Overview of the MOOSE Framework and Applications to Materials Science

Larry Aagesen, Yongfeng Zhang, Daniel Schwen, Xianming Bai, Pritam Chakraborty, Bulent Biner, Jianguo Yu, Chao Jiang, Ben Beeler, Wen Jiang, Karim Ahmed

Michael Tonks

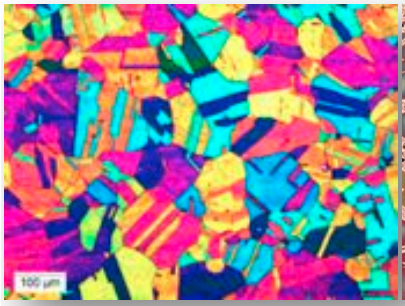
Paul Millett

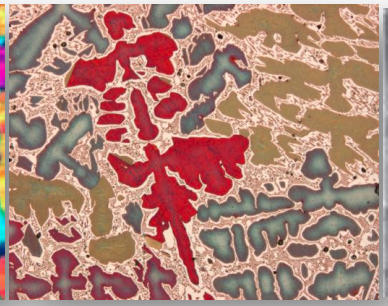
Overview

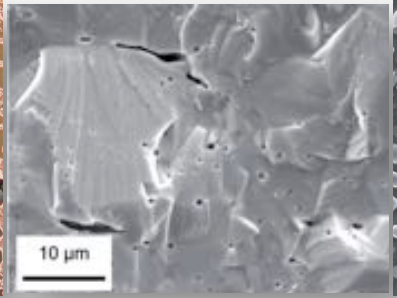
- General overview of the MOOSE framework
- MOOSE tools for meso-scale modeling
- Finite Element Method (FEM) and its implementation
- Phase-field modeling
- Examples and applications

Material Behavior

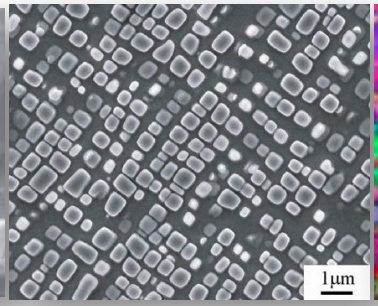
- A key objective of materials science is to understand the impact of microstructure on macroscale material behavior.


Annealed Stainless Steel

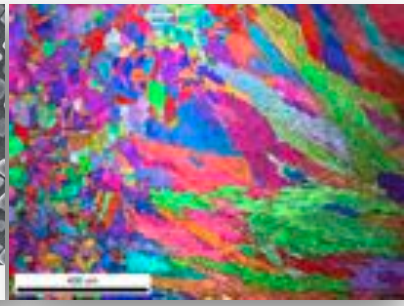

Cast Bronze



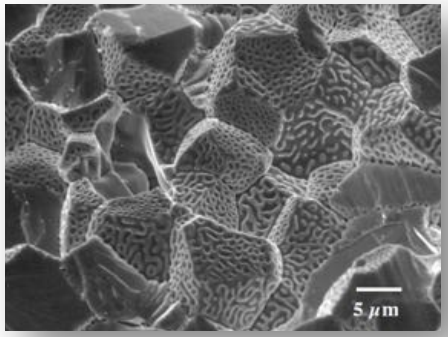
Sintered UO2



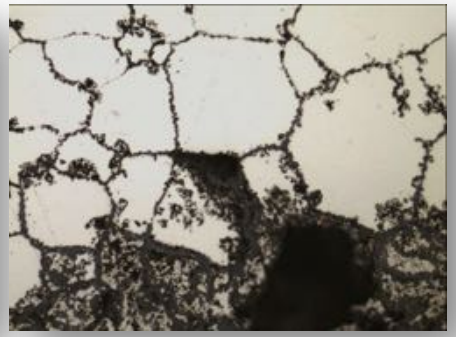
Co-Al-W Superalloy

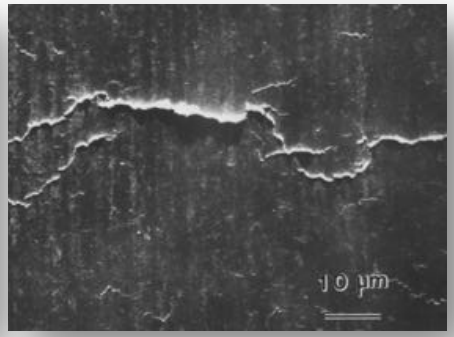

Friction stir welded stainless steel

- An essential part of that is predicting the impact of microstructure evolution.

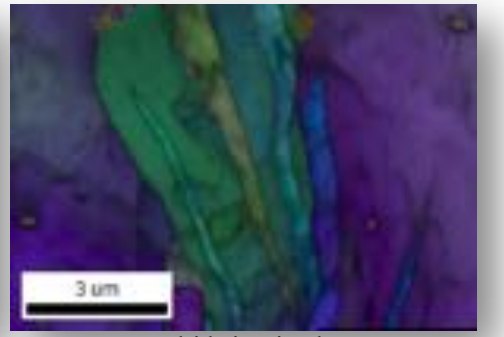


Irradiated UO_2 fuel


Corrosion in stainless steel



Micro-cracking in steel

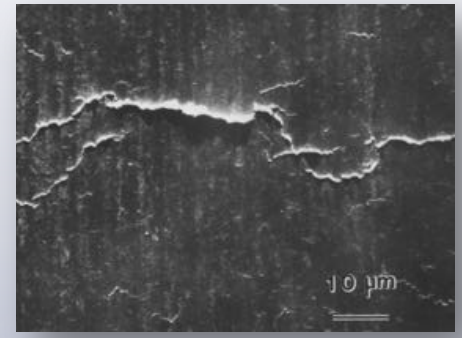


Hydride in Zircaloy

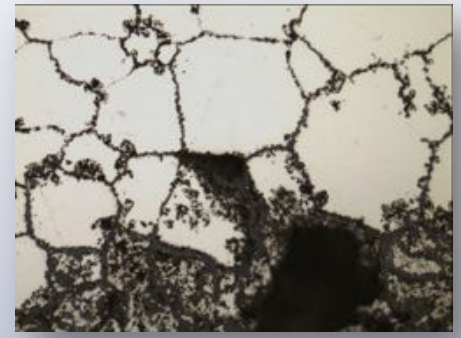
Material Behavior is Multiphysics

- Material behavior is influenced by many different physics, for example:

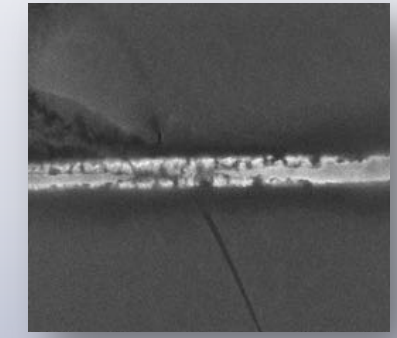
Mechanics
- Dislocations
- Cracking
- Stress-driven Diffusion



Chemistry
- Corrosion
- Oxidation
- Reactive transport



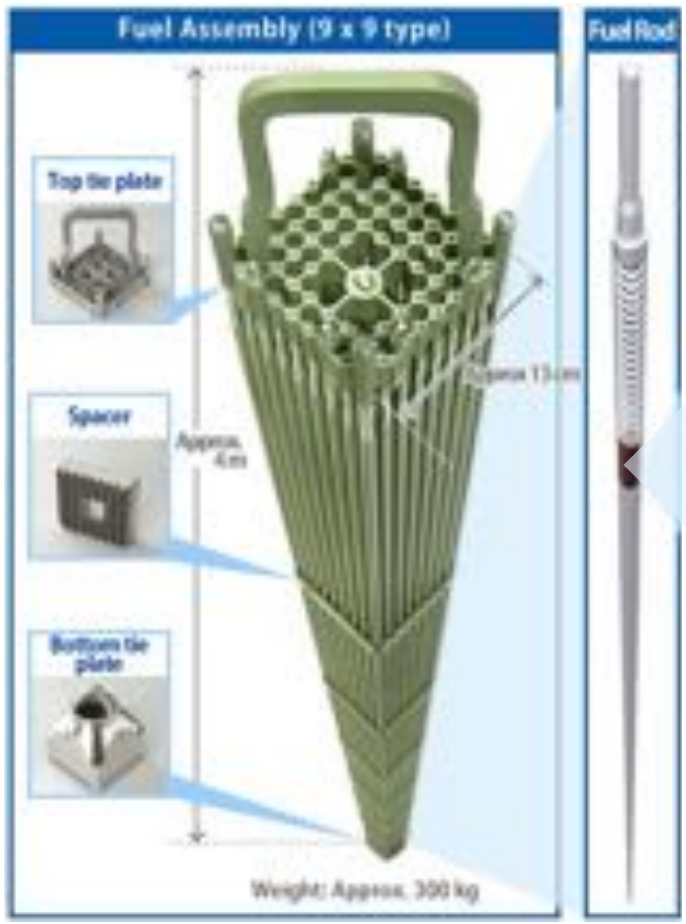
Electricity/Magnetism
- Electromigration
- Ferroelectricity
- Ferromagnetism

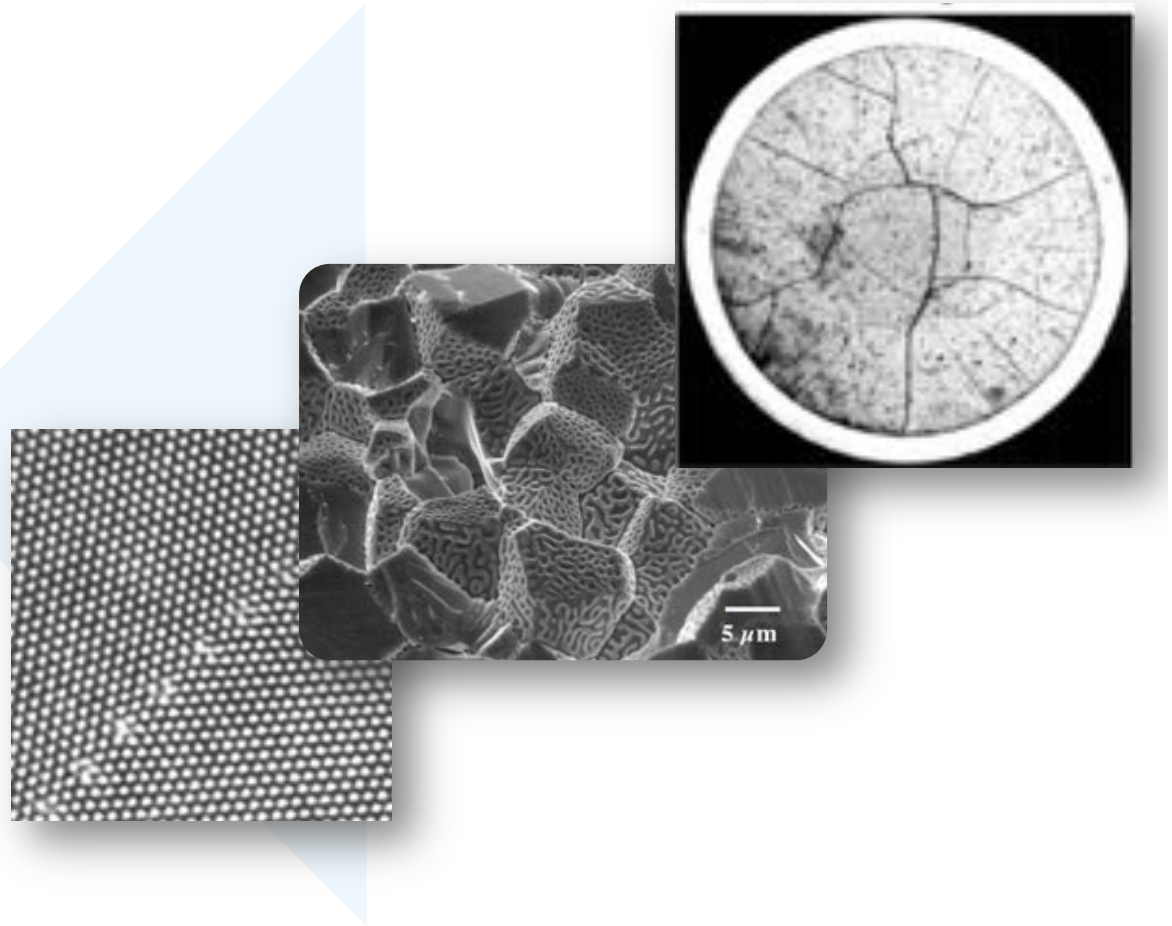


Heat Conduction
- Species transport
- Melting
- Precipitation

Material Behavior is Multiscale

- Material behavior at the atomistic and microscales drives macroscale response.



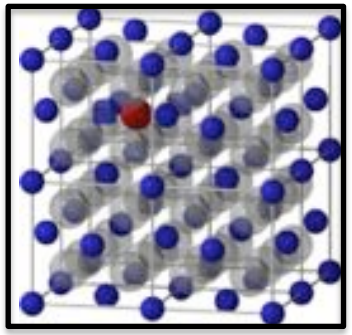


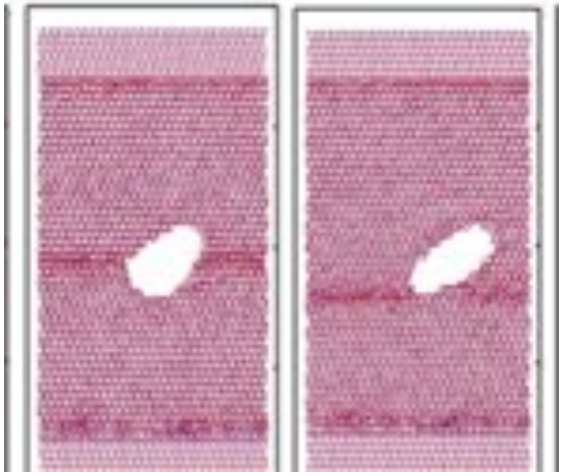
Multiscale Modeling Approach

- Simulations at smaller scales inform the models at increasing length scales

Atomic scale bulk DFT + MD

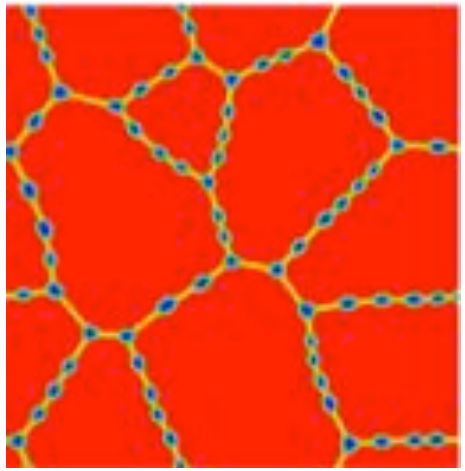
- Identify important bulk mechanisms
- Determine bulk material parameters





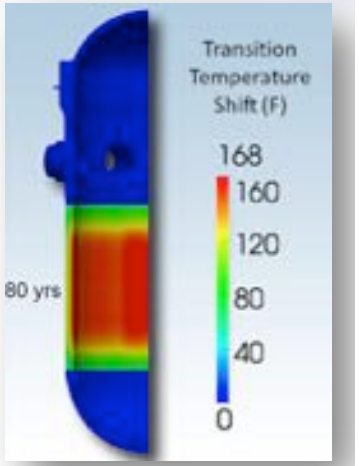
Atomic scale microstructure MD

- Investigate role of idealized interfaces
- Determine interfacial properties



Mesoscale models

- Predict and define microstructure evolution
- Determine effect of evolution on material properties



Engineering scale simulation

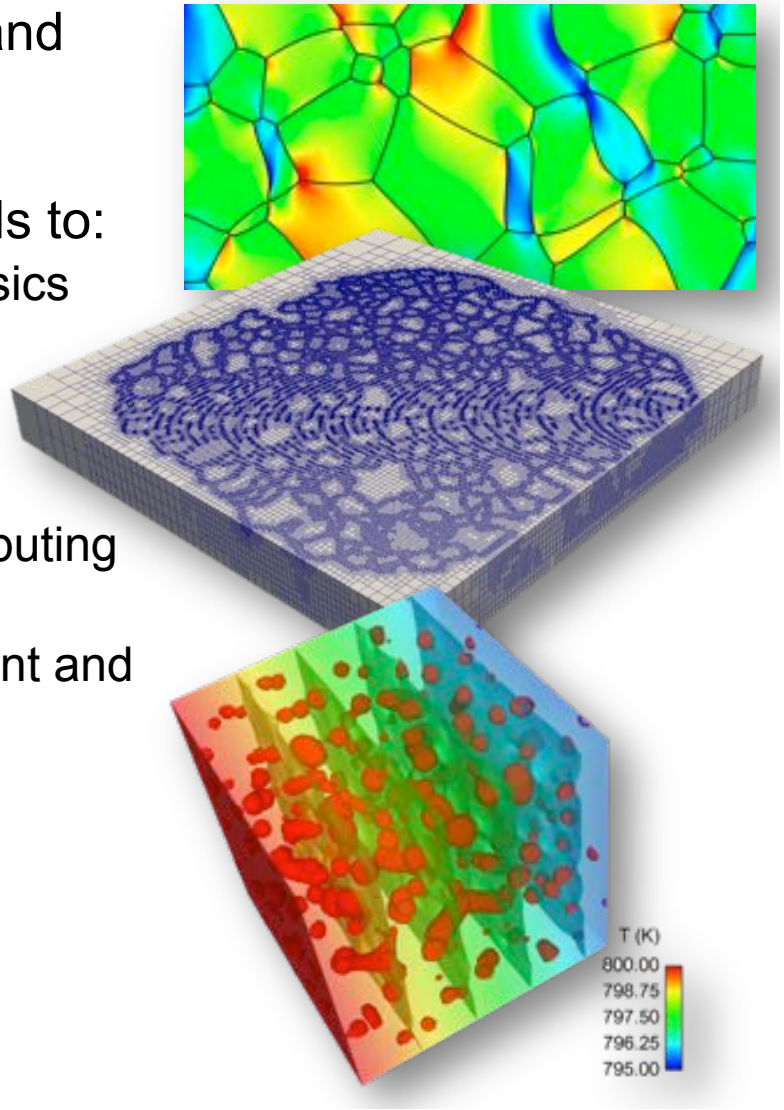
- Predictive modeling at the engineering scale

nm **µm** **mm**

Lengthscale

Materials Modeling Requirements

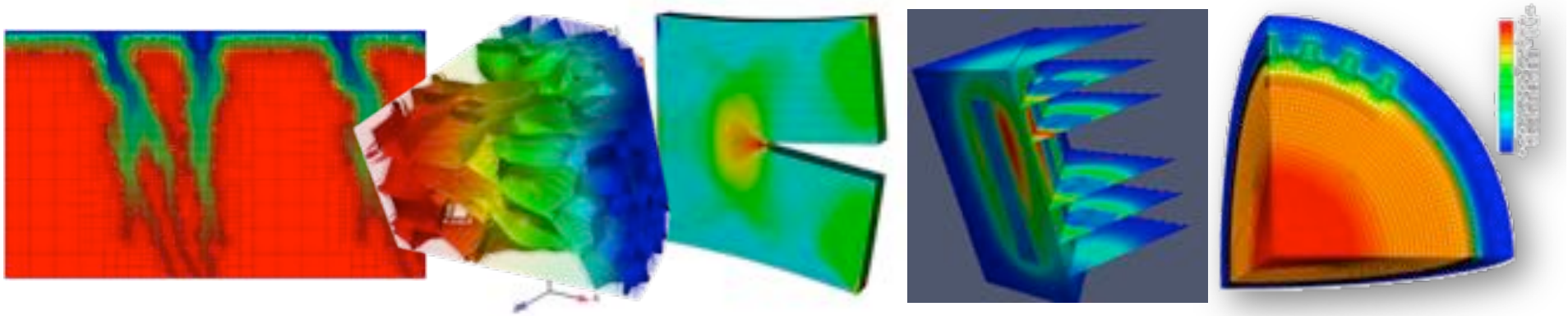
- To model material behavior at the meso- and macroscales requires that we deal with its inherent complexity.
- A tool for modeling material behavior needs to:
  - Easily handle multiple, tightly coupled physics
  - Have tools for multiscale modeling
- It would also be nice if it
  - Were simple to use and develop
  - Took advantage of high performance computing
  - Were free and open source
  - Had a team of full time staff for development and support
  - Had a strong user community



MOOSE

Multiphysics Object Oriented Simulation Environment

- MOOSE is a finite-element, multiphysics framework that simplifies the development of advanced numerical applications.
- It provides a high-level interface to sophisticated nonlinear solvers and massively parallel computational capability.

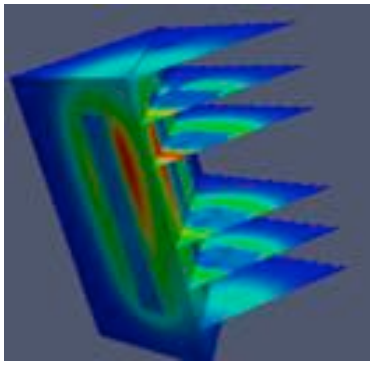


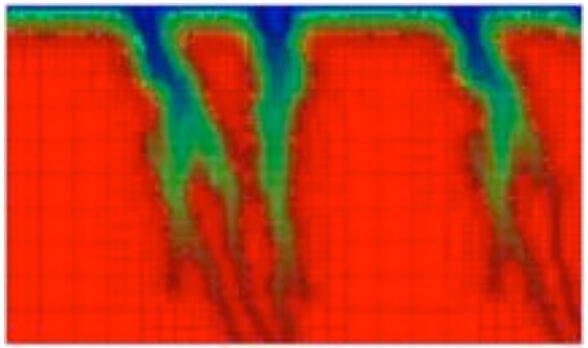
- MOOSE has been used to model thermomechanics, neutronics, geomechanics, reactive transport, microstructure modeling, computational fluid dynamics, and more every day!
- It is open source and freely available at mooseframework.org

MOOSE

- Tool for develop simulation tools that solve PDEs using FEM

- Spatial discretization with finite elements, where each variable can use a different element type, i.e. different shape functions
- Easy to couple multiple PDE
- Implicit or explicit time integration is available
- Dimension agnostic, same code can be used in 1- to 3-D
- Inherently parallel, solved with one to >10000 processors
- Provides access to mesh and time step adaptivity
- Easy simulation tool development
- Can read and write various mesh formats
- Strong user community
- Newton or Jacobian free solvers.



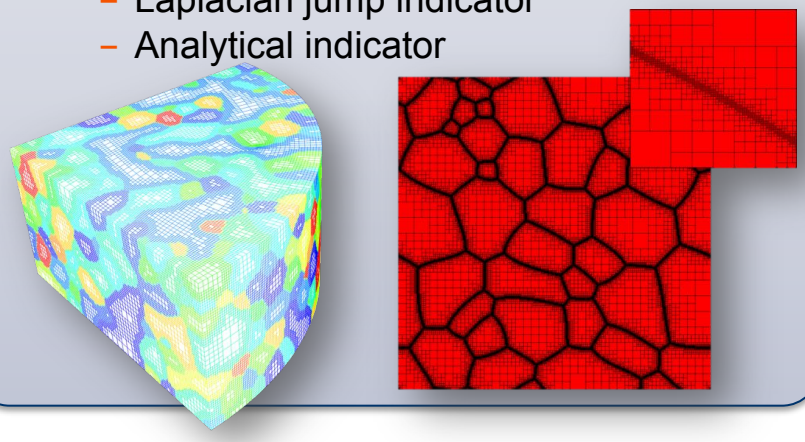


Mesh and Time Step Adaptivity

- Any model implemented with MOOSE has access to mesh and time step adaptivity

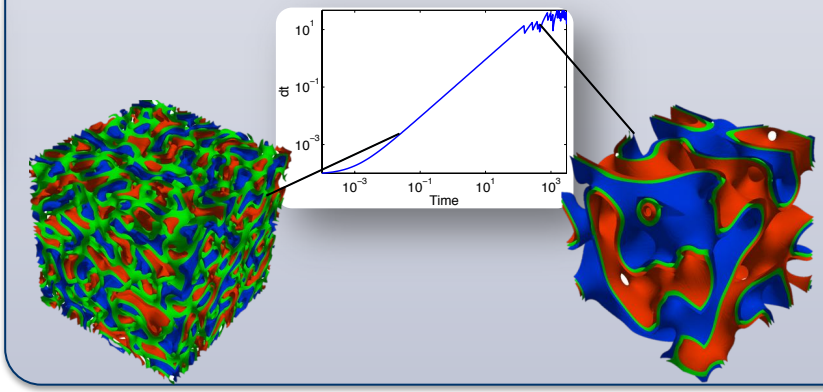
Mesh Adaptivity

- Requires no code development
- Refinement or coarsening is defined by a marker that be related to
  - An error estimator
  - Variable values
  - Stipulated by some other model
- Error indicators include the
  - Gradient jump indicator
  - Flux jump indicator
  - Laplacian jump indicator
  - Analytical indicator



Transient Time Step Adaptivity

- The time step in transient simulations can change with time
- Various time steppers exist to define dt:
  - Defined by a function
  - Adapts to maintain consistent solution behavior
  - Adapts to maintain consistent solution time
- Users can write new time steppers



Mesoscale Modeling with the MOOSE framework

- All of the code required to easily create your own phase field application is in the open source MOOSE modules (MOOSE-PF).

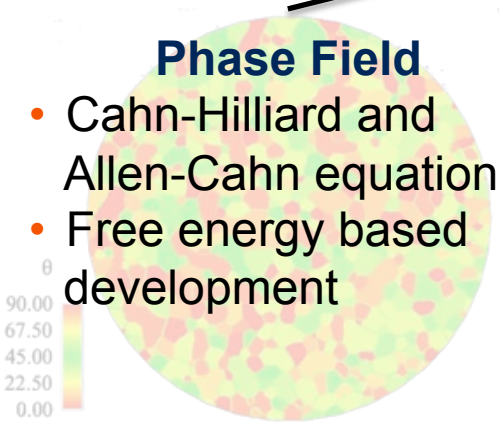
MOOSE

Framework

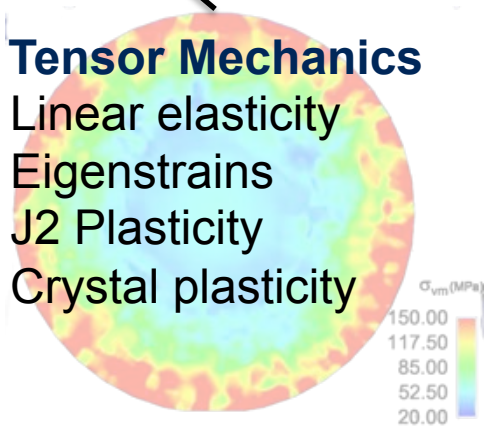
All of the code that forms the basis of the MOOSE framework

Modules

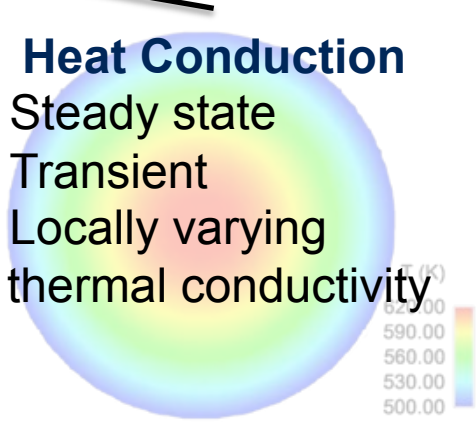
Phase Field
- Cahn-Hilliard and Allen-Cahn equations
- Free energy based development



Tensor Mechanics
- Linear elasticity
- Eigenstrains
- J2 Plasticity
- Crystal plasticity



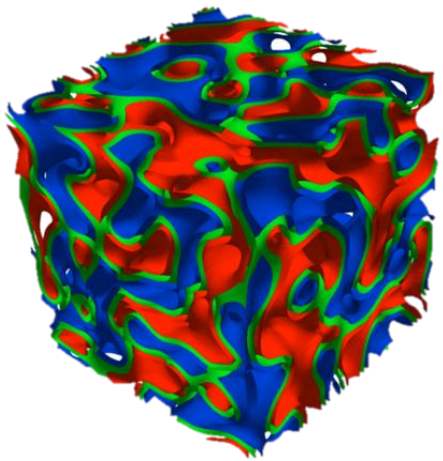
Heat Conduction
- Steady state
- Transient
- Locally varying thermal conductivity

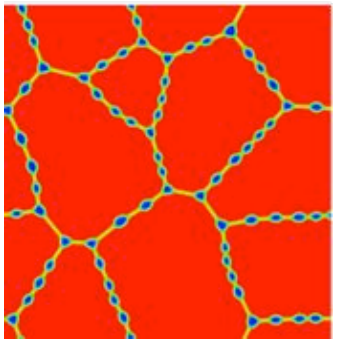


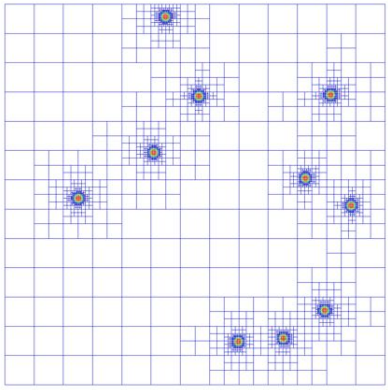
MOOSE-PF *Generic Phase Field Library*

- Provides the tools necessary to develop phase field models using FEM.

- Base classes for solving Cahn Hilliard equations
  - Direct solution
  - Split solution
- Base classes for Allen-Cahn equations
- Grain growth model
- Grain remapping algorithm for improved efficiency
- Initial conditions
- Postprocessors for characterizing microstructure



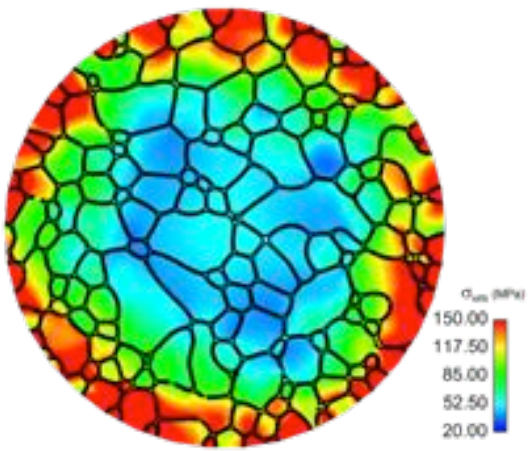


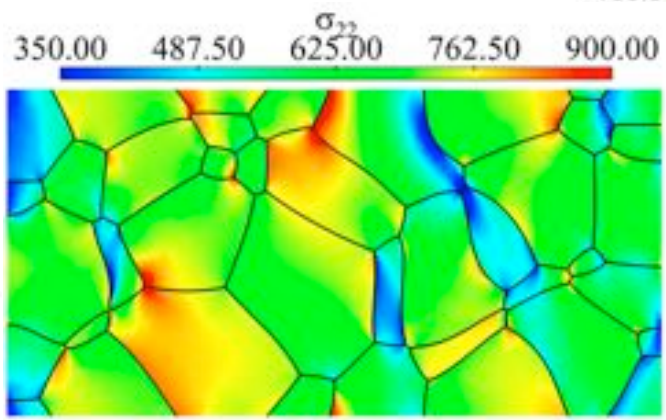


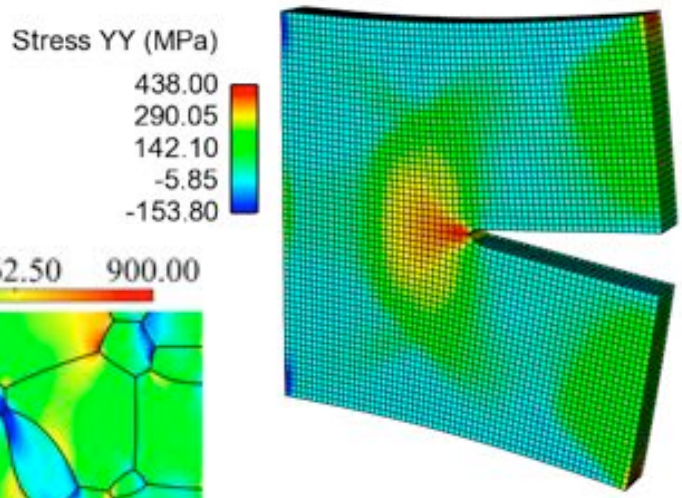
MOOSE-Tensor Mechanics

- Provides the tools necessary for modeling mechanical deformation and stress at the mesoscale.

- Anisotropic elasticity tensors that can change spatially
- Linear elasticity
- Eigen strains
- Finite strain mechanics
  - J2 plasticity
  - Crystal plasticity



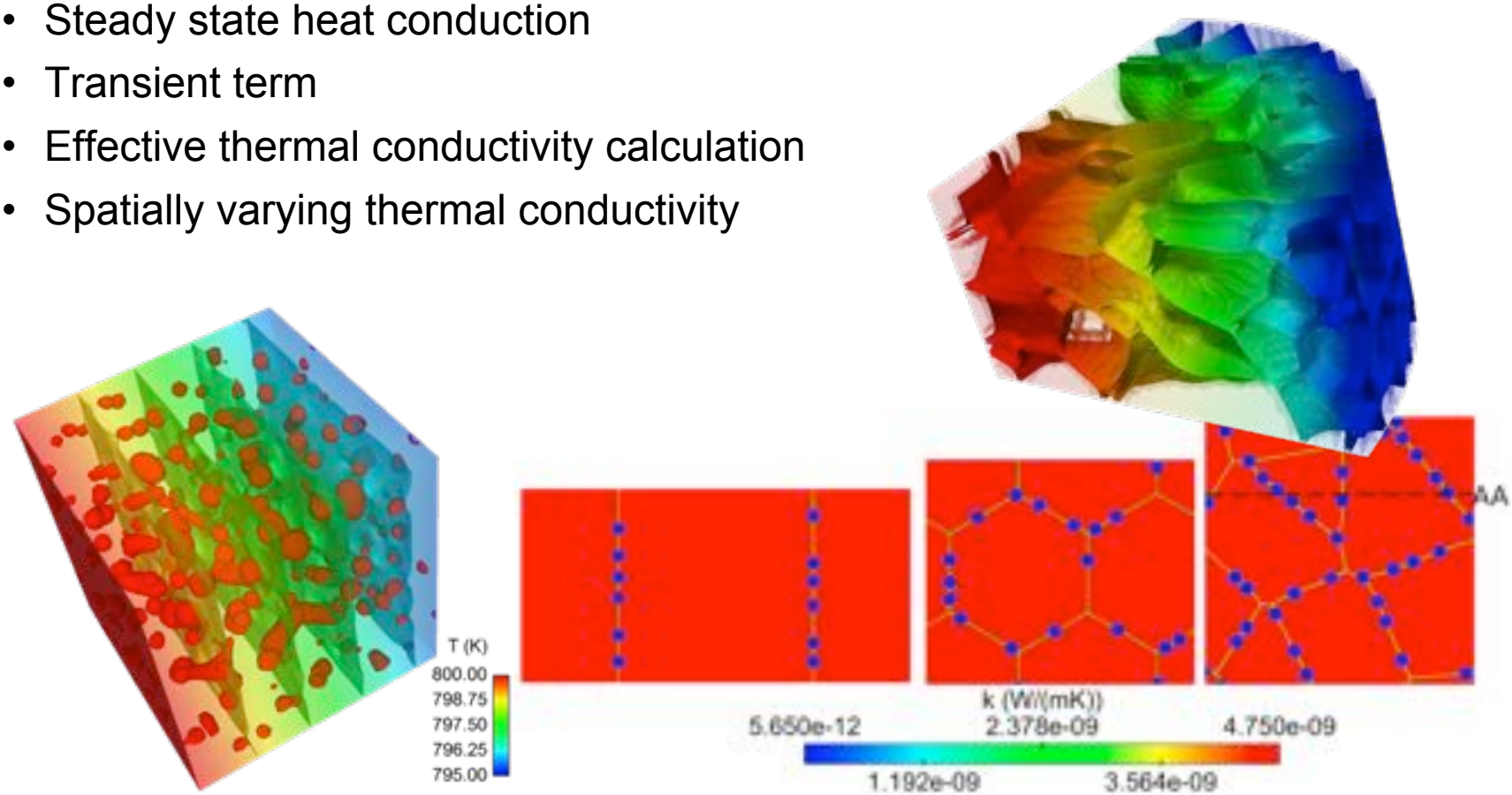




MOOSE-Heat Conduction

- Provides the tools necessary for modeling heat conduction and temperature gradients at the mesoscale.

- Steady state heat conduction
- Transient term
- Effective thermal conductivity calculation
- Spatially varying thermal conductivity



MARMOT

- Models the coevolution of microstructure and properties in reactor materials

No physics

Applicable to all materials

Specifically for reactor materials

- MARMOT is in use by researchers at laboratories and universities:

Overview of the Finite Element Method and Implementation

Polynomial Fitting

- To introduce the idea of finding coefficients to functions, let's consider simple polynomial fitting.
- In polynomial fitting (or interpolation) you have a set of points and you are looking for the coefficients to a function that has the form:

$$f(x) = a + bx + cx^2 + \ldots$$

- Where a, b and c are scalar coefficients and $1, x, x^2$ are "basis functions".
- Find a, b, c, etc. such that $f(x)$ passes through the points you are given.
- More generally you are looking for:

$$f(x) = \sum_{i=0}^{d} c_i x^i$$

  where the c_i are coefficients to be determined.
- $f(x)$ is unique and interpolary if $d + 1$ is the same as the number of points you need to fit.
- Need to solve a linear system to find the coefficients.

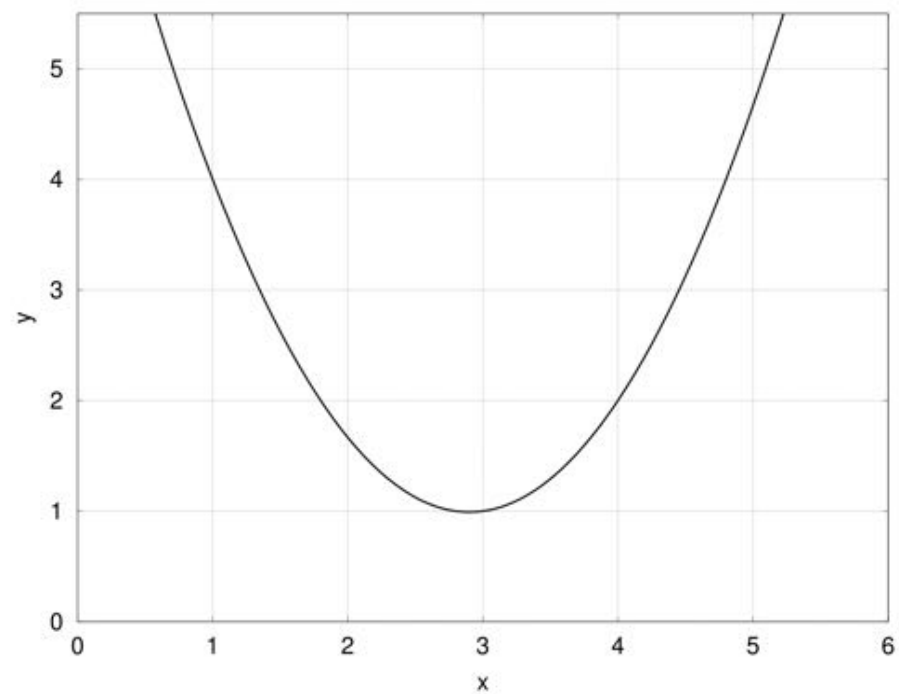
Example

1. Define a set of points:
   - $x = 1, 3, 4$
   - $y = 4, 1, 2$
2. Create the linear system:

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 3 & 9 \\ 1 & 4 & 16 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \\ 2 \end{bmatrix}$$

3. Solve for the coefficients:
   - $a = 8, b = \frac{29}{6}, c = \frac{5}{6}$
4. Define the complete solution function:

$$f(x) = 8 - \frac{29}{6}x + \frac{5}{6}x^2$$



Example (cont.)

- The coefficients themselves don't mean anything, by themselves they are just numbers.
- The solution is *not* the coefficients, but rather the *function* they create when they are multiplied by their respective basis functions and summed.
- The function $f(x)$ does go through the points we were given, *but it is also defined everywhere in between*.
- We can evaluate $f(x)$ at the point $x = 2$, for example, by computing:

$$f(2) = \sum_{i=0}^{2} c_i 2^i, \text{ or more generically: } f(2) = \sum_{i=0}^{2} c_i g_i(2),$$

  where the c_i correspond to the coefficients in the solution vector, and the g_i are the respective functions.
- Finally, note that the matrix consists of evaluating the functions at the points.

Finite Elements Simplified

- A method for numerically approximating the solution to Partial Differential Equations (PDEs).
- Works by finding a solution function that is made up of "shape functions" multiplied by coefficients and added together.
- Just like in polynomial fitting, except the functions aren't typically as simple as x^i (although they can be).
- The Galerkin Finite Element method is different from finite difference and finite volume methods because it finds a piecewise continuous function which is an approximate solution to the governing PDE.
- Just as in polynomial fitting you can evaluate a finite element solution anywhere in the domain.
- You do it the same way: by adding up "shape functions" evaluated at the point and multiplied by their coefficient.
- FEM is widely applicable for a large range of PDEs and domains.
- It is supported by a rich mathematical theory with proofs about accuracy, stability, convergence and solution uniqueness.

Weak Form

- Using FE to find the solution to a PDE starts with forming a "weighted residual" or "variational statement" or "weak form".
  - We typically refer to this process as generating a Weak Form.
- The idea behind generating a weak form is to give us some flexibility, both mathematically and numerically.
- A weak form is what you need to input into in order to solve a new problem.
- Generating a weak form generally involves these steps:
  1. Write down strong form of PDE.
  2. Rearrange terms so that zero is on the right of the equals sign.
  3. Multiply the whole equation by a "test" function ψ.
  4. Integrate the whole equation over the domain Ω.
  5. Integrate by parts (use the divergence theorem) to get the desired derivative order on your functions and simultaneously generate boundary integrals.

Refresher: The divergence theorem

- Transforms a volume integral into a surface integral:

$$\int_{\Omega} \nabla \cdot \vec{g} \, \mathrm{d}x = \int_{\partial\Omega} \vec{g} \cdot \hat{n} \, \mathrm{d}s$$

- In finite element calculations, for example with $\vec{g} = -k(x)\nabla u$, the divergence theorem implies:

$$-\int_{\Omega} \psi \left(\nabla \cdot k(x)\nabla u\right) \, \mathrm{d}x = \int_{\Omega} \nabla\psi \cdot k(x)\nabla u \, \mathrm{d}x - \int_{\partial\Omega} \psi \left(k(x)\nabla u \cdot \hat{n}\right) \, \mathrm{d}s$$

- We often use the following inner product notation to represent integrals since it is more compact:

$$-\left(\psi, \nabla \cdot k(x)\nabla u\right) = \left(\nabla\psi, k(x)\nabla u\right) - \left\langle \psi, k(x)\nabla u \cdot \hat{n}\right\rangle$$

- http://en.wikipedia.org/wiki/Divergence_theorem

Example: Convection Diffusion

- Write the strong form of the equation:

$$-\nabla \cdot k\nabla u + \vec{\beta} \cdot \nabla u = f$$

- Rearrange to zero is on the right-hand side:

$$-\nabla \cdot k\nabla u + \vec{\beta} \cdot \nabla u - f = 0$$

- Multiply by the test function ψ:

$$-\psi\left(\nabla \cdot k\nabla u\right) + \psi\left(\vec{\beta} \cdot \nabla u\right) - \psi f = 0$$

- Integrate over the domain Ω:

$$-\int_{\Omega} \psi\left(\nabla \cdot k\nabla u\right) + \int_{\Omega} \psi\left(\vec{\beta} \cdot \nabla u\right) - \int_{\Omega} \psi f = 0$$

Example: Convection Diffusion (cont.)

- Apply the divergence theorem to the diffusion term:

$$\int_{\Omega} \nabla\psi \cdot k\nabla u - \int_{\partial\Omega} \psi \left(k\nabla u \cdot \hat{n}\right) + \int_{\Omega} \psi \left(\vec{\beta} \cdot \nabla u\right) - \int_{\Omega} \psi f = 0$$

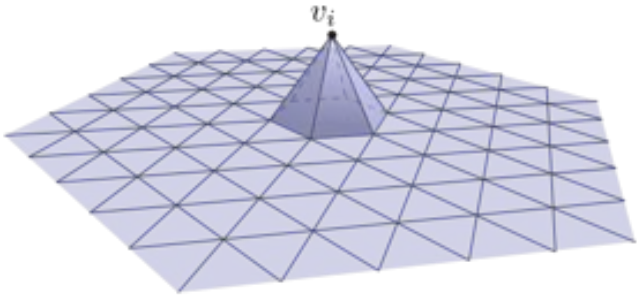
- Write in inner product notation, from which C++ code will be based. Each portion of the equation will inherit from an existing MOOSE type and the unique aspects of your equations defined.

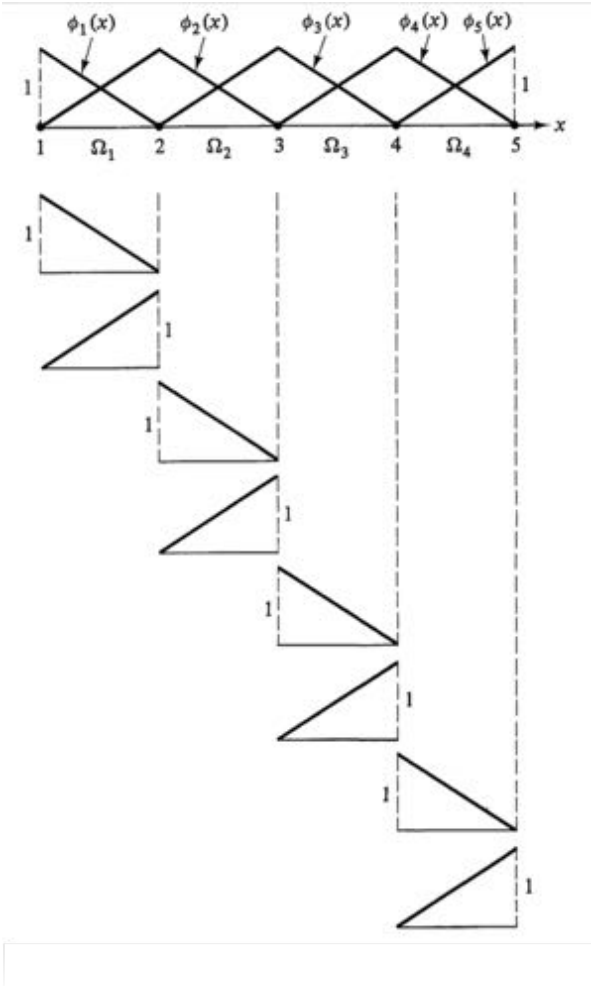
$$\underbrace{(\nabla\psi, k\nabla u)}_{Kernel} - \underbrace{\langle \psi, k\nabla u \cdot \hat{n}\rangle}_{BoundaryCondition} + \underbrace{\left(\psi, \vec{\beta} \cdot \nabla u\right)}_{Kernel} - \underbrace{(\psi, f)}_{Kernel} = 0$$

Finite Element Shape Functions

Basis Functions and Shape Functions

- While the weak form is essentially what you need for adding physics to MOOSE, in traditional finite element software more work is necessary.
- We need to discretize our weak form and select a set of simple "basis functions" amenable for manipulation by a computer.





Shape Functions

- Our discretized expansion of u takes on the following form:

$$u \approx u_h = \sum_{j=1}^{N} u_j \phi_j$$

  - The ϕ_j here are called "basis functions"
  - These ϕ_j form the basis for the "trial function", u_h
  - Analogous to the x^n we used earlier

- The gradient of u can be expanded similarly:

$$\nabla u \approx \nabla u_h = \sum_{j=1}^{N} u_j \nabla \phi_j$$

Shape Functions (cont.)

- In the Galerkin finite element method, the same basis functions are used for both the trial and test functions:

$$\psi = \{\phi_i\}_{i=1}^{N}$$

- Substituting these expansions back into our weak form, we get:

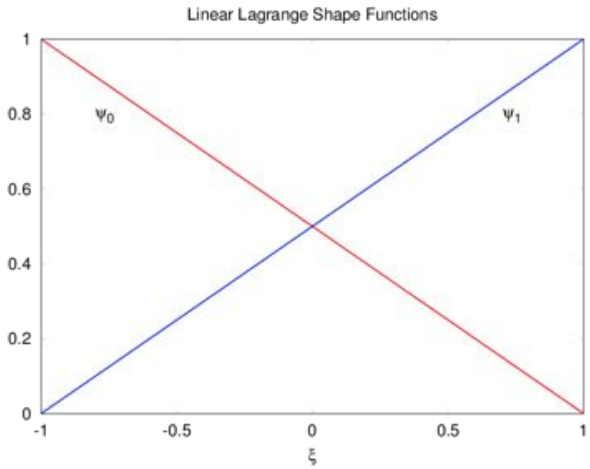
$$(\nabla\psi_i, k\nabla u_h) - \langle\psi_i, k\nabla u_h \cdot \hat{n}\rangle + \left(\psi_i, \vec{\beta} \cdot \nabla u_h\right) - (\psi_i, f) = 0, \quad i = 1, \ldots, N$$

- The left-hand side of the equation above is what we generally refer to as the ith component of our "Residual Vector" and write as $R_i(u_h)$.

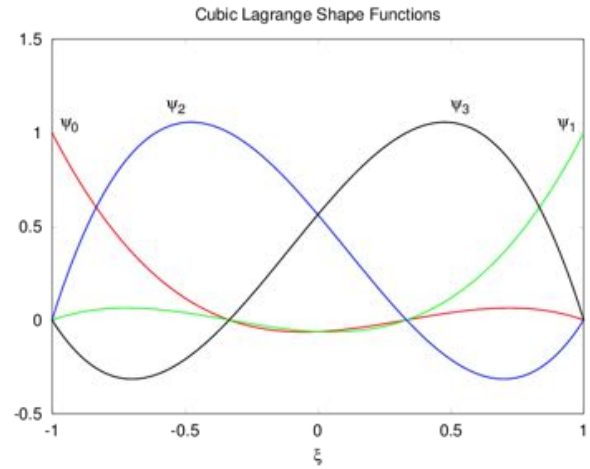
Shape Functions (cont.)

- Shape Functions are the functions that get multiplied by coefficients and summed to form the solution.
- Individual shape functions are finite pieces of the global basis functions.
- They are analogous to the x^n functions from polynomial fitting (in fact, you can use those as shape functions).
- Typical shape function families: Lagrange, Hermite, Hierarchic, Monomial, Clough-Toucher
  - MOOSE has support for all of these.
- Lagrange shape functions are the most common.
  - They are interpolary at the nodes, i.e., the coefficients correspond to the values of the functions at the nodes.

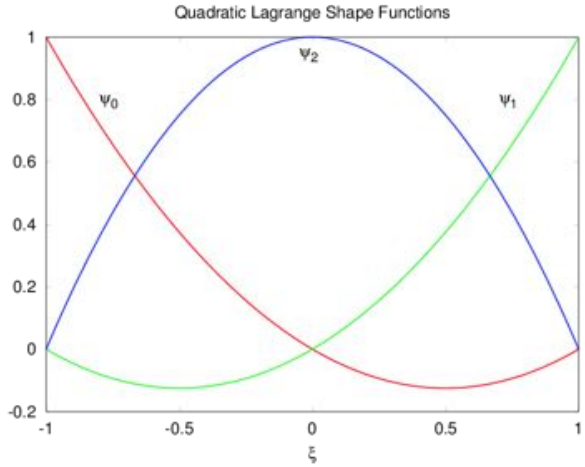
Example 1D Shape Functions

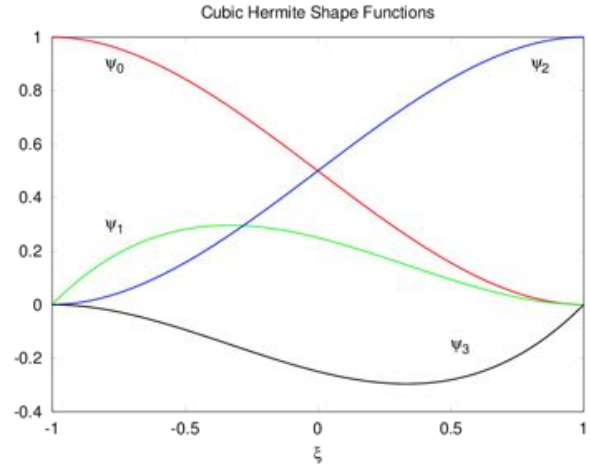


Linear Lagrange



Cubic Lagrange

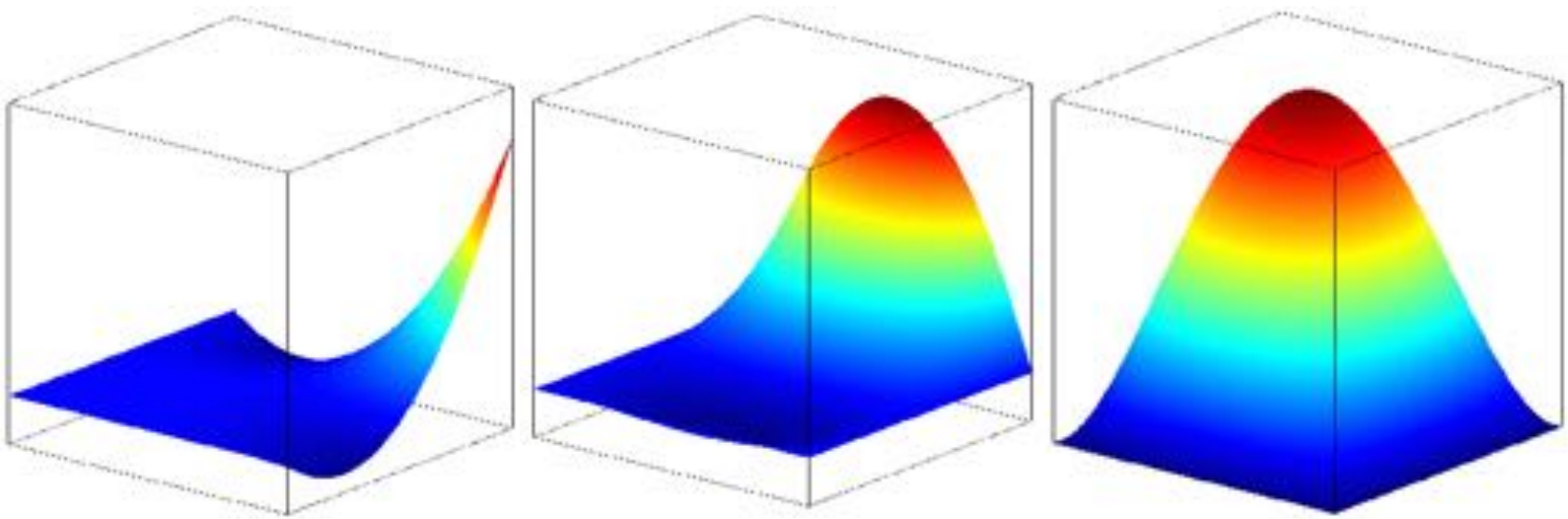




2D Lagrange Shape Functions

Example bi-quadratic basis functions defined on the Quad9 element:

- ψ_0 is associated to a "corner" node, it is zero on the opposite edges.
- ψ_4 is associated to a "mid-edge" node, it is zero on all other edges.
- ψ_8 is associated to the "center" node, it is symmetric and ≥ 0 on the element.



ψ_0 ψ_4 ψ_8

Numerical Implementation

Numerical Integration

- The only remaining non-discretized parts of the weak form are the integrals.
- We split the domain integral into a sum of integrals over elements:

$$\int_{\Omega} f(\vec{x})\ \mathrm{d}\vec{x} = \sum_{e} \int_{\Omega_e} f(\vec{x})\ \mathrm{d}\vec{x}$$

- Through a change of variables, the element integrals are mapped to integrals over the "reference" elements $\hat{\Omega}_e$.

$$\sum_{e} \int_{\Omega_e} f(\vec{x})\ \mathrm{d}\vec{x} = \sum_{e} \int_{\hat{\Omega}_e} f(\vec{\xi})\, |\mathcal{J}_e|\ \ \mathrm{d}\vec{\xi}$$

- $\mathcal{J}_e$ is the Jacobian of the map from the physical element to the reference element.

Numerical Integration (cont.)

- To approximate the reference element integrals numerically, we use quadrature (typically "Gaussian Quadrature"):

$$\sum_e \int_{\hat{\Omega}_e} f(\vec{\xi})\, |\mathcal{J}_e| \;\, \mathrm{d}\vec{\xi} \approx \sum_e \sum_{qp} w_{qp} f(\vec{x}_{qp})\, \left|\mathcal{J}_e(\vec{x}_{qp})\right|$$

- $\vec{x}_{qp}$ is the spatial location of the qpth quadrature point and w_{qp} is its associated associated weight.
- MOOSE handles multiplication by the Jacobian and the weight automatically, thus your `Kernel` is only responsible for computing the $f(\vec{x}_{qp})$ part of the integrand.
- Under certain common situations, the quadrature approximation is exact!
  - For example, in 1 dimension, Gaussian Quadrature can exactly integrate polynomials of order $2n-1$ with n quadrature points.

Numerical Integration (cont.)

- Note that sampling u_h at the quadrature points yields:

$$u(\vec{x}_{qp}) \approx u_h(\vec{x}_{qp}) = \sum u_j \phi_j(\vec{x}_{qp})$$
$$\nabla u(\vec{x}_{qp}) \approx \nabla u_h(\vec{x}_{qp}) = \sum u_j \nabla \phi_j(\vec{x}_{qp})$$

- And our weak form becomes:

$$R_i(u_h) = \sum_e \sum_{qp} w_{qp} \left|\mathcal{J}_e\right| \left[\nabla \psi_i \cdot k \nabla u_h + \psi_i \left(\vec{\beta} \cdot \nabla u_h\right) - \psi_i f\right] (\vec{x}_{qp})$$
$$- \sum_f \sum_{qp_{face}} w_{qp_{face}} \left|\mathcal{J}_f\right| \left[\psi_i k \nabla u_h \cdot \vec{n}\right] (\vec{x}_{qp_{face}})$$

- The second sum is over boundary faces, f.
- MOOSE `Kernels` must provide each of the terms in square brackets (evaluated at $\vec{x}_{qp}$ or $\vec{x}_{qp_{face}}$ as necessary).

Newton's Method

- We now have a nonlinear system of equations,

$$R_i(u_h) = 0, \qquad i = 1, \ldots, N$$

  to solve for the coefficients $u_j, j = 1, \ldots, N$.

- Newton's method has good convergence properties, we use it to solve this system of nonlinear equations.

- Newton's method is a "root finding" method: it finds zeros of nonlinear equations.
- Newton's Method in "Update Form" for finding roots of the scalar equation $f(x) = 0, f(x) : \mathbb{R} \to \mathbb{R}$ is given by

$$\begin{aligned} f'(x_n)\delta x_{n+1} &= -f(x_n) \\ x_{n+1} &= x_n + \delta x_{n+1} \end{aligned}$$

Newton's Method (cont.)

- We don't have just one scalar equation: we have a system of nonlinear equations.
- This leads to the following form of Newton's Method:

$$\begin{aligned}\mathbf{J}(\vec{u}_n)\delta\vec{u}_{n+1} &= -\vec{R}(\vec{u}_n)\\ \vec{u}_{n+1} &= \vec{u}_n + \delta\vec{u}_{n+1}\end{aligned}$$

- Where $\mathbf{J}(\vec{u}_n)$ is the Jacobian matrix evaluated at the current iterate:

$$J_{ij}(\vec{u}_n) = \frac{\partial R_i(\vec{u}_n)}{\partial u_j}$$

- Note that:

$$\frac{\partial u_h}{\partial u_j} = \sum_k \frac{\partial}{\partial u_j}(u_k\phi_k) = \phi_j \qquad \frac{\partial\left(\nabla u_h\right)}{\partial u_j} = \sum_k \frac{\partial}{\partial u_j}(u_k\nabla\phi_k) = \nabla\phi_j$$

Newton for a Simple Equation

- Consider the convection-diffusion equation with nonlinear k, $\vec{\beta}$, and f:

$$-\nabla \cdot k\nabla u + \vec{\beta} \cdot \nabla u = f$$

- The ith component of the residual vector is:

$$R_i(u_h) = (\nabla\psi_i, k\nabla u_h) - \langle \psi_i, k\nabla u_h \cdot \hat{n}\rangle + \left(\psi_i, \vec{\beta} \cdot \nabla u_h\right) - (\psi_i, f)$$

Newton for a Simple Equation (cont.)

- Using the previously-defined rules for $\frac{\partial u_h}{\partial u_j}$ and $\frac{\partial(\nabla u_h)}{\partial u_j}$, the (i,j) entry of the Jacobian is then:

$$J_{ij}(u_h) = \left(\nabla\psi_i, \frac{\partial k}{\partial u_j}\nabla u_h\right) + \left(\nabla\psi_i, k\nabla\phi_j\right) - \left\langle \psi_i, \frac{\partial k}{\partial u_j}\nabla u_h \cdot \hat{n}\right\rangle$$
$$- \left\langle \psi_i, k\nabla\phi_j \cdot \hat{n}\right\rangle + \left(\psi_i, \frac{\partial\vec{\beta}}{\partial u_j} \cdot \nabla u_h\right) + \left(\psi_i, \vec{\beta}\cdot\nabla\phi_j\right) - \left(\psi_i, \frac{\partial f}{\partial u_j}\right)$$

- Note that even for this "simple" equation, the Jacobian entries are nontrivial: they depend on the partial derivatives of k, $\vec{\beta}$, and f, which may be difficult or time-consuming to compute analytically.

- In a multiphysics setting with many coupled equations and complicated material properties, the Jacobian might be extremely difficult to determine.

Chain Rule

- On the previous slide, the term $\frac{\partial f}{\partial u_j}$ was used, where f was a nonlinear forcing function.
- The chain rule allows us to write this term as

$$\begin{aligned}\frac{\partial f}{\partial u_j} &= \frac{\partial f}{\partial u_h}\frac{\partial u_h}{\partial u_j} \\ &= \frac{\partial f}{\partial u_h}\phi_j\end{aligned}$$

- If a functional form of f is known, e.g. $f(u) = \sin(u)$, this formula implies that its Jacobian contribution is given by

$$\frac{\partial f}{\partial u_j} = \cos(u_h)\phi_j$$

Jacobian Free Newton Krylov

- $\mathbf{J}(\vec{u}_n)\delta\vec{u}_{n+1} = -\vec{R}(\vec{u}_n)$ is a linear system solved during each Newton step.
- For simplicity, we can write this linear system as $\mathbf{A}\vec{x} = \vec{b}$, where:
  - $\mathbf{A} \equiv \mathbf{J}(\vec{u}_n)$
  - $\vec{x} \equiv \delta\vec{u}_{n+1}$
  - $\vec{b} \equiv -\vec{R}(\vec{u}_n)$
- We employ an iterative Krylov method (e.g. GMRES) to produce a sequence of iterates $\vec{x}_k \to \vec{x}, k = 1, 2, \ldots$
- $\mathbf{A}$ and $\vec{b}$ remain *fixed* during the iterative process.
- The "linear residual" at step k is defined as

$$\rho_k \equiv \mathbf{A}\vec{x}_k - \vec{b}$$

- MOOSE prints the norm of this vector, $\|\rho_k\|$, at each iteration, if you set `print_linear_residuals = true` in the `Outputs` block.
- The "nonlinear residual" printed by MOOSE is $\|\vec{R}(\vec{u}_n)\|$.

Jacobian Free Newton Krylov (cont.)

- By iterate k, the Krylov method has constructed the subspace

$$\mathcal{K}_k = \text{span}\{\vec{b}, \mathbf{A}\vec{b}, \mathbf{A}^2\vec{b}, \ldots, \mathbf{A}^{k-1}\vec{b}\}$$

- Different Krylov methods produce the $\vec{x}_k$ iterates in different ways:
  - Conjugate Gradients: $\vec{\rho}_k$ orthogonal to $\mathcal{K}_k$.
  - GMRES/MINRES: $\vec{\rho}_k$ has minimum norm for $\vec{x}_k$ in $\mathcal{K}_k$.
  - Biconjugate Gradients: $\vec{\rho}_k$ is orthogonal to $\mathcal{K}_k(\mathbf{A}^T)$
- $\mathbf{J}$ is never explicitly needed to construct the subspace, only the action of $\mathbf{J}$ on a vector is required.

Jacobian Free Newton Krylov (cont.)

- This action can be approximated by:

$$\mathbf{J}\vec{v} \approx \frac{\vec{R}(\vec{u} + \epsilon\vec{v}) - \vec{R}(\vec{u})}{\epsilon}$$

- This form has many advantages:
  - No need to do analytic derivatives to form $\mathbf{J}$
  - No time needed to compute $\mathbf{J}$ (just residual computations)
  - No space needed to store $\mathbf{J}$

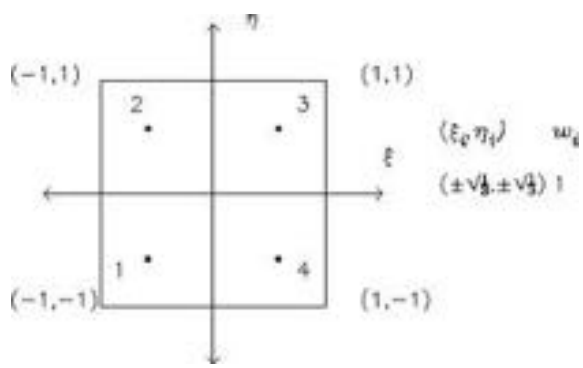
Wrap Up

- The Finite Element Method is a way of numerically approximating the solution of PDEs.
- Just like polynomial fitting, FEM finds coefficients for basis functions.
- The "solution" is the combination of the coefficients and the basis functions, and the solution can be sampled anywhere in the domain.
- We compute integrals numerically using quadrature.
- Newton's Method provides a mechanism for solving a system of nonlinear equations.
- The Jacobian Free Newton Krylov (JFNK) method allows us to avoid explicitly forming the Jacobian matrix while still computing its "action".

Code Implementation

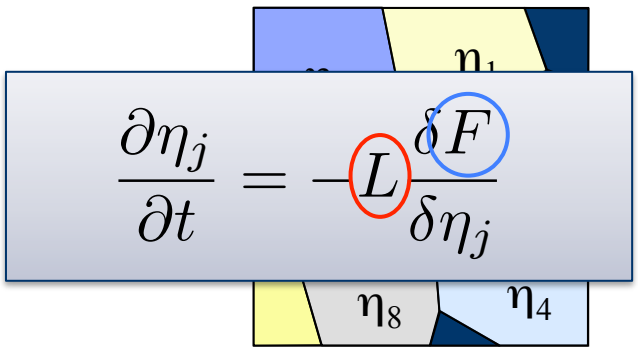
- FEM can be implemented by hand, but can be fairly complicated.
- Many commercial FEM codes exist, but they are expensive and are often not very flexible for solving multiphysics problems.
- Open source options for FEM exist
- Will demonstrate solving this thermo/mechanical system in MOOSE

Equations	Code
$\mathcal{R}_T = \int_\Omega k \nabla T \cdot \nabla \phi_i \, dV$	`_k[_qp] * _grad_u[_qp] * _grad_phi[_qp][_j]`
$\mathcal{R}_{\mathbf{u}} = \int_\Omega \boldsymbol{\sigma} \cdot \nabla \phi_i \, dV$	`_stress[_qp] * _grad_phi[_qp][_j]`

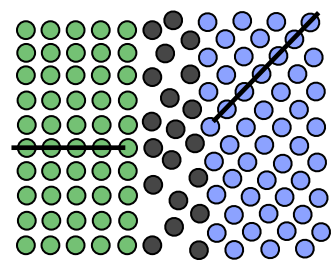


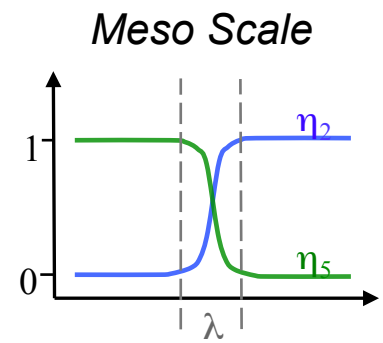
The Phase Field Method

- Microstructure described by a set of continuous variables…
  - Non-Conserved Order Parameters

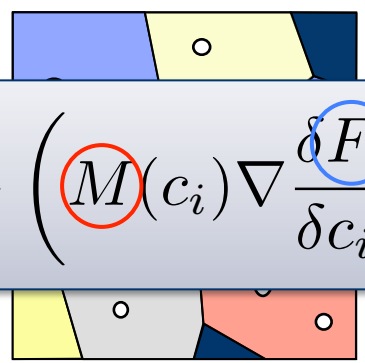


$$\frac{\partial \eta_j}{\partial t} = -L \frac{\delta F}{\delta \eta_j}$$

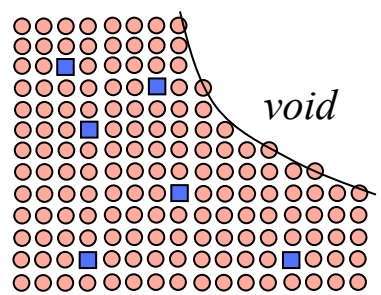


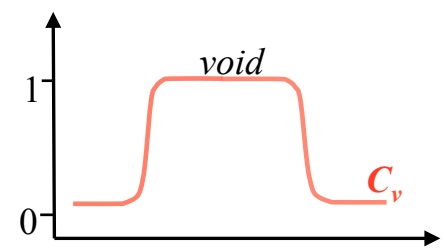


  - Conserved Concentrations



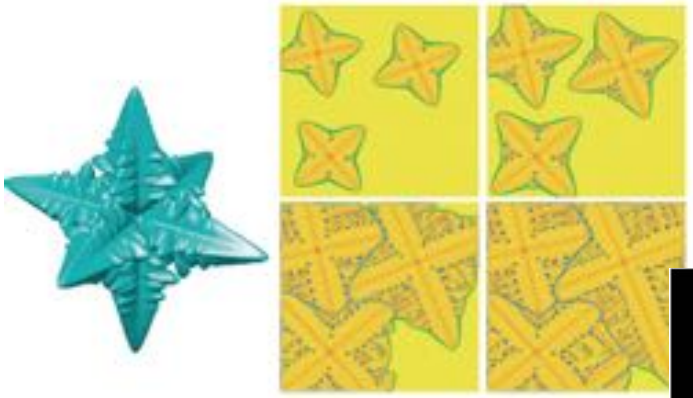
$$\frac{\partial c_i}{\partial t} = \nabla \cdot \left(M(c_i) \nabla \frac{\delta F}{\delta c_i} \right)$$



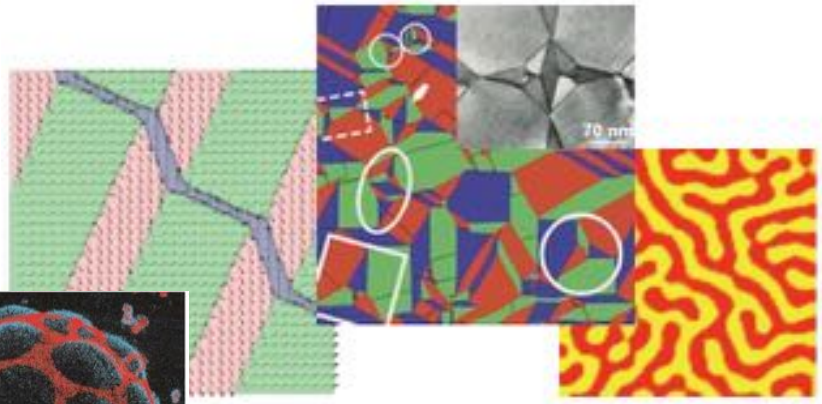


- The variables evolve to minimize a functional defining the free energy

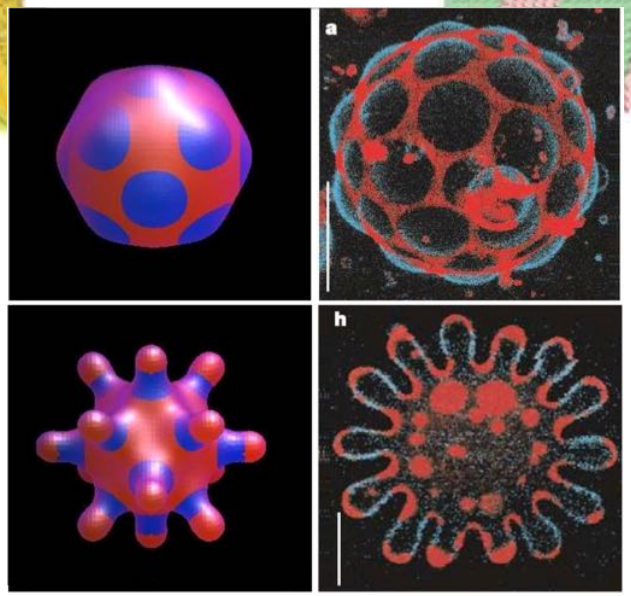
Phase Field Has Been Used in Many Areas

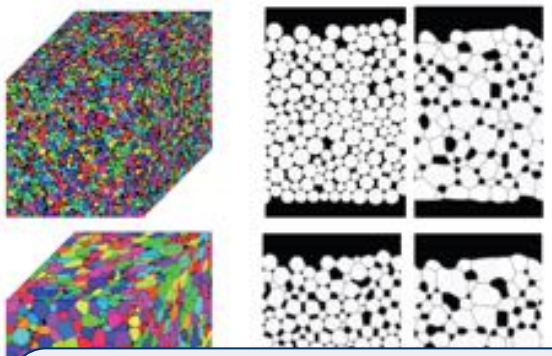


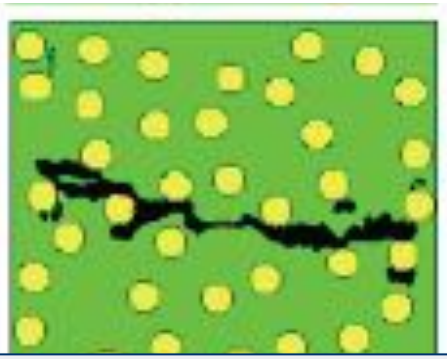
solidification (dendrite growth)



phase transformations



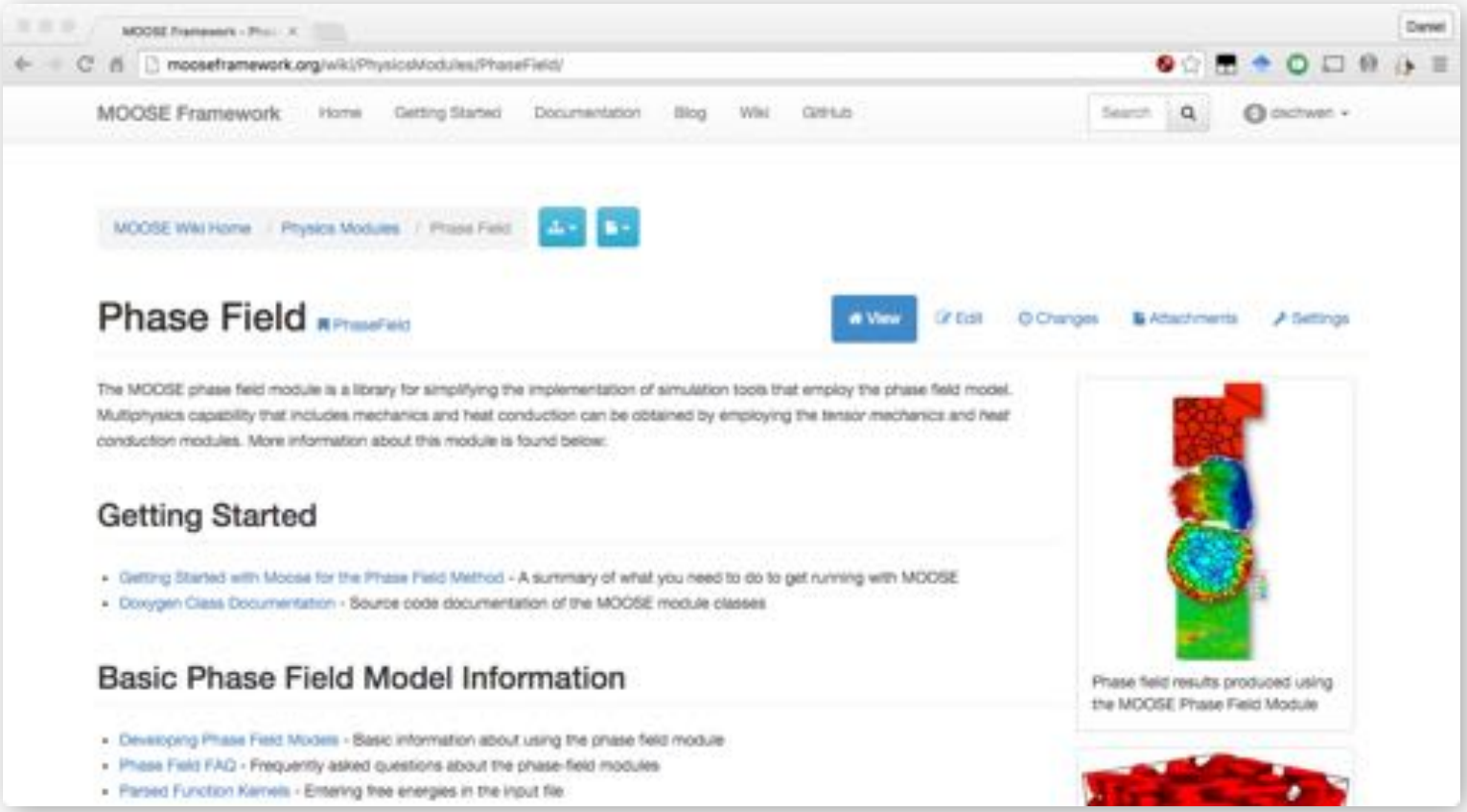




- The phase field method is our method of choice because it can be:
  - Easily coupled to other physics such as mechanics or heat conduction
  - Quantitative and can represent real materials

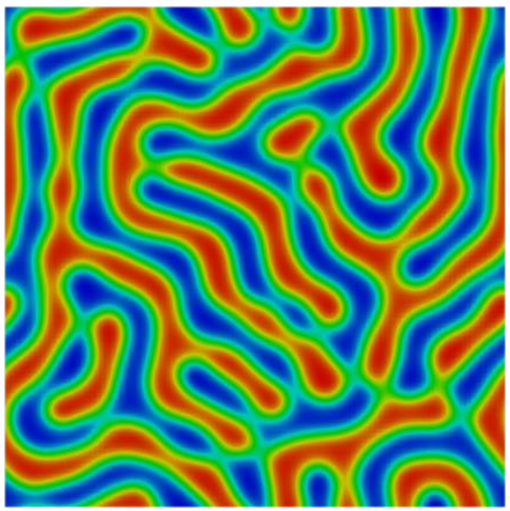
Phase Field Documentation

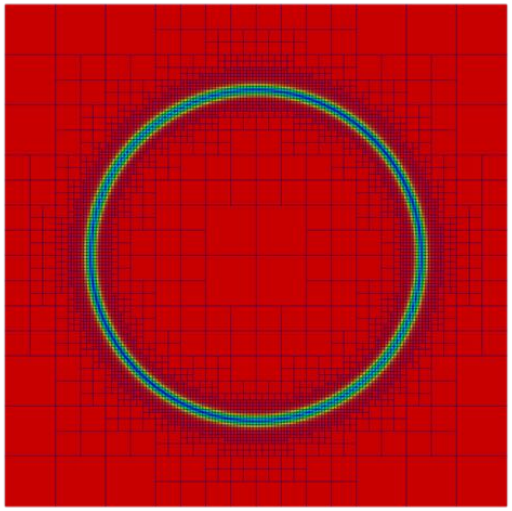
- Documentation for the phase field module is found on the mooseframework.org wiki:
  - http://mooseframework.org/wiki/PhysicsModules/PhaseField/



Examples

- Example input files for MOOSE-PF can be found in the examples directory in each project folder.
  - These are midsized 2D problems that run well on four processors





- The tests can serve as additional examples
  - There are many tests for the various components of MOOSE
  - Each test runs in less then 2 seconds on one processor

The Phase Field Equations

- Non-conserved variables (phases, grains, etc.) are evolved using an **Allen-Cahn** (aka Ginzburg-Landau) type equation:

$$\frac{\partial \eta_j}{\partial t} = -L \frac{\delta F}{\delta \eta_j}$$

- Conserved variables are evolved using a **Cahn-Hilliard** type equation:

$$\frac{\partial c_i}{\partial t} = \nabla \cdot \left(M(c_i) \nabla \frac{\delta F}{\delta c_i} \right)$$

- Both equations are functions of *variational derivatives* of a functional defining the free energy of the system in terms of the variables

$$F = \int_V \left(\underbrace{f_{loc}(c_i, \eta_j, ..., T) + E_d}_{\text{Local energy}} + \underbrace{\sum_i \frac{\kappa_i}{2} (\nabla c_i)^2 + \sum_j \frac{\kappa_j}{2} (\nabla \eta_j)^2}_{\text{Gradient energy}} \right) dV$$

Variational Derivative

The functional derivative (or variational derivative) relates a change in a functional to a change in a function that the functional depends on.

Wikipedia, "Functional derivative"

$$F = \int f(r, c, \nabla c)\, dV$$

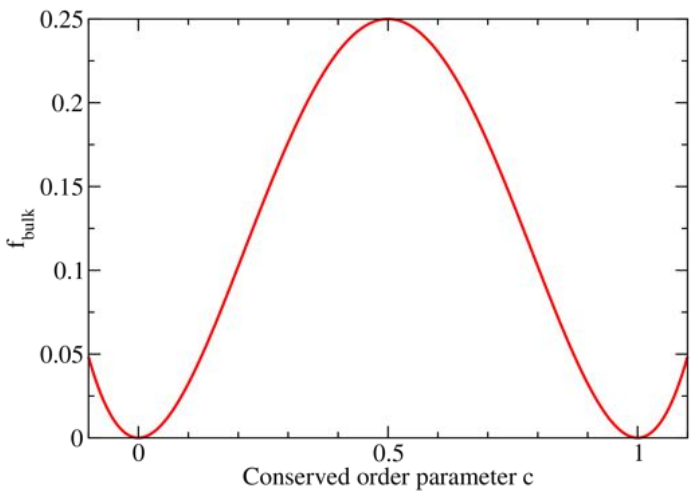
$$\frac{\delta F}{\delta c} = \frac{\partial f}{\partial c} - \nabla \cdot \frac{\partial f}{\partial \nabla c}$$

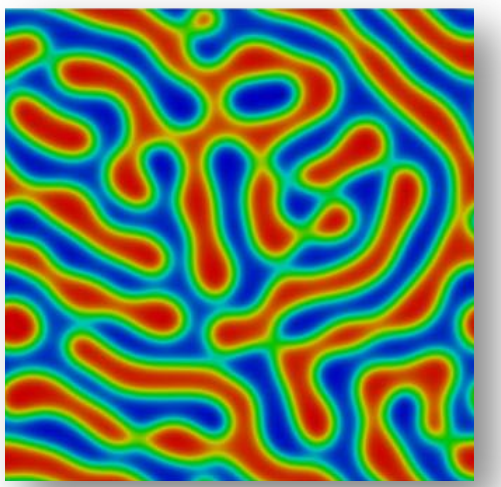
F	total free energy
f	free energy **density**

- Derivative with respect to the **gradient!**
- Gradient energy term in phase field (very few functional forms)
- Bulk free energy (contains the thermodynamics of the system)
  - Simple partial derivative

Phase Field Implementation in MOOSE

- The kernels required to solve the phase field equations have been implemented in the phase field module
- In general, a developer will not need to change the kernels but simply use the kernels that have already been implemented
- New models are implemented by defining the free energy and mobility with their derivatives in *material* objects.





Derivative Function Materials

- Each MOOSE Material class can provide **multiple Material Properties**

- A Derivative Function Material is a MOOSE Material class that provides a well defined set of Material Properties

  - A function value, stored in the material property `F` (the `f_name` of the Material)
  - All derivatives of `F` up to a given order with respect to the non-linear variables `F` depends on

- The derivatives are regular Material Properties with an enforced naming convention

  - Example `F`, `dF/dc`, `d^2F/dc^2`, `dF/deta`, `d^2F/dcdeta` ...
  - You don't need to know the property names besides `F`, unless you want to look at them in the output!

- Recap:
  Each Derivative Function Material provides **one** Function **together** with its derivatives!

- That function can be a *Free Energy Density*, a *Mobility*, or whatever you may need.

Solving the Allen-Cahn Equation

- After taking the variational derivative, the strong form of the Allen-Cahn residual equation is

$$\frac{\partial \eta_j}{\partial t} = -L\left(\frac{\partial F}{\partial \eta_j} + \frac{\partial E_d}{\partial \eta_j} - \kappa_j \nabla^2 \eta_j\right)$$

- Each piece of the weak form of the residual equation has been implemented in a kernel:

$$\mathcal{R}_{\eta_j} = \underbrace{\left(\frac{\partial \eta_j}{\partial t}, \psi_m\right)}_{\texttt{TimeDerivative}} + \underbrace{(L_j \kappa_j \nabla \eta_j, \nabla \psi_m)}_{\texttt{ACInterface}} + \underbrace{L_j\left(\frac{\partial f_{loc}}{\partial \eta_j} + \frac{\partial E_d}{\partial \eta_j}, \psi_m\right)}_{\texttt{AllenCahn}}$$

- Parameters must be defined in a *material* object
- The free energy density and its derivatives are defined in a Derivative Function Material

Solving the Cahn-Hilliard Equation

- Due to the fourth-order derivative, solving the Cahn-Hilliard equation can be hard. In MOOSE there are two available approaches
  - Residual: $\mathcal{R}_{c_i} = \frac{\partial c_i}{\partial t} - \nabla \cdot M(c_i)\left(\nabla \frac{\partial f_{loc}}{\partial c_i} + \nabla \frac{\partial E_d}{\partial c_i}\right) + \nabla \cdot M(c_i)\nabla\left(\kappa_i \nabla^2 c_i\right)$
  - We can put this in weak form:

$$\left(\frac{\partial c_i}{\partial t}, \psi_m\right) = -(\kappa_i \nabla^2 c_i, \nabla \cdot (M_i \nabla \psi_m)) - \left(M_i \nabla \left(\frac{\partial f_{loc}}{\partial c_i} + \frac{\partial E_d}{\partial c_i}\right), \nabla \psi_m\right)$$

  - But, solving this residual requires **higher order elements**

- Another option is to split the equation into two:

Strong Form

$$\frac{\partial c_i}{\partial t} = \nabla \cdot (M_i \nabla \mu_i)$$

$$\mu_i = \frac{\partial f_{loc}}{\partial c_i} - \kappa_i \nabla^2 c_i + \frac{\partial E_d}{\partial c_i}$$

Weak Form

$$\left(\frac{\partial c_i}{\partial t}, \psi_m\right) = -\left(M_i \nabla \mu_i, \nabla \psi_m\right)$$

$$(\mu_i, \psi_m) = \left(\frac{\partial f_{loc}}{\partial c_i}, \psi_m\right) + (\kappa_i \nabla c_i, \nabla \psi_m) + \left(\frac{\partial E_d}{\partial c_i}, \psi_m\right)$$

  - The split form can be solved with **first-order elements**.

The Direct Solution of the Cahn-Hilliard Equation

- Each piece of the weak form of the Cahn-Hilliard residual equation has been implemented in a kernel

$$\mathcal{R}_{c_i} = \underbrace{\left(\frac{\partial c_i}{\partial t}, \psi_m\right)}_{\texttt{TimeDerivative}} + \underbrace{\left(\kappa_i \nabla^2 c_i, \nabla \cdot (M_i \nabla \psi_m)\right)}_{\texttt{CHInterface}} + \underbrace{\left(M_i \nabla \left(\frac{\partial f_{loc}}{\partial c_i} + \frac{\partial E_d}{\partial c_i}\right), \nabla \psi_m\right)}_{\texttt{CahnHilliard}}$$

- Parameters must be defined in a material object
- The free energy density and its derivatives are defined in an energy material object (e.g. `DerivativeParsedMaterial`)
- Mobilities can also depend on non-linear variables M(c) and can be supplied through Derivative Function Materials
- Due to the second order derivative, third order Hermite elements must be used to discretize the variables

The Split Solution of the Cahn-Hilliard Equation

- The weak form of the split Cahn-Hilliard residual equation has also been implemented in kernels:

$$\mathcal{R}_{\mu_i} = \left(\frac{\partial c_i}{\partial t}, \psi_m\right) + (M_i \nabla \mu_i, \nabla \psi_m)$$

`CoupledTimeDerivative` `SplitCHWRes`

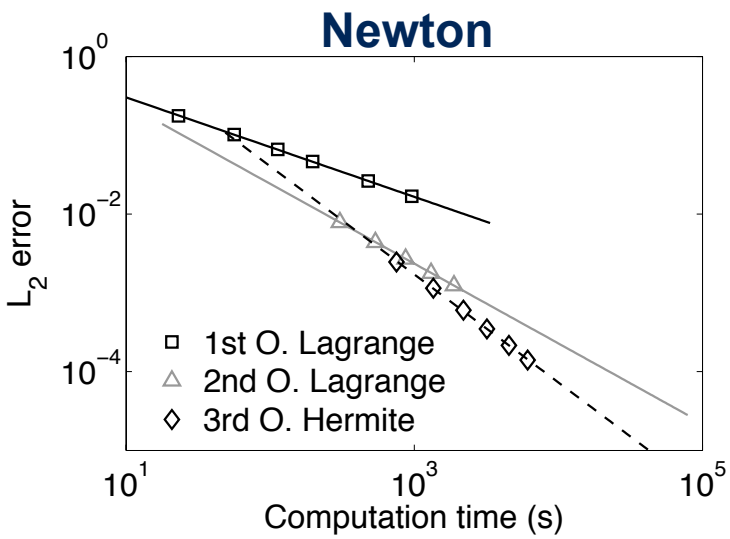
$$\mathcal{R}_{c_i} = (\kappa_i \nabla c_i, \nabla \psi_m) + \left(\left(\frac{\partial f_{loc}}{\partial c_i} + \frac{\partial E_d}{\partial c_i} - \mu_i\right), \psi_m\right)$$

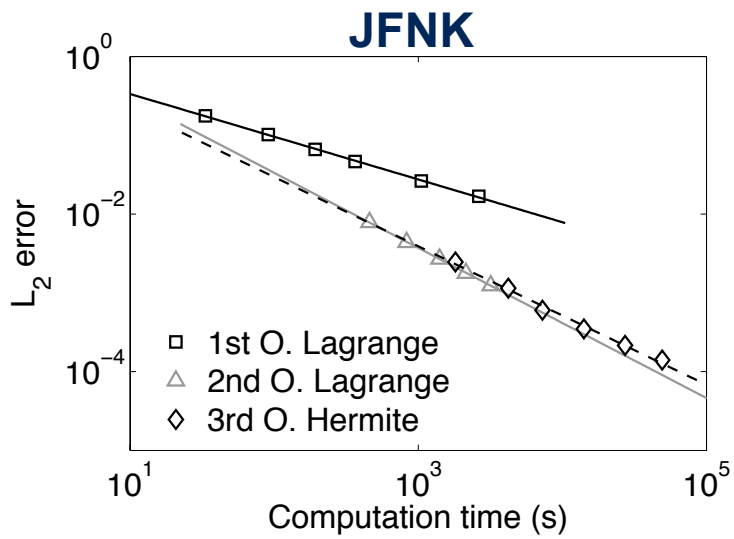
`SplitCHParsed`

- Parameters must be defined in a material object
- The free energy density and its derivatives are defined in an energy material object (as with the direct solve, making it easy to switch between the two)
- Residuals are reversed to improve convergence (`CoupledTimeDerivative`)

Cahn-Hilliard Solution

- We have done a quantitative comparison between the direct and the split solutions of the Cahn-Hilliard equation.
  - The split with 1st order elements is the most efficient.
  - The direct solution has the least error.





- However, practically speaking the split is often the best choice, since our simulations can be computationally expensive.

Simple Phase Field Model Development

- As stated above, the microstructure evolves to minimize the free energy
- Thus, the free energy functional is the major piece of the model
- Phase field model development is modular, with all development focused around the free energy

Free energy: $$F = \int_V \left(f_{loc}(c_i, \eta_j, ..., T) + E_d + \sum_i \frac{\kappa_i}{2}(\nabla c_i)^2 + \sum_j \frac{\kappa_j}{2}(\nabla \eta_j)^2 \right) dV$$

Differential equations: $$\left(M_i \nabla \left(\frac{\partial f_{loc}}{\partial c_i} + \frac{\partial E_d}{\partial c_i} \right), \nabla \psi_m \right) \quad (\kappa_i \nabla c_i, \nabla \psi_m) + \left(\left(\frac{\partial f_{loc}}{\partial c_i} + \frac{\partial E_d}{\partial c_i} - \mu_i \right), \psi_m \right)$$

CahnHilliard

SplitCHParsed

```
Free Energy Density Material
fbulk = 1/4*(1 + c)^2*(1 - c)^2
dfbulk/dc      = c^3 - c
d^2fbulk/dc^2 = 3*c^2 - 1
d^3fbulk/dc^3 = 6*c
```

Reminder:

$$\nabla f(c,\eta) = \nabla c \frac{\partial f}{\partial c} + \nabla \eta \frac{\partial f}{\partial \eta}$$

Phase field models that are not based on a free energy can be implemented using normal MOOSE syntax

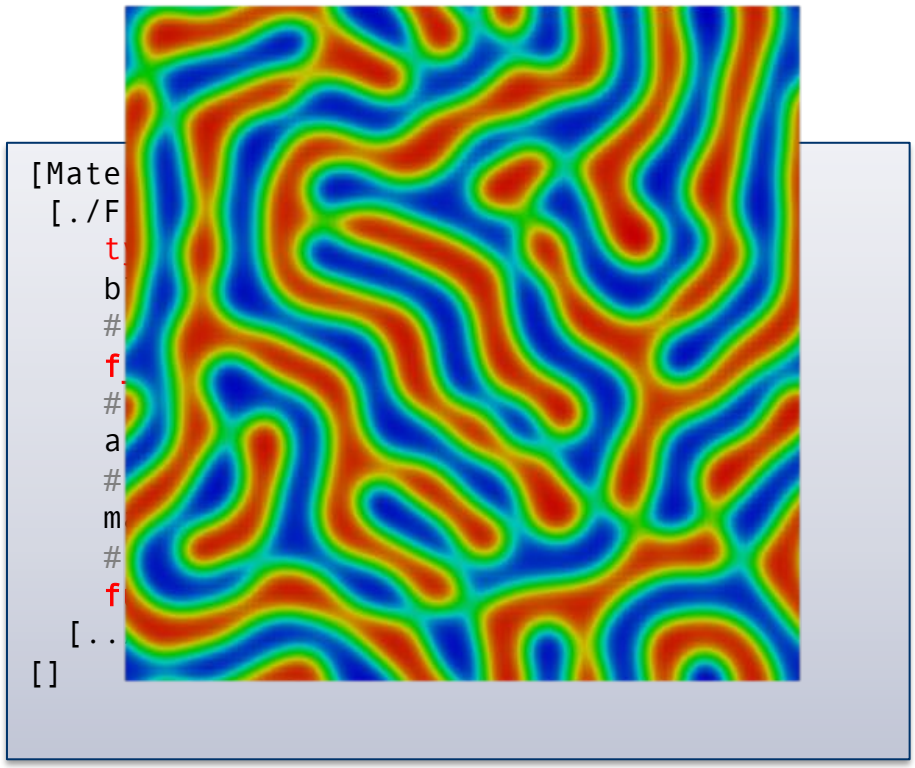
Derivative Function Materials

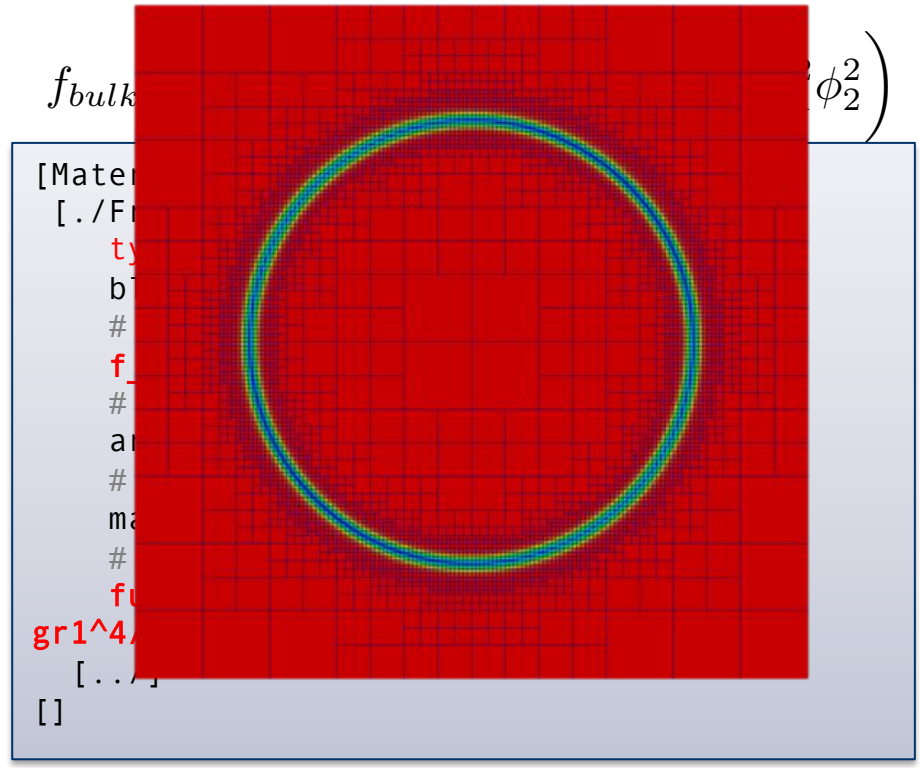
- The free energy and its derivatives can be defined in materials classes in four different ways:
  - The derivatives can be defined directly by the user, by inheriting from `DerivativeFunctionMaterialBase`
  - The derivatives can be calculated automatically, with the free energy defined in the input file using `DerivativeParsedMaterial`
  - The derivatives can be calculated automatically, with the free energy hard coded in a material object (`ExpressionBuilder`)
  - CALPHAD free energies (only for simple models now)

- A derivative material has an `f_name` (the function name)
- Property names of the derivatives are constructed automatically
  (using the value of `f_name` according to fixed rules set in the `DerivativeMaterialPropertyNameInterface` class)
- Add Derivative Function Materials using the `DerivativeSumMaterial`
  (sums function values and derivatives)

Automatic Free Energy Differentiation

- To simplify development even more, you can only enter the free energy functional and all derivatives are automatically evaluated analytically





+ Cahn-Hilliard

+ Allen-Cahn

Automatic Differentiation

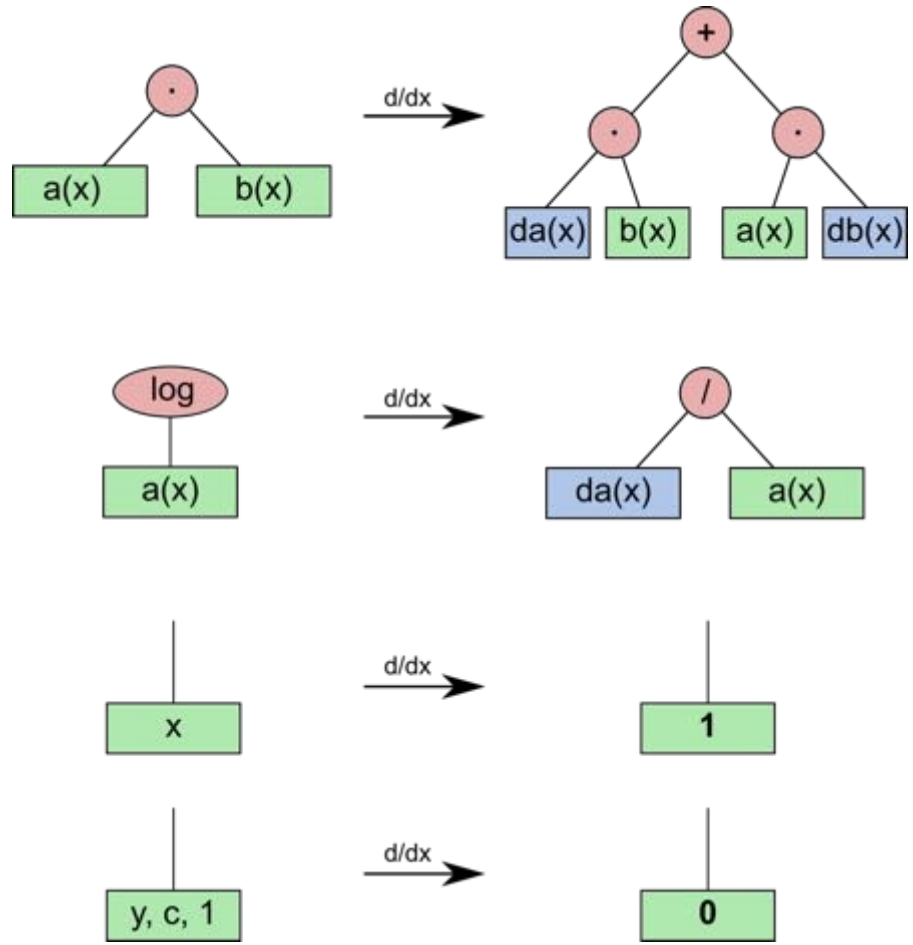
Symbolic differentiation of free energy expressions

- Based on FunctionParser
  http://warp.povusers.org/FunctionParser/
  to allow **runtime** specification of mathematical expressions

- Mathematical expressions
  ➔ Tree data structures

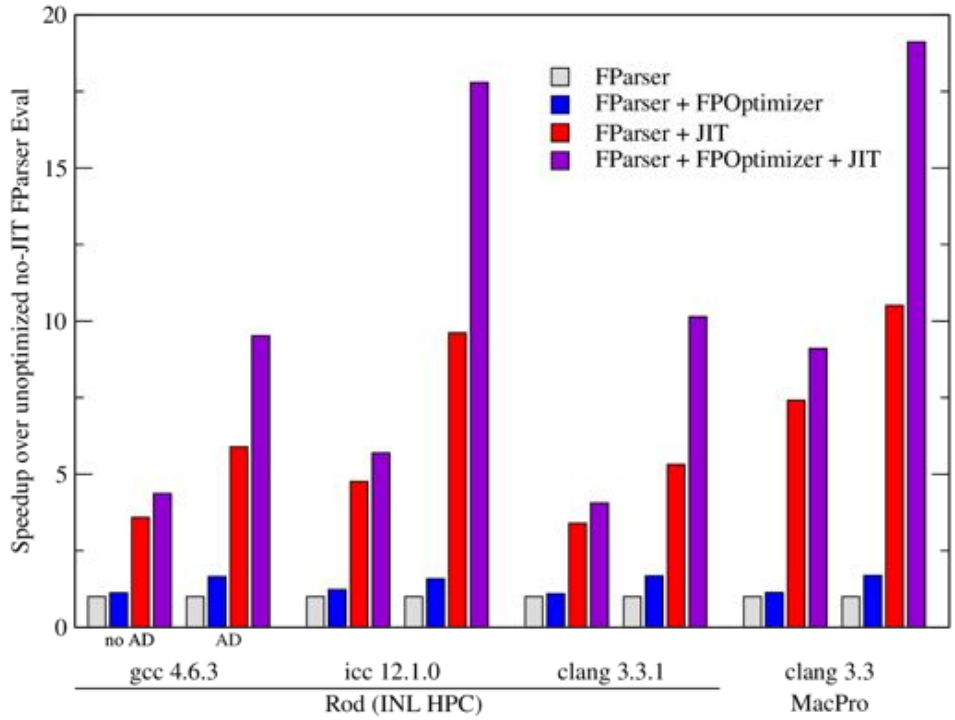
- Recursively apply differentiation rules starting at the root of the tree

- Eliminate source of human error
- Conserve developer time



Performance considerations

- Aren't interpreted functions slower than natively compiled functions?
- Just In Time (JIT) compilation for FParser functions
- Parsed functions (automatic differentiation) now as fast as hand coded functions
- Makes the rapid Phase Field model development more attractive
- ~80ms compile time per function. Results cached.



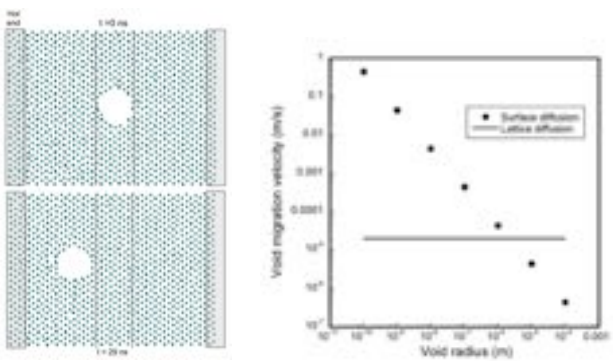
Examples and Applications

MARMOT Example: Void Migration

- Multiscale investigation of void migration in a temperature gradient (Soret effect):

Atomistic

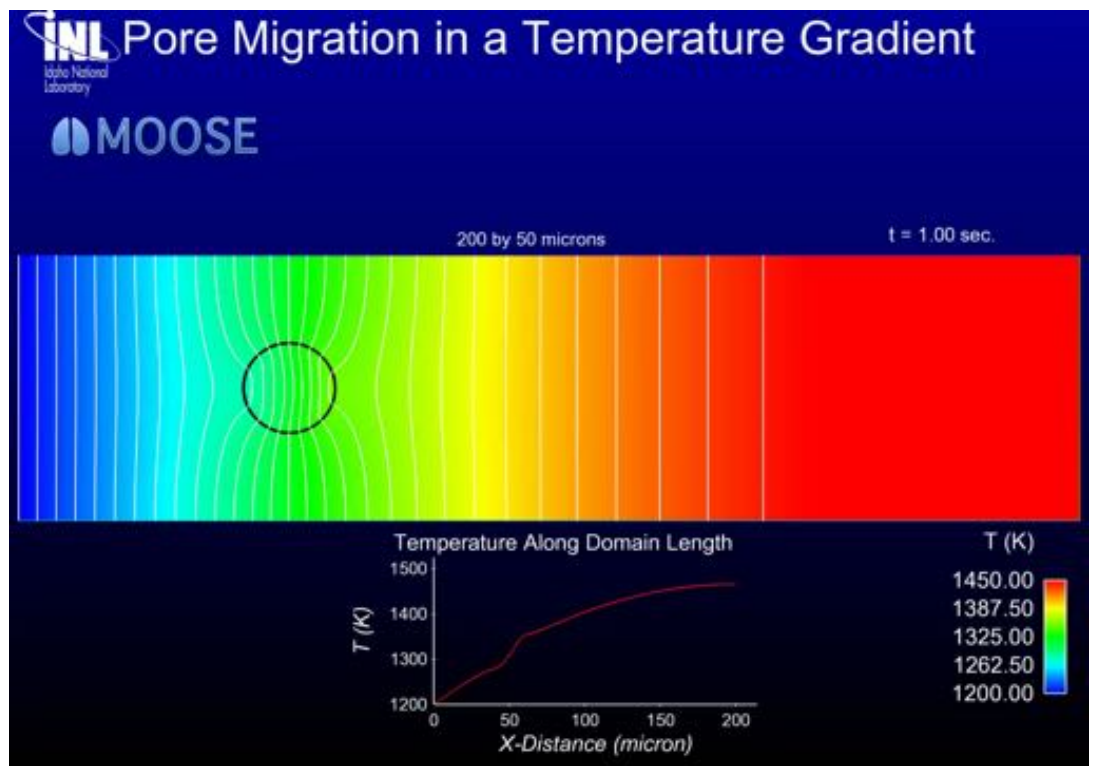
- MD studies identify the diffusion mechanisms active in the migration of nanovoids



From Desai (2009)

Mesoscale

- The migration of larger voids is modeled with MARMOT with surface and lattice diffusion



Zhang et al., Computational Materials Science, 56 (2012) 161-5

Particle and Pore Pinning

- Defects such as pores or precipitates on GBs impede the GB migration by applying an opposing force.
- To account for the interaction of GBs with a particle defined by the variable c, we add a term to the free energy

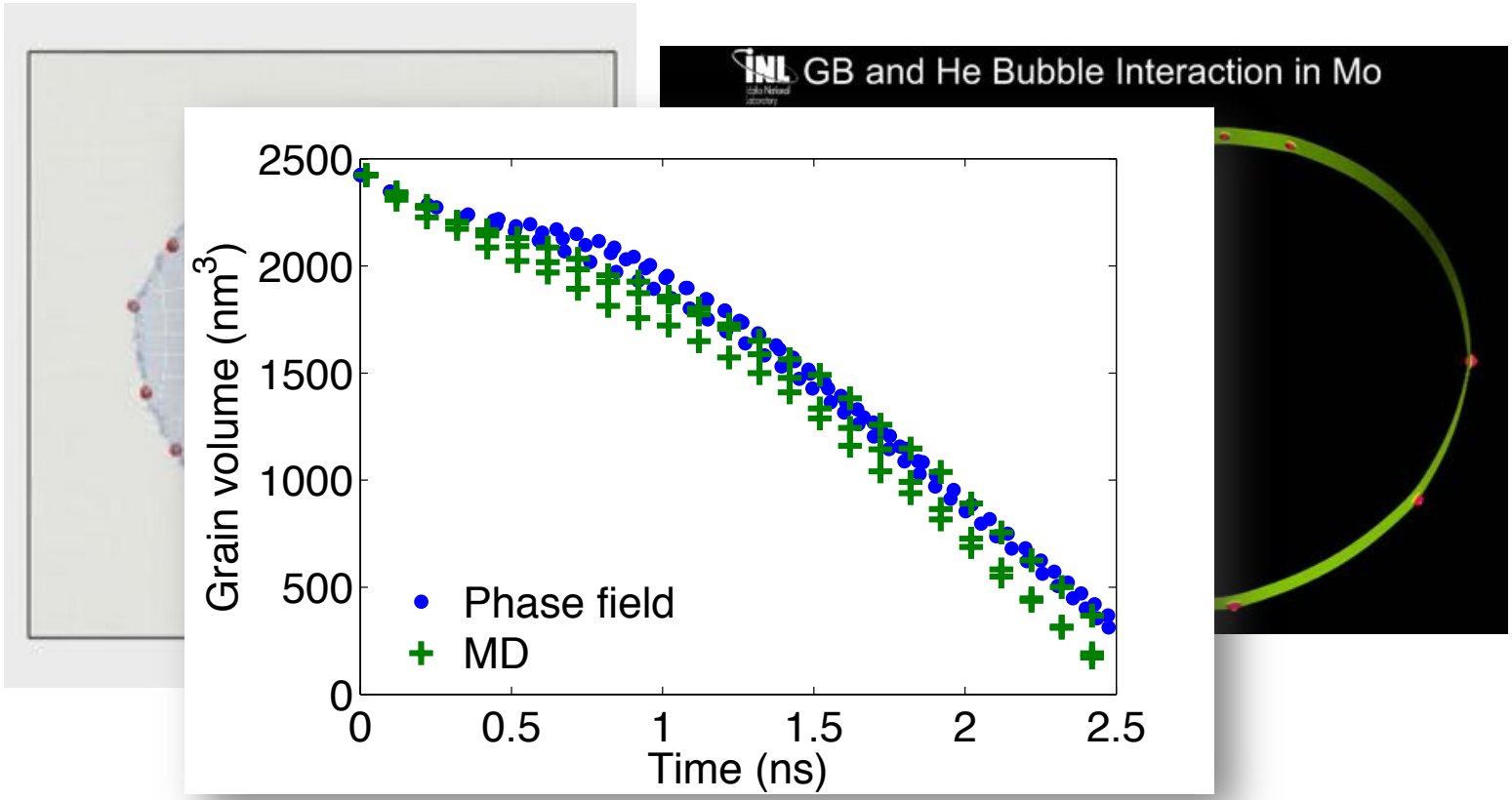
$$f(c,\eta_i) = \sum_i \left(\frac{\eta_i^4}{4} - \frac{\eta_i^2}{2}\right) + \left(\frac{c^4}{4} - \frac{c^2}{2}\right) + a_{GB}\sum_i \sum_{j>i} \eta_i^2 \eta_j^2 + a_s \sum_i c^2 \eta_i^2$$

- The term is implemented in the kernel `ACGBPoly`
- It is activated using the simplified grain growth syntax by adding a coupled variable `c`

```
[Kernels]
  [./PolycrystalKernel]
    c = c
  [../]
[]
```

Particle and Pore Pinning

- We verified this model by simulating an identical system using MD simulation and the phase field model
  - 10 He bubbles (r = 6 nm) in Mo bicrystal (R = 20 nm) at 2700 K.

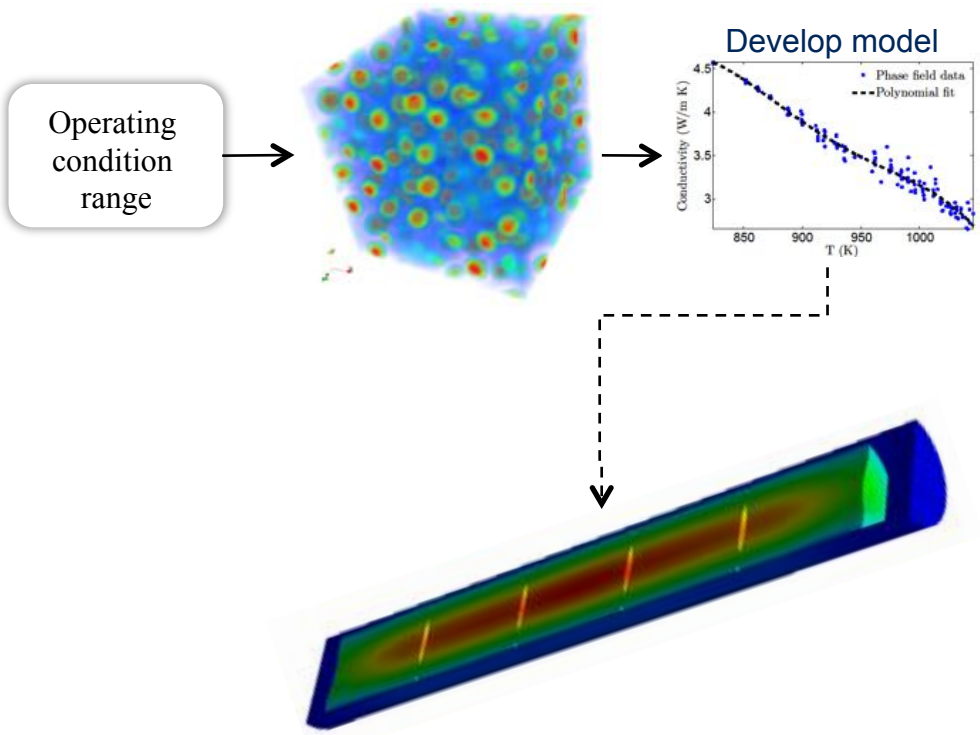


Coupling to Larger Length-Scales

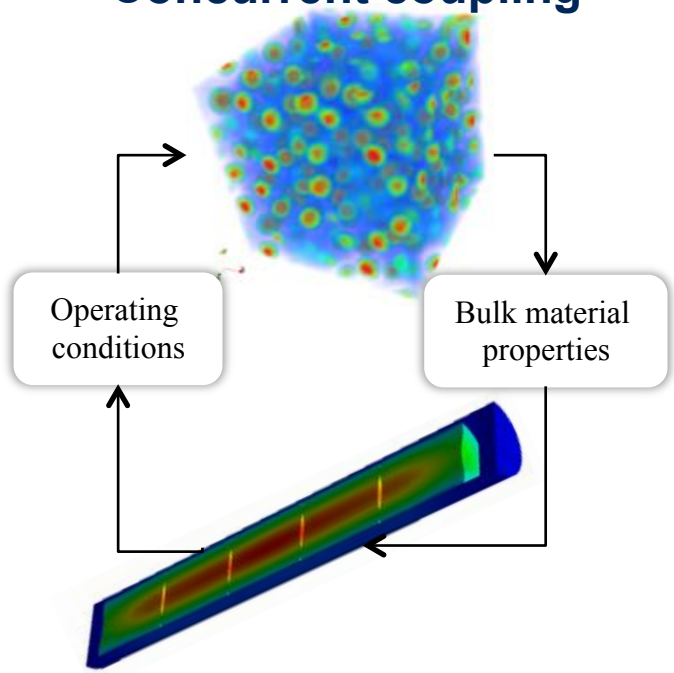
- MARMOT can be used in both hierarchical and concurrent coupling

Hierarchical coupling

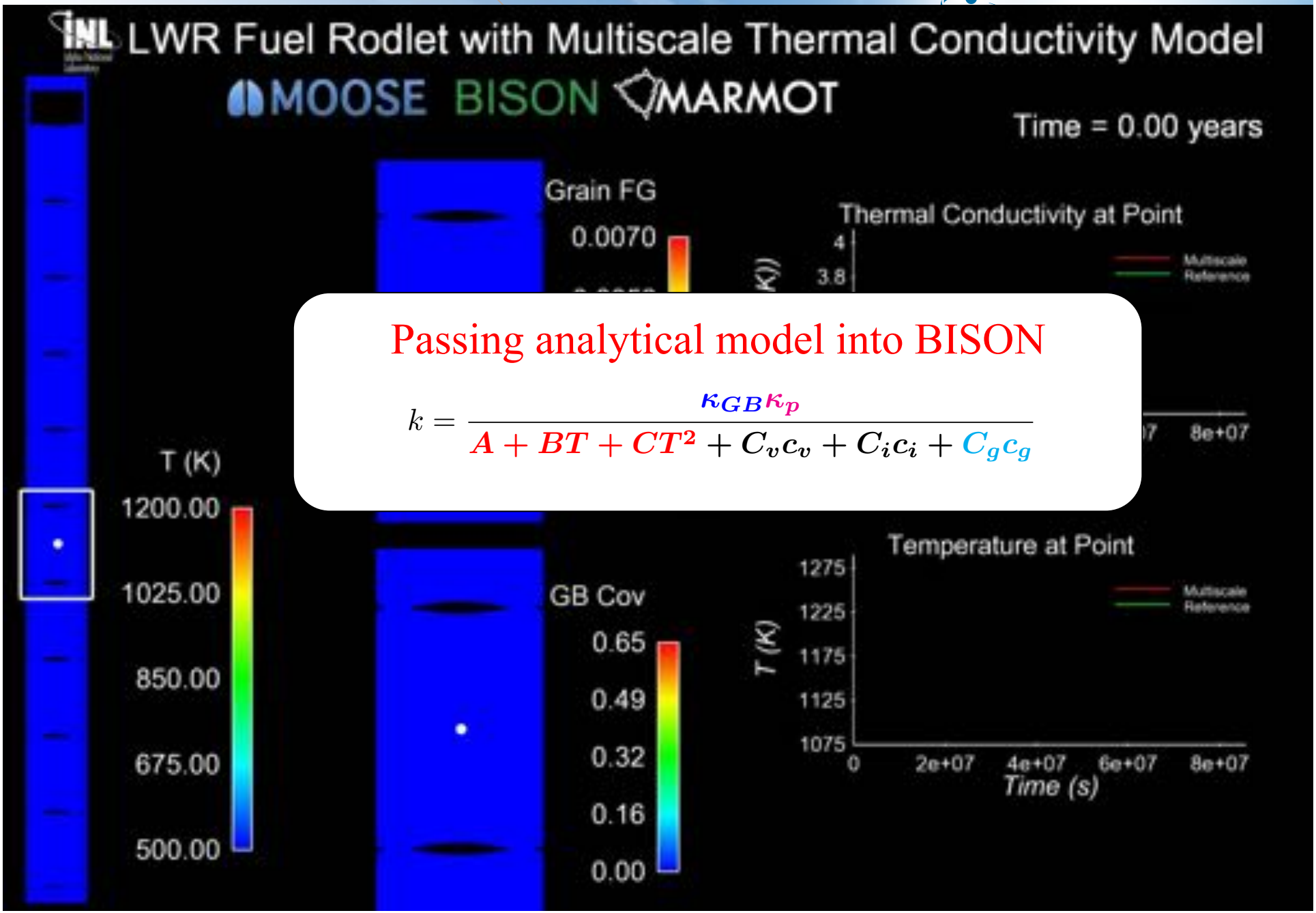
- Lower length-scale models are run separately to construct materials models.
- Macroscale simulations are efficient.



Concurrent coupling



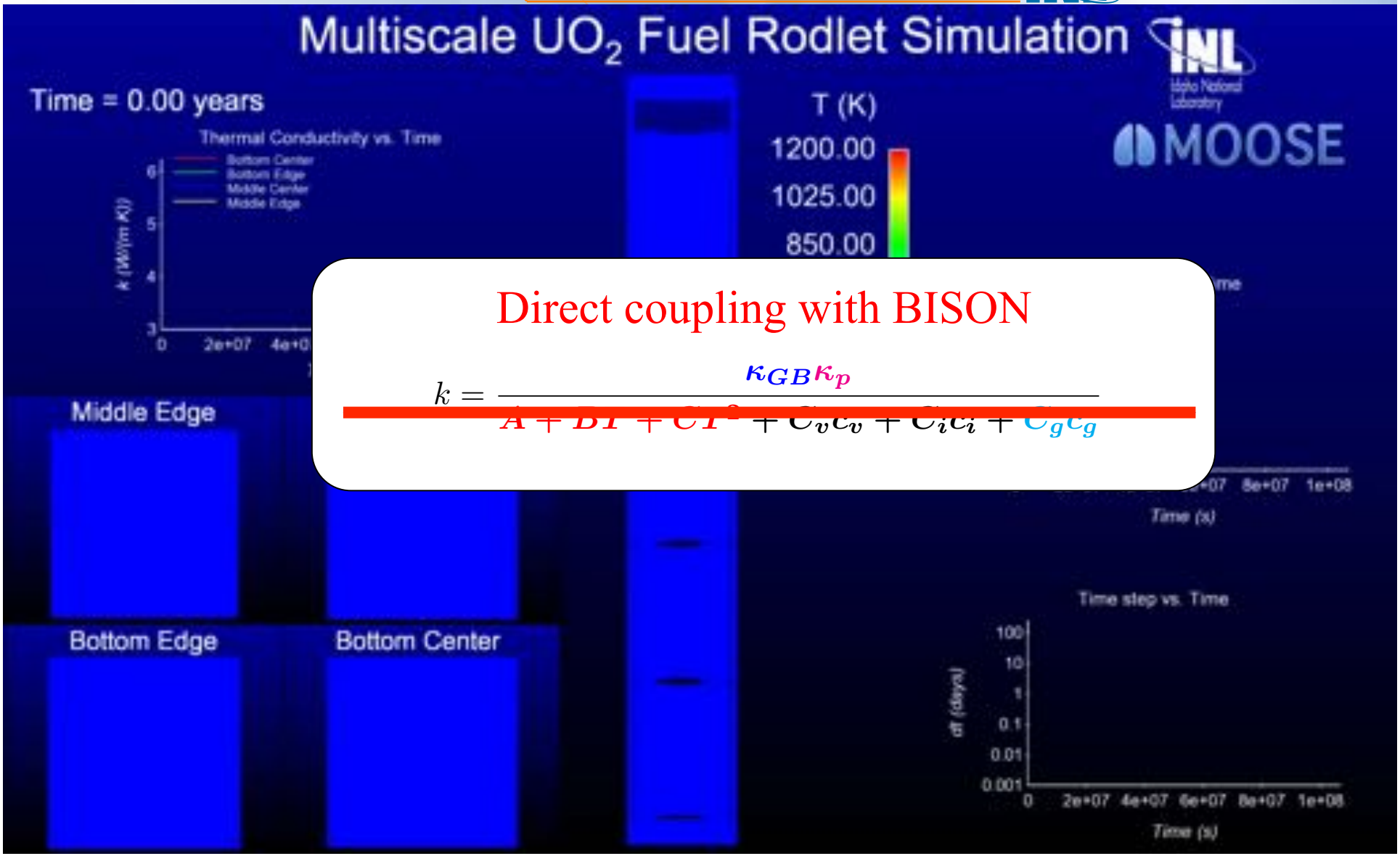
- Codes are run simultaneously and information is passed back and forth.
- Captures interaction between the scales
- Can locate important coupled behaviors
- More computationally expensive



Passing analytical model into BISON

$$k = \frac{\kappa_{GB}\kappa_p}{A + BT + CT^2 + C_v c_v + C_i c_i + C_g c_g}$$

Multiscale UO_2 Fuel Rodlet Simulation



Direct coupling with BISON

$$k = \frac{\kappa_{GB}\kappa_p}{A + BT + CT^2 + C_v c_v + C_i c_i + C_g c_g}$$

Thank you!

- For more information, please see http://mooseframework.org
- Github repository: https://github.com/idaholab/moose
- 3 day training workshops at INL and other locations (keep an eye on the website for dates and locations)
- Mailing list: to subscribe, send an email to

  moose-users+subscribe@googlegroups.com

  or see

  http://mooseframework.org/getting-started/