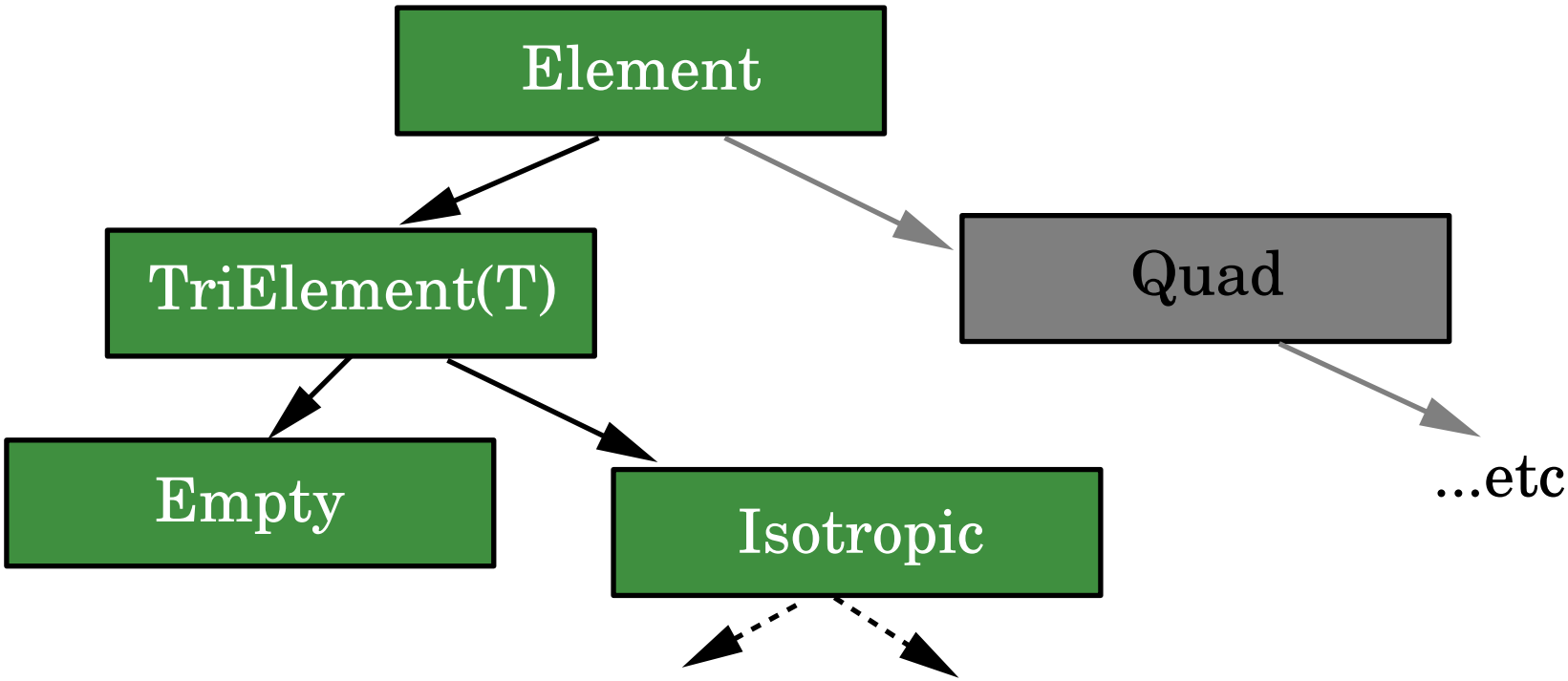
Scope

The general case of an OOF–accessible problem:

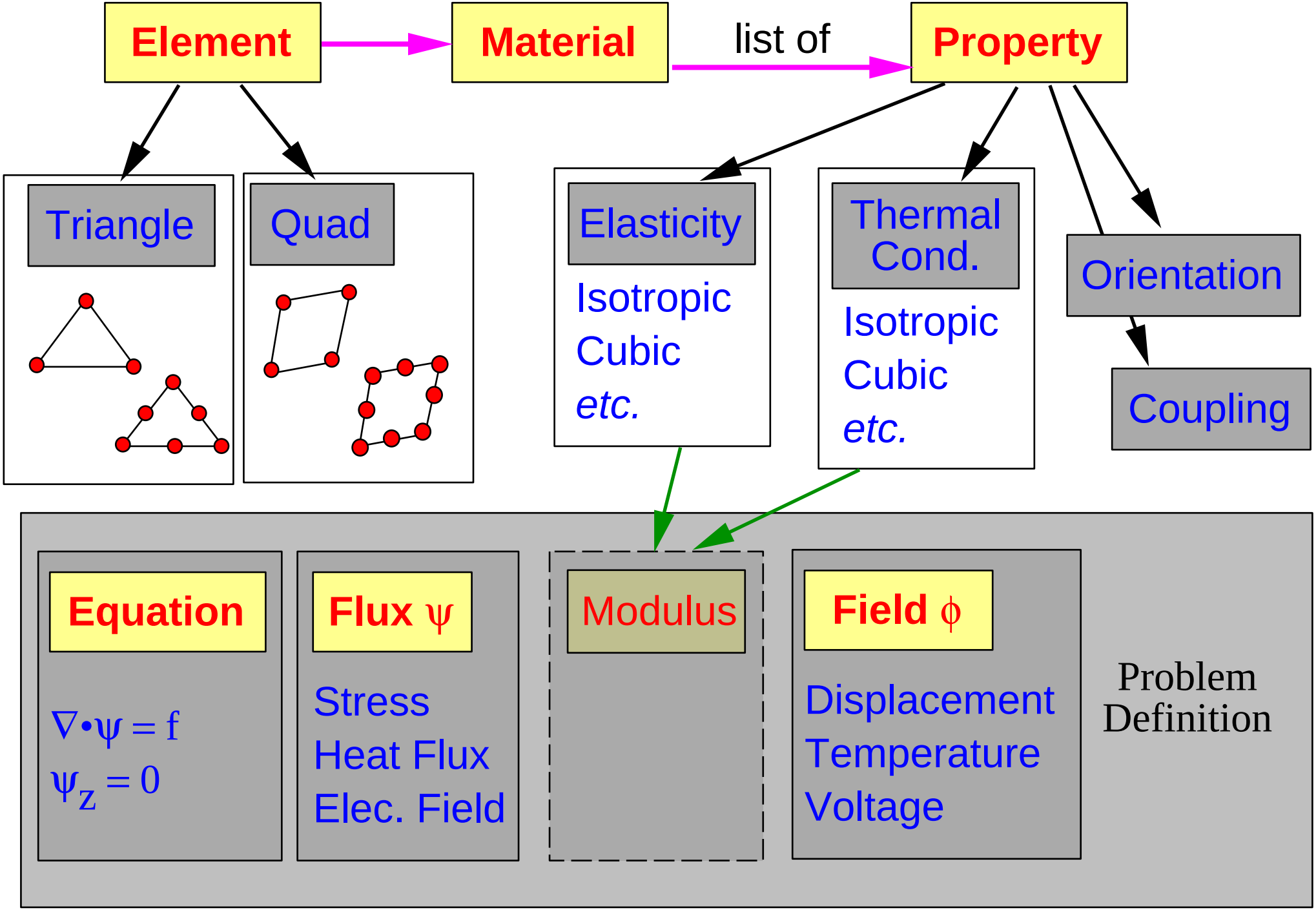


Architecture: OOF1 vs OOF2

In OOF1, elements contain everything. Adding new functionality means building a new kind of element.



In OOF2, elements are primarily geometric objects. They have pointers to materials and properties, and associated fluxes. Fields are defined separately from elements.


Element
Material
list of
Property
Triangle
Quad
Elasticity
Isotropic
Cubic
etc.
Thermal Cond.
Isotropic
Cubic
etc.
Orientation
Coupling
Equation
$\nabla \cdot \psi = f$
$\psi_z = 0$
Flux ψ
Stress
Heat Flux
Elec. Field
Modulus
Field ϕ
Displacement
Temperature
Voltage
Problem Definition

OOF1 Extensions

Many of the lessons learned about extensibility took place in the context of interesting and valuable extensions in the OOF1 code by developers and users

- Solution mesh refinement
- Piezo–electric properties
- Viscoelasticity

These refinements, with the lessons learned, will be included in the OOF2 code

OOF2 Extensibility

OOF2 has additional design features intended to make use and extension easy

- High–level code written in Python
- Input files are Python scripts –– human–readable, platform independent
- User interface will allow import, export of materials and properties

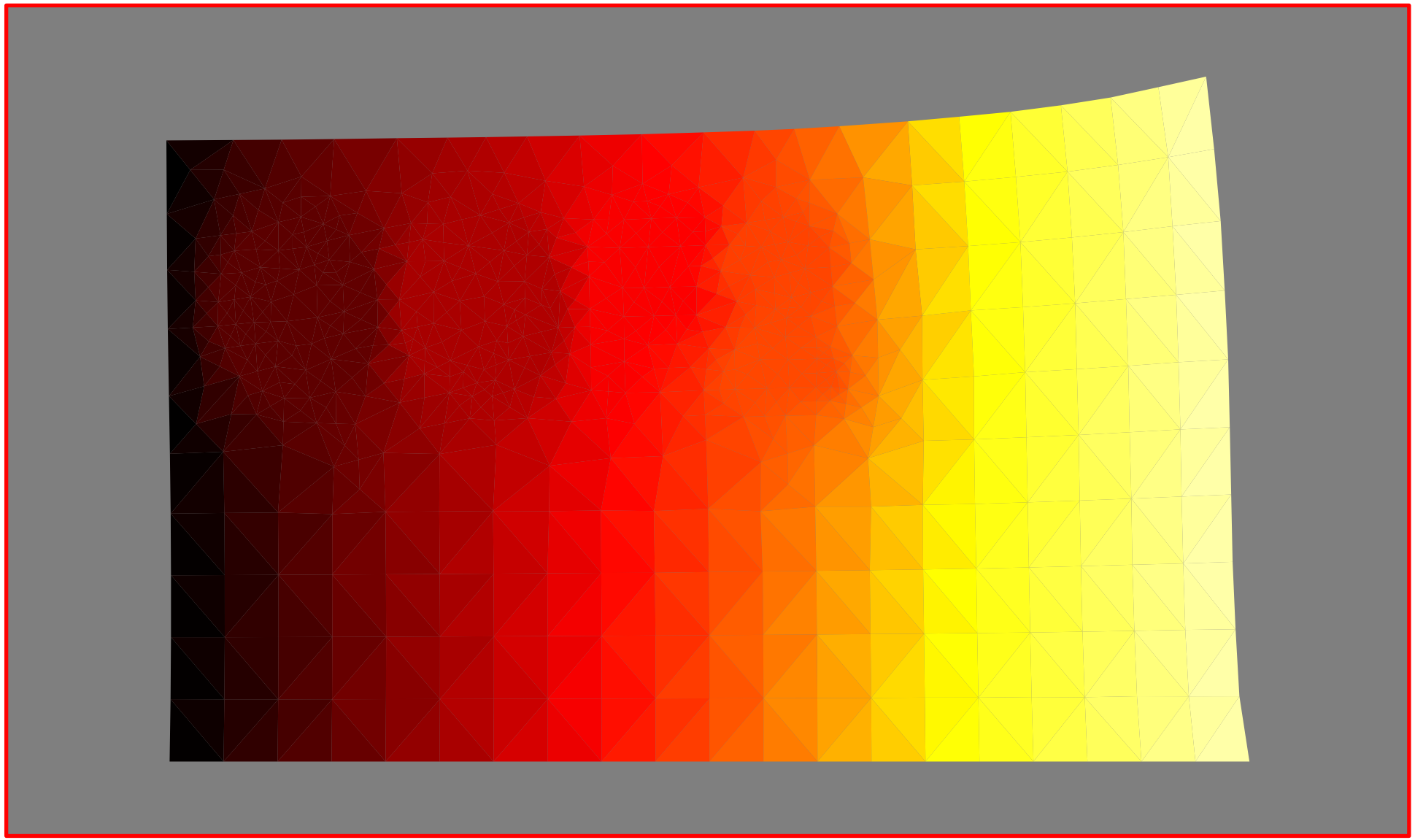
Progress

Working pieces of OOF2:

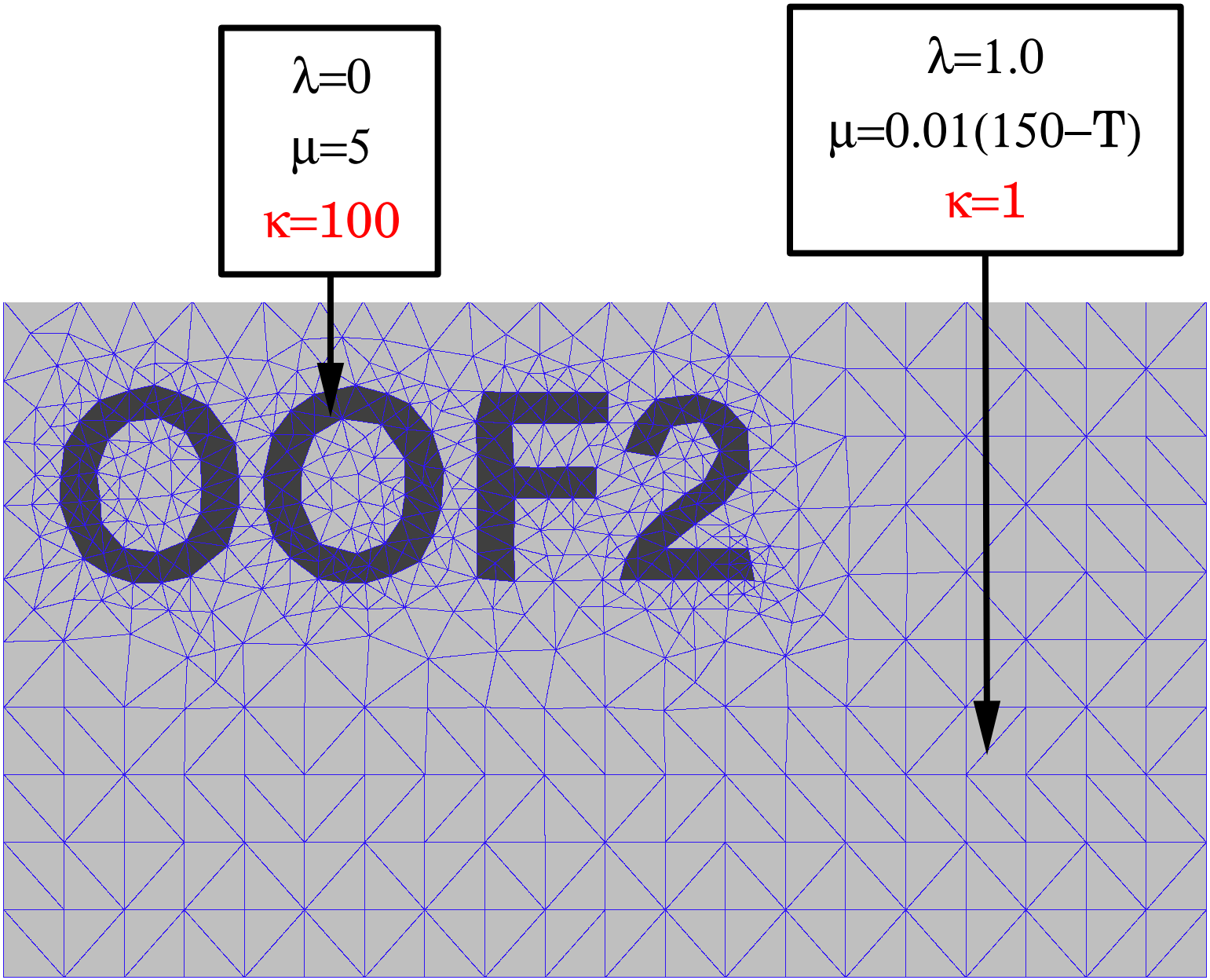
- Linear matrix solver
- Non–topological boundary conditions
- Linear elastic and simple thermal materials
- Python–scripted input
- Graphical output (preliminary)

An OOF2 Result:

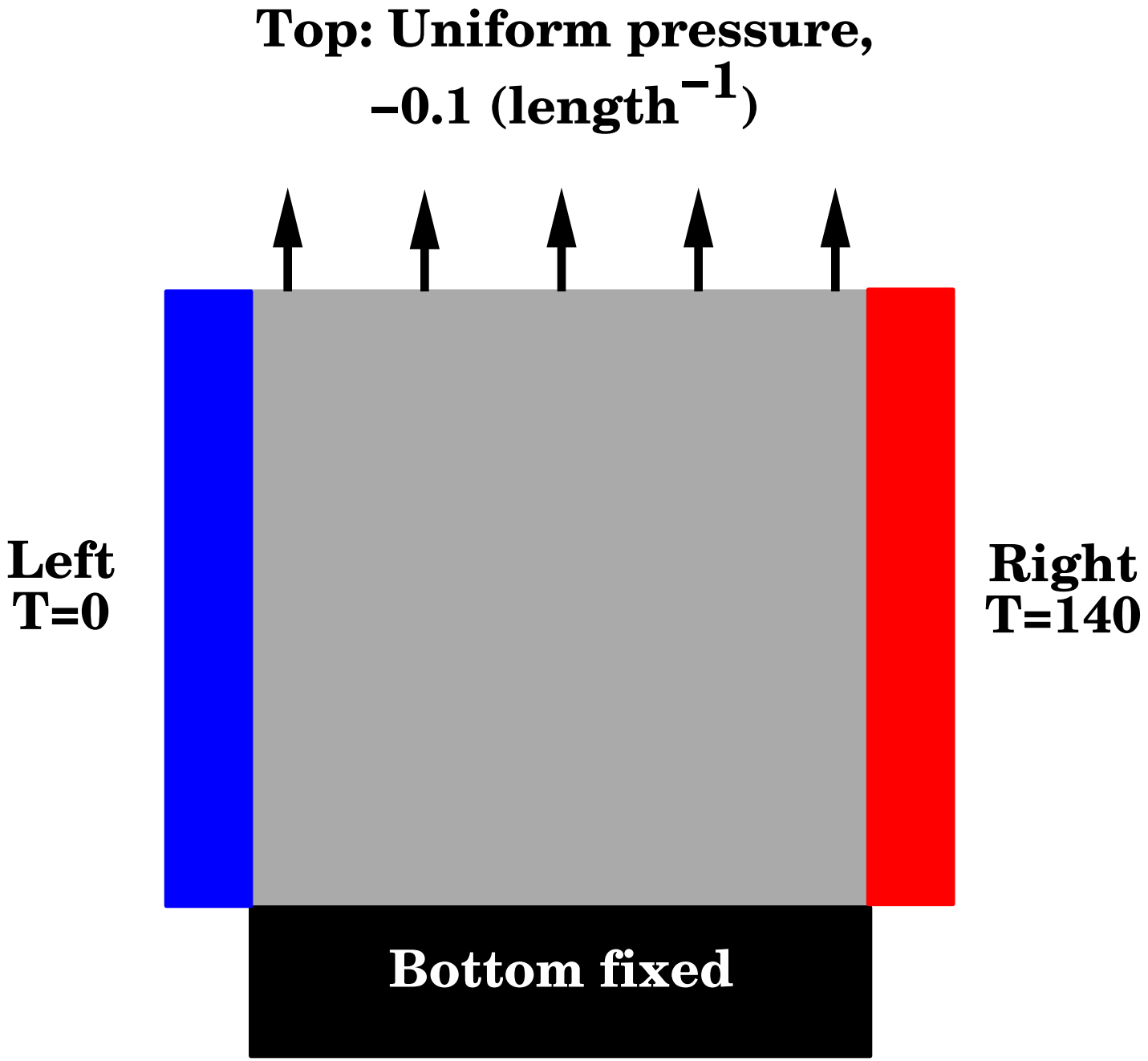
Temperature–dependent elasticity with inclusions:



Anatomy of the system:



Boundary conditions:

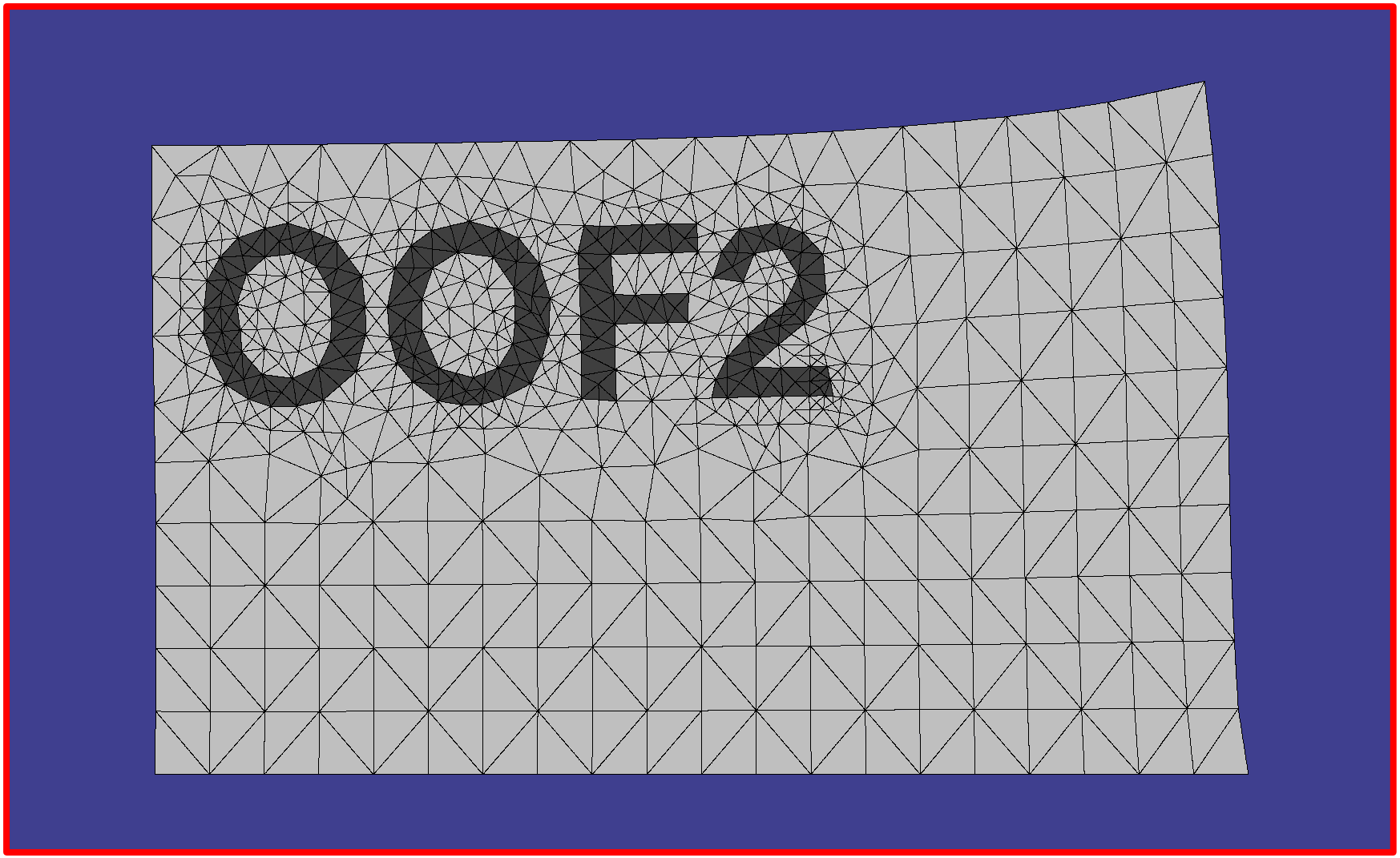


Solution method:

In the absence of a nonlinear solver, we use a two–stage technique:

(1) Equilibrate temperature field

(2) Apply mechanical load in fixed temperature environment

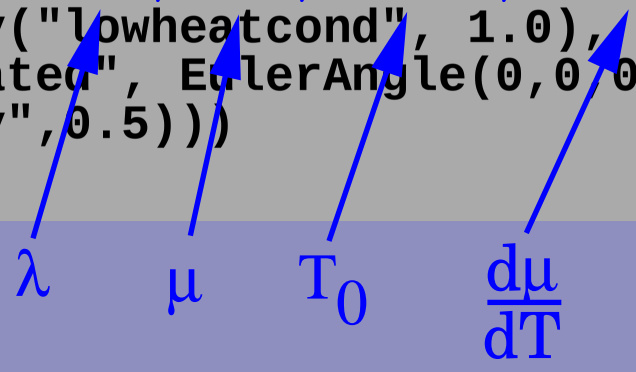


Coder's view: Using the material

Top–level Python code:

The temperature–dependent matrix material:

```
papermaterial = newMaterial("paper",
                ThermoElasticity("Soft-shear",
                                 1.0, 0.0, 150.0, -0.01),
                IsoHeatConductivity("lowheatcond", 1.0),
                Orientation("unrotated", EulerAngle(0,0,0)),
                ColorProperty("gray",0.5)))
```



The temperature–independent inclusion material:

```
lettermaterial = newMaterial("Iso-elastic",
                  IsoElasticity("test", 10.0, 0.0),
                  IsoHeatConductivity("highheatcond", 1000.0),
                  Orientation("unrotated", EulerAngle(0,0,0)))
```

The definition of “Temperature”:

In a global file containing non–mesh–dependent variables:

```
temperature = ScalarField("Temperature")
heat_flux = VectorFlux("Heat Flux")
heat_eqn = DivergenceEquation("Heat", heat_flux, 1)

heat_eqn.uses_flux(heat_flux)
```

In a property that uses the field:

In the property object’s initializer, simply set a local variable:

```
temperature = getField("Temperature");
```

Now, temperature is known to the "universe", so values can be assigned, and known to the property, so it can have the required dependency.

The T–dependent modulus:

Define a new material:
(Currently C++, will be accessible from Python)

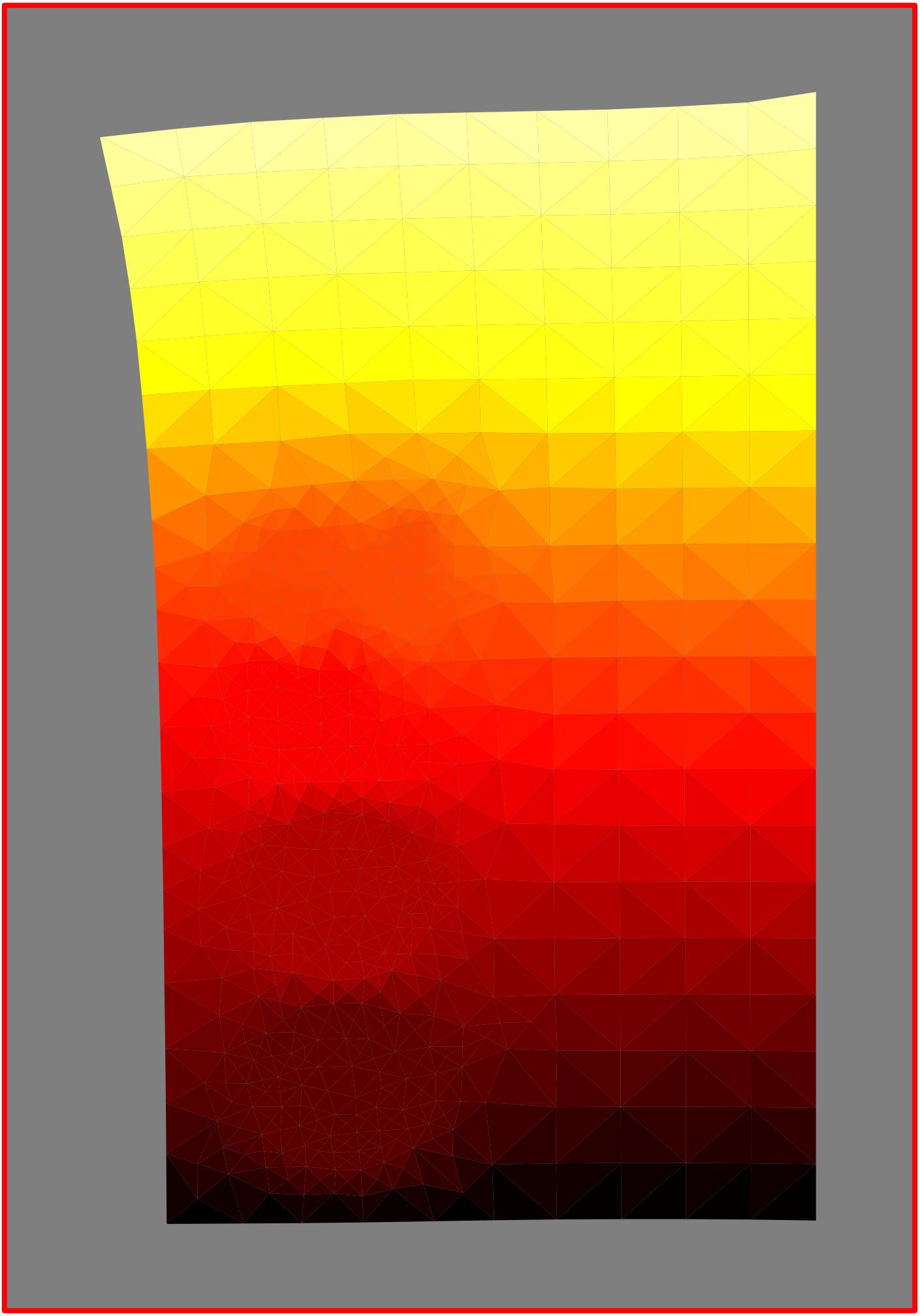
```
const Cijkl ThermoElasticity::cijkl(const Mesh *m,
                                    const Element *e,
                                     const MasterPosition &x) const {
  // Get local temperature
  double temp = e->interpolate(m, *temperature, 0, x);
  Cijkl modulus;
  double local_mu = mu + dmudt * (temp - tzero);
  modulus(0,0) = modulus(1,1) = modulus(2,2) =
                                        lambda+2.0*local_mu;
  modulus(0,1) = modulus(0,2) = modulus(1,2) = lambda;
  modulus(3,3) = modulus(4,4) = modulus(5,5) = local_mu;

  return modulus;
}
```

This is the code in the ThermoElasticity class that computes the elastic modulus. To make it T–dependent:

(1) Retrieve the local temperature

(2) Compute the local value of μ

(3) Carry on as usual

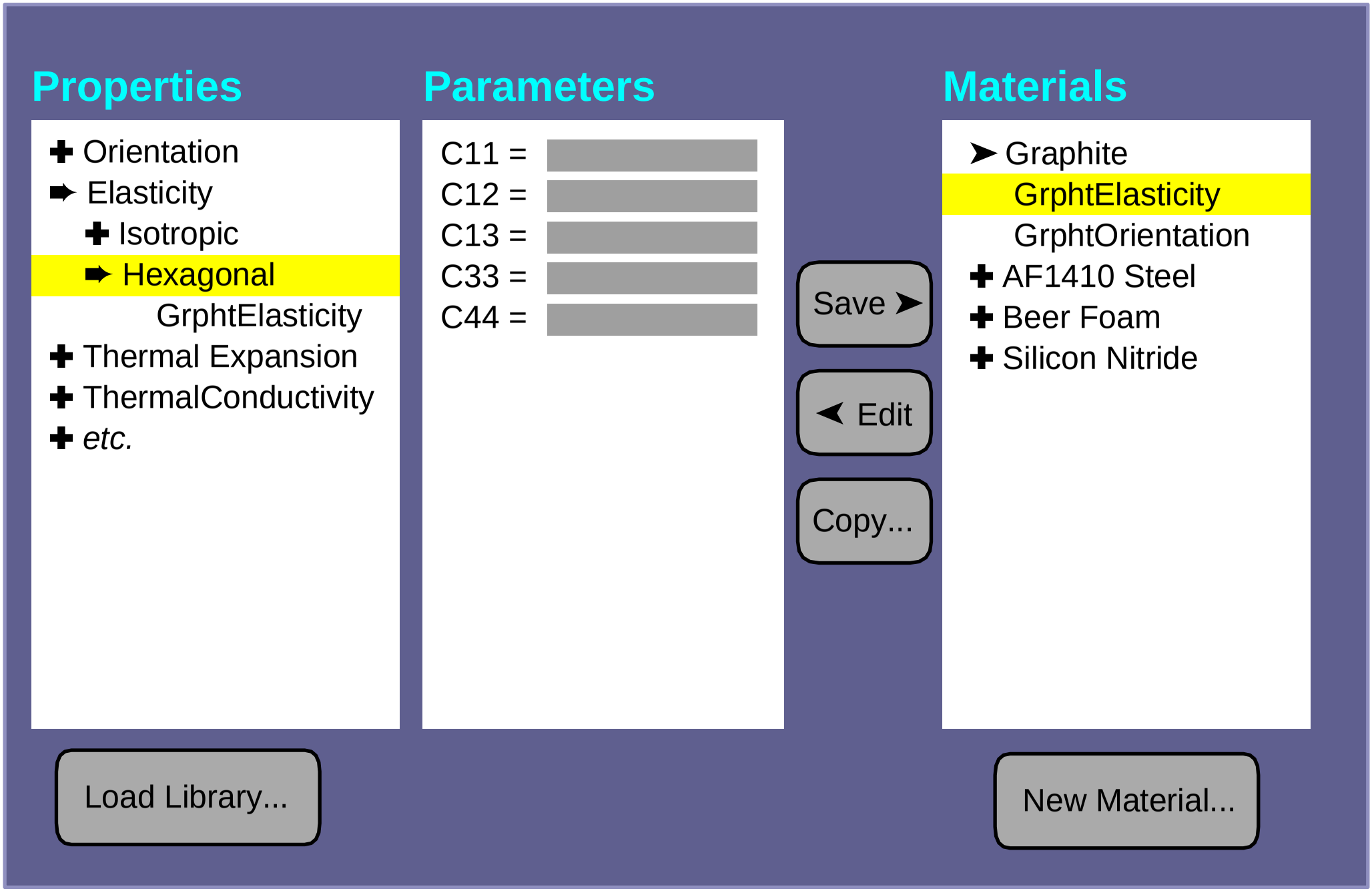


The Near Future

Steps remaining before an OOF2 release with OOF1 functionality

- Robust graphical output
- ppm2oof–like GUI input
- Front–end/back–end interface specification
- Integration mechanism for body forces

Material Builder (artist's conception)



Plans

OOF is designed to be user–friendly and user–extensible. We expect many users will be able to take on a wide variety of interesting materials problems with minimal assistance from the development team.

The development team does have a number of ideas for extensions and expansions of OOF2:

- **Plasticity**
- **3D**
- **Dynamics**
- **Higher–order continuity**

Plasticity

- Often–requested feature from users
- In complex microstructures, simple loads can have complex response, e.g. simultaneous loading and unloading
- Can draw on existing finite–element techniques
- Implement by field variables indicating plastic state –– yield stress, plastic strain, possibly others

3D

- Actual dimensionality of real materials
- Greater computational power, advances in microscopy make 3D data sets more common
- 3D FE techniques exist
- Primary challenge is to make the transition from data set to mesh in a structured, efficient way –– work on this is underway

Dynamics

- "Easy" dynamics –– time–dependent fields in materials with constant properties, e.g. transients in diffusion
- "Hard" dynamics –– changes in structure or properties in response to changing fields, e.g. phase transitions
- Present in preliminary form in OOF1

Higher–order continuity

- Ordinary FE analysis requires that fields be continuous everywhere, but they can have discontinuous derivatives
- For some constitutive models (e.g. strain–gradients), this may make physical quantities undefined
- Makes contact with phase–field techniques

Summary

- OOF2 offers OOF1 functionality in a more flexible, extensible architecture
- Core OOF2 system is working now
- Maintains Materials Science focus –– features driven by scientific user community