

MOOSE Workshop

How to get stuff done

www.inl.gov



Table of Contents /

Overview	4
Finite Elements: The MOOSE Way	14
The MOOSE Framework	33
The Input File	37
Mesh Types	50
The Outputs Block	60
InputParameters and MOOSE Types	66
Kernels	75
Coupling	85
Boundary Conditions	98
Mesh Adaptivity	110
Executioners	120
Initial Conditions	133
Materials	141
Peacock	154
Auxiliary System	159
Preconditioning	165
Functions	183
Postprocessors	188
Parallel-Agnostic Random Number Generation	201
MOOSEApp	205
Testing	210
MOOSE Modules	219

Table of Contents II

MOOSE Internal SVN Development	225
MOOSE External GitHub Development	239
Visualization Tools	244
Stork	248
Custom Actions (Advanced)	251
TimeSteppers (Advanced)	259
Dirac Kernels (Advanced)	263
Scalar Kernels (Advanced)	271
Geometric Search (Advanced)	277
Dampers (Advanced)	281
Discontinuous Galerkin (Advanced)	286
UserObjects (Advanced)	288
MultiApps (Advanced)	296
Transfers (Advanced)	303
Debugging (Advanced)	308

MOOSE Overview

www.inl.gov

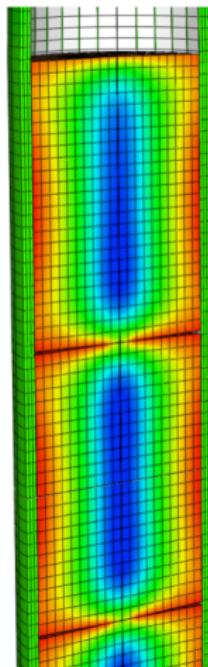


Core MOOSE Team

- Derek Gaston
 - derek.gaston@inl.gov
 - @friedmud
- Cody Permann
 - cody.permann@inl.gov
 - @permcdy
- David Andrs
 - david.andrs@inl.gov
 - @andrsdave
- John Peterson
 - jw.peterson@inl.gov
 - @peterson512
- Andrew Slaughter
 - andrew.slaughter@inl.gov
 - @aeslaughter98
- Jason Miller
 - jason.miller@inl.gov
 - @mjmiller96

MOOSE: Multiphysics Object Oriented Simulation Environment

- A framework for solving computational engineering problems in a well-planned, managed, and coordinated way
 - Leveraged across multiple programs
- Designed to significantly reduce the expense and time required to develop new applications
- Designed to develop analysis tools
 - Uses very robust solution methods
 - Designed to be easily extended and maintained
 - Efficient on both a few and many processors



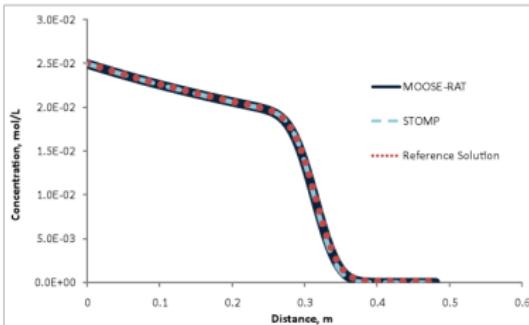
Capabilities

- 1D, 2D and 3D
 - User code agnostic of dimension
- Finite Element Based
 - Continuous and Discontinuous Galerkin (and Petrov Galerkin)
- Fully Coupled, Fully Implicit
- Unstructured Mesh
 - All shapes (Quads, Tris, Hexes, Tets, Pyramids, Wedges, ...)
 - Higher order geometry (curvilinear, etc.)
 - Reads and writes multiple formats
- Mesh Adaptivity
- Parallel
 - User code agnostic of parallelism
- High Order
 - User code agnostic of shape functions
 - p-Adaptivity
- Built-in Postprocessing

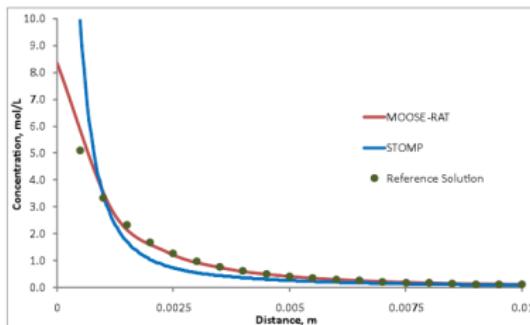
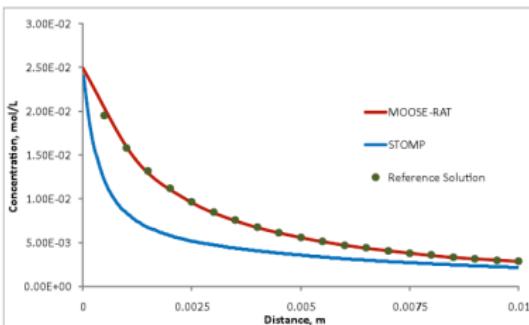
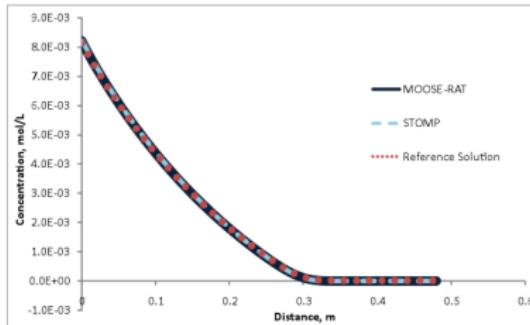
And much more ...

MOOSE solves problems that challenge others

Profile of A concentration at 4480 s



Profile of C concentration at 4480 s



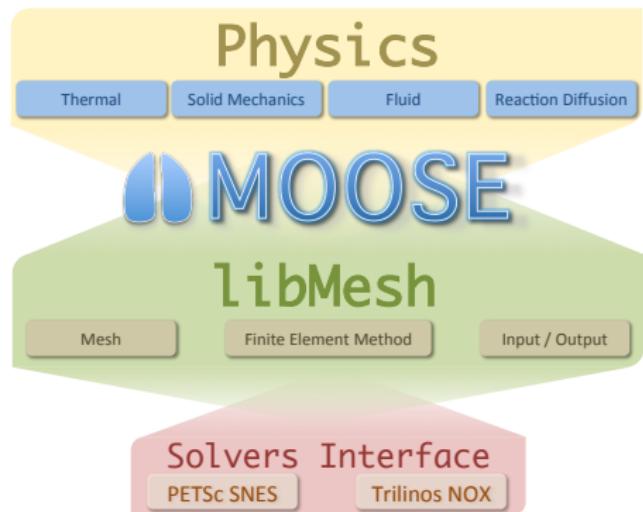
Weak coupling – excellent agreement between fully-coupled and operator-split approaches
 Strong coupling – better agreement between fully-coupled and the reference solution

MOOSE Ecosystem

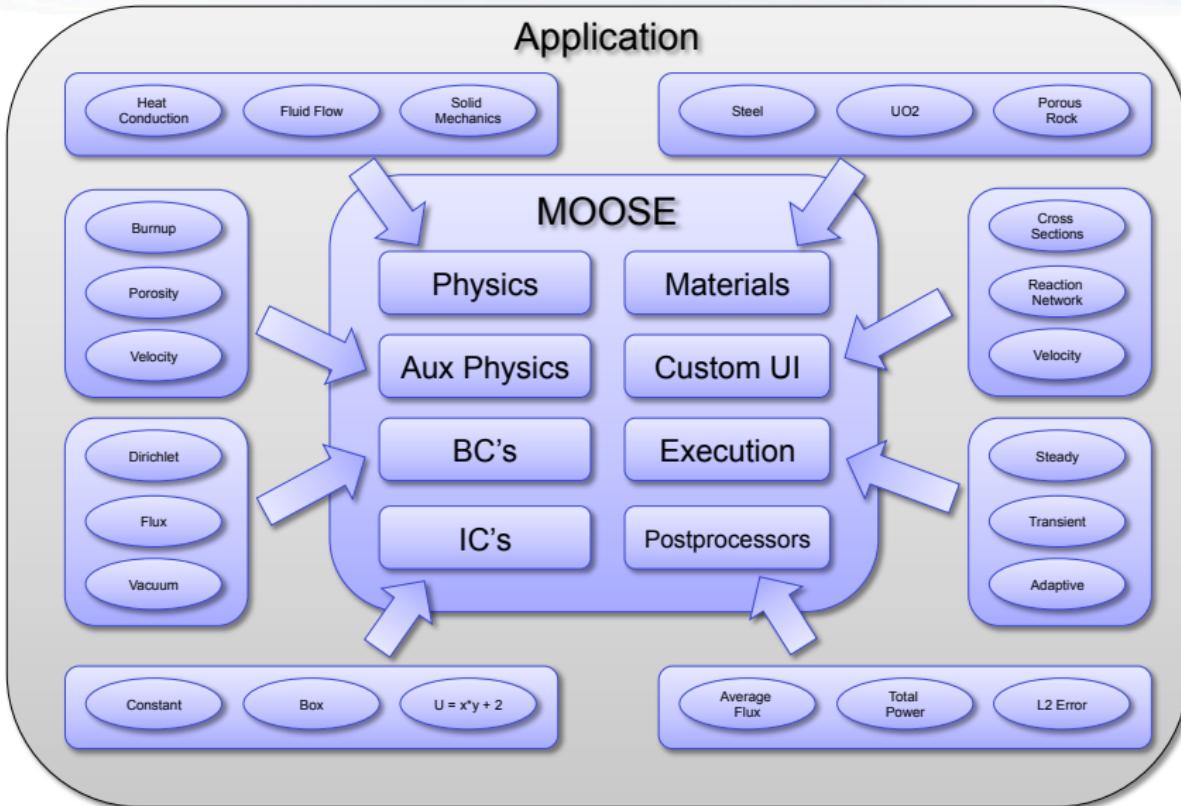
Application	Physics	Results	LoC
BISON	Thermo-mechanics, Chemical, diffusion, coupled mesoscale	4 months	3,000
PRONGHORN	Neutronics, Porous flow, eigenvalue	3 months	7,000
MARMOT	4 th order phasefield mesoscale	1 month	6,000
RAT	Porous ReActive Transport	1 month	1,500
FALCON	Geo-mechanics, coupled mesoscale	3 months	3,000
MAMBA	Chem. Rxn, Precipitation, Porous Flow	5 weeks	2,500
HYRAX	phase field, ZrH precipitation	3 months	1,000

Code Platform

- MOOSE is a simulation framework allowing rapid development of new simulations tools.
 - Specifically designed to simplify development of multiphysics tools.
- Provides an object-oriented, pluggable system for defining all aspects of a simulation tool.
 - Leverages multiple DOE developed scientific computational tools
- Allows scientists and engineers to develop state of the art simulation capabilities.
 - **Maximize Science/\$**
- Currently ~64,000 lines of code.



Current MOOSE Architecture



MOOSE Software Quality Practices

- MOOSE currently meets all NQA-1 (Nuclear Quality Assurance Level 1) requirements
- All commits to MOOSE undergo review using GitHub Pull Requests and must pass a set of application regression tests before they are available to our users.
- All changes must be accompanied by issue numbers and assessed an appropriate risk level.
- We maintain a regression test code coverage level of 80% or better at all times.
- We follow strict code style and formatting guidelines (available on our wiki).
- We monitor code comments with third party tools for better readability.

MOOSE produces results

1. M. Tonks, D. Gaston, P. Millett, D. Andrs, and P. Talbot. **An object-oriented finite element framework for multiphysics phase field simulations.** *Comp. Mat. Sci.*, 51(1):20–29, 2012.
2. R. Williamson, J. Hales, S. Novascone, M. Tonks, D. Gaston, C. Permann, D. Andrs, and R. Martineau. **Multidimensional Multiphysics Simulation of Nuclear Fuel Behavior.** *Submitted J. Nuclear Materials*, 2011.
3. M. Tonks, D. Gaston, C. Permann, P. Millett, G. Hansen, C. Newman, and D. Wolf. **A coupling methodology for mesoscale-informed nuclear fuel performance codes.** *Nucl. Engrg. Design*, 240:2877–2883, 2010.
4. D. Gaston, G. Hansen, S. Kadioglu, D. Knoll, C. Newman, H. Park, C. Permann, and W. Taitano. **Parallel multiphysics algorithms and software for computational nuclear engineering.** *Journal of Physics: Conference Series*, 180(1):012012, 2009.
5. M. Tonks, G. Hansen, D. Gaston, C. Permann, P. Millett, and D. Wolf. **Fully-coupled engineering and mesoscale simulations of thermal conductivity in UO₂ fuel using an implicit multiscale approach.** *Journal of Physics: Conference Series*, 180(1):012078, 2009.
6. C. Newman, G. Hansen, and D. Gaston. **Three dimensional coupled simulation of thermomechanics, heat, and oxygen diffusion in UO₂ nuclear fuel rods.** *Journal of Nuclear Materials*, 392:6–15, 2009.
7. D. Gaston, C. Newman, G. Hansen, and D. Lebrun-Grandjean. **MOOSE: A parallel computational framework for coupled systems of nonlinear equations.** *Nucl. Engrg. Design*, 239:1768–1778, 2009.
8. G. Hansen, C. Newman, D. Gaston, and C. Permann. **An implicit solution framework for reactor fuel performance simulation.** In *20th International Conference on Structural Mechanics in Reactor Technology (SMiRT 20)*, paper 2045, Espoo (Helsinki), Finland, August 9–14 2009.
9. G. Hansen, R. Martineau, C. Newman, and D. Gaston. **Framework for simulation of pellet cladding thermal interaction (PCTI) for fuel performance calculations.** In *American Nuclear Society 2009 International Conference on Advances in Mathematics, Computational Methods, and Reactor Physics, Saratoga Springs, NY, May 3–7 2009*.
10. C. Newman, D. Gaston, and G. Hansen. **Computational foundations for reactor fuel performance modeling.** In *American Nuclear Society 2009 International Conference on Advances in Mathematics, Computational Methods, and Reactor Physics, Saratoga Springs, NY, May 3–7 2009*.
11. D. Gaston, C. Newman, and G. Hansen. **MOOSE: a parallel computational framework for coupled systems of nonlinear equations.** In *American Nuclear Society 2009 International Conference on Advances in Mathematics, Computational Methods, and Reactor Physics, Saratoga Springs, NY, May 3–7 2009*.
12. L. Guo, H. Huang, D. Gaston, and G. Redden. **Modeling of calcite precipitation driven by bacteria-facilitated urea hydrolysis in a flow column using a fully coupled, fully implicit parallel reactive transport simulator.** In *Eos Transactions American Geophysical Union*, 90(52), Fall Meeting Supplement, AGU 90(52), San Francisco, CA, Dec 14–18 2009.
13. R. Podgornay, H. Huang, and D. Gaston. **Massively parallel fully coupled implicit modeling of coupled thermal-hydrological-mechanical processes for enhanced geothermal system reservoirs.** In *Proceedings, 35th Stanford Geothermal Workshop, Stanford University, Palo Alto, CA, Feb 1–3 2010*.
14. H. Park, D. Knoll, D. Gaston, and R. Martineau. **Tightly Coupled Multiphysics Algorithms for Pebble Bed Reactors.** *Nuclear Science and Engineering*, 166(2):118–133, 2010.

Finite Elements

The MOOSE way



www.inl.gov

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:

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

- Where a , b and c are scalar coefficients and 1 , x , x^2 are “basis functions”.
- The idea is to 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 interpolatory 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

Points

x	y
1	4
3	1
4	2

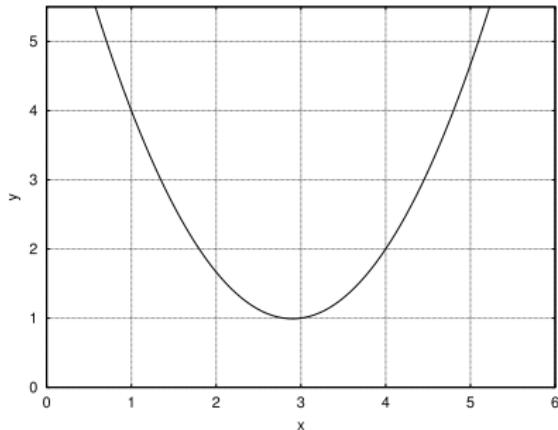
Linear System

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

Solution

$$\begin{bmatrix} 8 \\ -\frac{29}{6} \\ \frac{5}{6} \end{bmatrix}$$

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



Example Continued

- First, note that the coefficients themselves don't mean anything.
 - By themselves they are just numbers.
- Second, the solution is *not* the coefficients, but rather the *function* they create when they are multiplied by their respective basis functions and summed.
- Yes, $f(x)$ does go through the points we were given, *but it is also defined everywhere in between*.
- We can evaluate $f(x)$ at the point $x = 2$ (say) 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 MOOSE 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.
 6. Code up with MOOSE ... and run!

Refresher: The divergence theorem

- Transforms a volume integral into a surface integral:

$$\int_{\Omega} \nabla \cdot \mathbf{F} \, dx = \int_{\partial\Omega} \mathbf{F} \cdot \hat{\mathbf{n}} \, ds$$

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

$$-\int_{\Omega} \psi (\nabla \cdot k(x)\nabla u) \, dx = \int_{\Omega} \nabla \psi \cdot k(x)\nabla u \, dx - \int_{\partial\Omega} \psi (k(x)\nabla u \cdot \hat{\mathbf{n}}) \, ds$$

- In this talk, we will use the following inner product notation to represent integrals since it is more compact:

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

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

Example: Convection Diffusion

1. $-\nabla \cdot k \nabla u + \beta \cdot \nabla u = f$
2. $-\nabla \cdot k \nabla u + \beta \cdot \nabla u - f = 0$
3. $-\psi (\nabla \cdot k \nabla u) + \psi (\beta \cdot \nabla u) - \psi f = 0$
4. $-\int_{\Omega} \psi (\nabla \cdot k \nabla u) + \int_{\Omega} \psi (\beta \cdot \nabla u) - \int_{\Omega} \psi f = 0$
5. $\int_{\Omega} \nabla \psi \cdot k \nabla u - \int_{\partial \Omega} \psi (k \nabla u \cdot \hat{n}) + \int_{\Omega} \psi (\beta \cdot \nabla u) - \int_{\Omega} \psi f = 0$
6.
$$\underbrace{(\nabla \psi, k \nabla u)}_{\text{Kernel}} - \underbrace{\langle \psi, k \nabla u \cdot \hat{n} \rangle}_{BC} + \underbrace{(\psi, \beta \cdot \nabla u)}_{\text{Kernel}} - \underbrace{(\psi, f)}_{\text{Kernel}} = 0$$

More Depth

- While the weak form is essentially what you need for adding physics to MOOSE, in traditional finite element software more work is necessary.
- The next step is to discretize the weak form by selecting an expansion of u :

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

- The ϕ_j are generally called “shape functions”
- In the expansion of u_h , the ϕ_j are sometimes called “trial functions”
- Analogous to the x^n we used earlier
- The gradient of u can be expanded similarly:

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

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

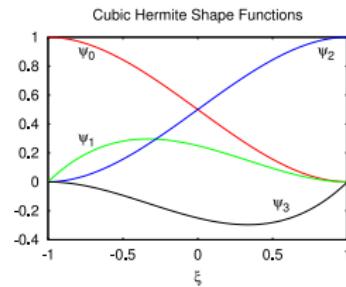
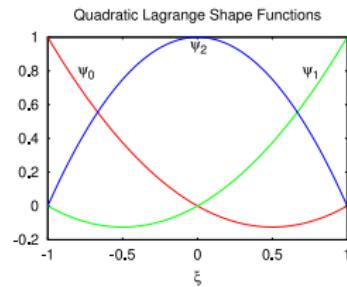
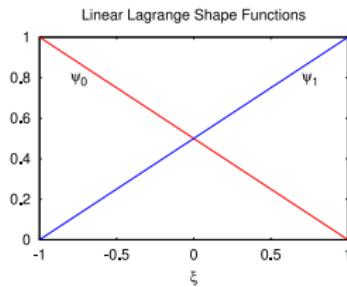
$$\psi = \{\phi_i\}_{i=1}^N$$

More Depth

- Substituting these expansions back into our weak form, we get:
$$(\nabla \psi_i, k \nabla u_h) - \langle \psi_i, k \nabla u_h \cdot \hat{\mathbf{n}} \rangle + (\psi_i, \boldsymbol{\beta} \cdot \nabla u_h) - (\psi_i, f) = 0, \quad i = 1, \dots, N$$
- The left-hand side of the equation above is what we generally refer to as the i^{th} component of our “Residual Vector” and write as $R_i(u_h)$.

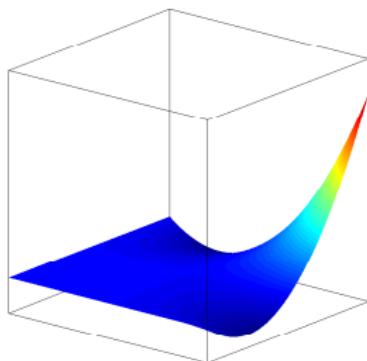
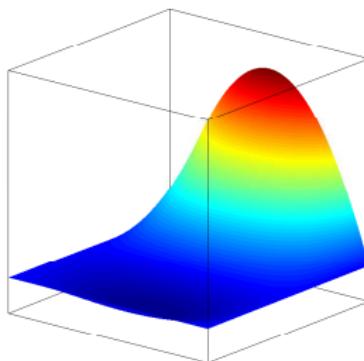
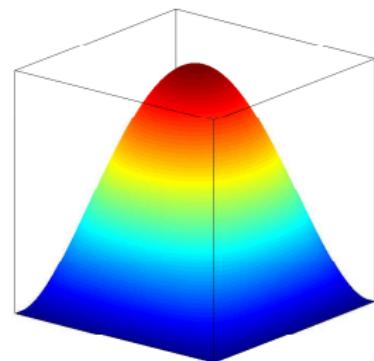
Shape Functions

- “Shape Functions” are the functions that get multiplied by coefficients and added up to form our solution.
- They are akin 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 are the most common.
 - They are interpolatory at the nodes.
 - This means the coefficients correspond to the values of the functions at the nodes.



2D Lagrange Shape Functions

- Some biquadratic basis functions defined on the Quad9 element:


 ψ_0

 ψ_4

 ψ_8

- 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.

Numerical Integration

- The only analytical piece left in the weak form are the integrals.
- To allow a computer to numerically compute these integrals we use quadrature (typically “Gaussian Quadrature”).
- Quadrature approximates continuous integrals by discrete sums:

$$\int f(\mathbf{x}) \approx \sum_{qp} f(\mathbf{x}_{qp}) w_{qp} \quad (1)$$

- Here \mathbf{x}_{qp} is the position of the quadrature point, and w_{qp} is the associated weight.
- Under certain common situations, the approximation (1) is exact!
 - Ex: In 1 dimension, Gaussian Quadrature can exactly integrate polynomials of order $2n - 1$ with n quadrature points.

Numerical Integration

- Note that sampling u for quadrature amounts to:

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

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

- And our weak form becomes:

$$\begin{aligned}
 R_i(u_h) &= \sum_{qp} w_{qp} \nabla \psi_i(\mathbf{x}_{qp}) \cdot k(\mathbf{x}_{qp}) \nabla u_h(\mathbf{x}_{qp}) \\
 &\quad - \sum_{qp_{face}} w_{qp_{face}} \psi_i(\mathbf{x}_{qp_{face}}) k(\mathbf{x}_{qp_{face}}) \nabla u_h(\mathbf{x}_{qp_{face}}) \cdot \mathbf{n}(\mathbf{x}_{qp_{face}}) \\
 &\quad + \sum_{qp} w_{qp} \psi_i(\mathbf{x}_{qp}) (\beta(\mathbf{x}_{qp}) \cdot \nabla u_h(\mathbf{x}_{qp})) \\
 &\quad - \sum_{qp} w_{qp} \psi_i(\mathbf{x}_{qp}) f(\mathbf{x}_{qp})
 \end{aligned}$$

Newton's Method

- We now have a nonlinear system of equations,

$$R_i(u_h) = 0, \quad i = 1, \dots, N$$

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

- 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 $f(x) : \mathbb{R} \rightarrow \mathbb{R}$

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

$$x_{n+1} = x_n + \delta x_{n+1}$$

Newton's Method Continued

- 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:

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

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

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

- Note that:

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

Newton For A Simple Equation

- Consider the convection-diffusion equation with nonlinear k , β and f :

$$-\nabla \cdot k \nabla u + \beta \cdot \nabla u = f$$

- The i^{th} component of the residual vector is:

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

- 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:

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

- Note that even for this “simple” equation the Jacobian entries are nontrivial, especially when the partial derivatives of k , β and f are actually expanded.
- In a multiphysics setting with many coupled equations and complicated material properties, the Jacobian might be extremely difficult to determine.

Jacobian Free Newton Krylov

- $\mathbf{J}(\mathbf{u}_n)\delta\mathbf{u}_{n+1} = -\mathbf{R}(\mathbf{u}_n)$ is a linear system of equations to solve during each Newton step.
- Lots of choices but we typically employ GMRES (a Krylov subspace method).
 - Scales well in parallel.
 - Effective on a diverse set of problems.
- In a Krylov iterative method (such as GMRES) we have the representation:

$$\delta\mathbf{u}_{n+1,k} = a_0\mathbf{r}_0 + a_1\mathbf{J}\mathbf{r}_0 + a_2\mathbf{J}^2\mathbf{r}_0 + \cdots + a_k\mathbf{J}^k\mathbf{r}_0$$

- Note that \mathbf{J} is never explicitly needed. Instead, only the action of \mathbf{J} on a vector needs to be computed.
- This action can be approximated by:

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

- 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 to PDEs.
- Just like polynomial fitting, FEM finds coefficients for basis functions.
- The “solution” is the combination of the coefficients and the basis functions, and just like polynomial fitting the solution can be sampled anywhere in the domain.
- We can numerically find the values of integrals 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 circumvent tedious (and error prone) derivative calculations.

The MOOSE Framework



www.inl.gov

MOOSE Requirements

- MOOSE requires several pieces of software to be in place before it will build:
 - MPI
 - Hypre (Optionally if you want to use AMG)
 - PETSc
 - libMesh
- If you are working on your own workstation you will have to compile / install all of these.
- We maintain binary builds of these packages for the following operating systems:
 - Ubuntu (deb), OpenSUSE (rpm), tarball installer coming soon
 - OS X 10.7, 10.8, and 10.9
- <https://www.mooseframework.org>
 - Getting Started Instructions
 - Issue System
 - Documentation, Blog, and Wiki

Anatomy

- Any MOOSE-based application should have the following directory structure:

```
application/
    LICENSE
    Makefile
    run_tests
    doc/
    lib/
    src/
        main.C
        base/
        actions/
        auxkernels/
        bcs/
        dampers/
        dirackernels/
        executioners/
        functions/
        ics/
        kernels/
        materials/
        postprocessors/
        utils/
    tests/
        ... (same stuff as src)
```

Looking at MOOSE

- All classes are separated out into directories associated with that set of classes:
 - kernels/
 - bcs/
 - ics/
 - auxkernels/
 - dgkernels/
 - functions/
 - actions/
 - etc...

The Input File



www.inl.gov

The Input File

- By default MOOSE uses a hierarchical, block-structured input file.
- Within each block, any number of name/value pairs can be listed.
- The syntax is completely customizable, or replaceable.
- To specify a simple problem, you will need to populate five or six top-level blocks.
- We will briefly cover a few of these blocks in this section and will illustrate the usage of the remaining blocks throughout this manual.

Main Input File Blocks

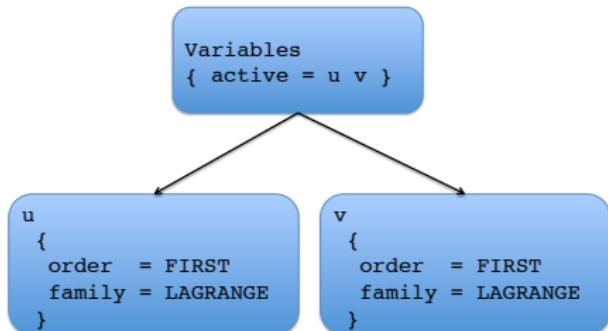
- MOOSE expects the following basic blocks for a simple problem:
 - [Mesh]
 - [Variables]
 - [Kernels]
 - [BCs]
 - [Executioner]
 - [Outputs]

Hierarchical Block-Structure

```
[Variables]
active = 'u v'
[./u]
order = FIRST
family = LAGRANGE
[../]

[./v]
order = FIRST
family = LAGRANGE
[../]
[]

[Kernels]
...
[]
```



The Mesh Block

```
[Mesh]
# Optional Type
type = <FileMesh | GeneratedMesh>

# FileMesh
file = <filename>

# Some other commonly used options
uniform_refine = <n>
second_order = <true|false>
[]
```

- The Mesh block is generally associated with an Action that reads and/or constructs the mesh for the simulation.
- The default type, FileMesh, is suitable for reading any normal mesh format from a file.
- The base class MooseMesh can be extended to construct or modify your own Meshes during runtime as needed.
- There are additional advanced options for this and the following blocks which are not listed in this section...

The Variables Block

```
[Variables]
[./nonlinear_variable1]
  order = <FIRST | SECOND | ...>
  family = <LAGRANGE | HERMITE | ...>
[../]

[./nonlinear_variable2]
  ...
[../]
[]
```

- The **Variables** block declares the nonlinear variables that will be solved for in the simulation.
- The default **order** and **family** are **FIRST** and **LAGRANGE**, respectively.

The Kernels Block

```
[Kernels]
[./my_kernel1]
  type = <Any Registered Kernel>
  variable = <Nonlinear Variable Name>
[../]

[./my_kernel2]
...
[../]
[]
```

- The **Kernels** block declares PDE operators that will be used in the simulation.
- The **type** parameter is used to specify the type of kernel to instantiate.

The Boundary Conditions Block

```
[BCs]
[./bottom_bc]
  type = <Any Registered BC>
  variable = <Nonlinear Variable Name>
[...]
[./top_bc]
...
[...]
[]
```

- The `BCs` block declares the boundary conditions that will be used in the simulation.
- The `type` parameter is used to specify the type of boundary condition to instantiate.

The Executioner Block

```
[Executioner]
  type = <Steady | Transient | ...>
[]
```

- The Executioner block declares the executioner that will be used in the simulation.
- The `type` parameter is used to specify the type of executioner to instantiate.

The Outputs Block

```
[Outputs]
  file_base = <base_file_name>
  exodus = true
  console = true
[]
```

- The Outputs block controls the various output (to screen and file) used in the simulation.

Example Input File

```
[Mesh]
  file = mug.e
[]

[Variables]
  active = 'diffused'

  [./diffused]
    order = FIRST
    family = LAGRANGE
  [...]
[]

[Kernels]
  active = 'diff'

  [./diff]
    type = Diffusion
    variable = diffused
  [...]
[]

[BCs]
  active = 'bottom top'

  [./bottom]
    type = DirichletBC
    variable = diffused
    boundary = 'bottom'
    value = 1
  [...]

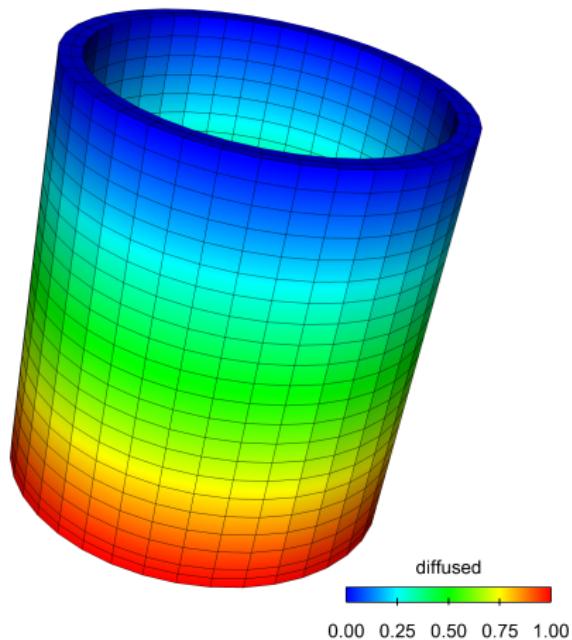
  [./top]
    type = DirichletBC
    variable = diffused
    boundary = 'top'
    value = 0
  [...]
[]
```

```
[Executioner]
  type = Steady
[]

[Outputs]
  file_base = out
  exodus = true
  console = true
[]
```

Look at Example 1 (page E3)

Example 1 Output



The Mesh Block and Types

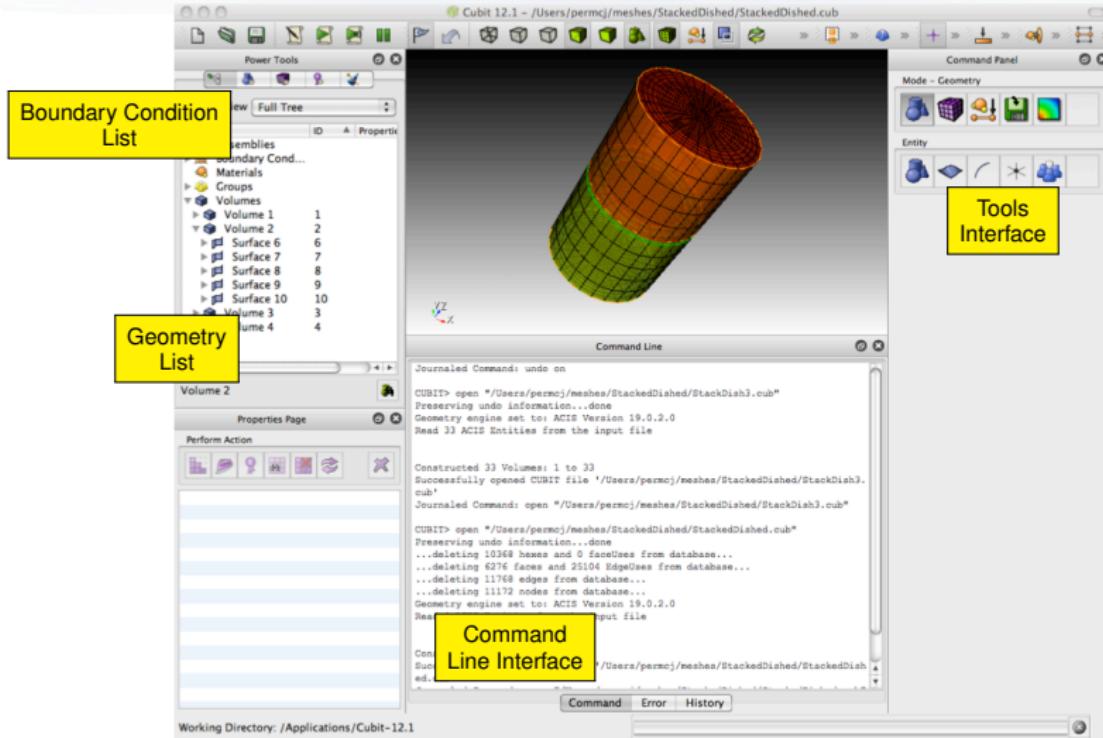


www.inl.gov

Creating a Mesh

- For complicated geometries, we generally use CUBIT from Sandia National Laboratories.
- CUBIT can be licensed from CSimSoft for a fee depending on the type of organization you work for.
- Other mesh generators can work as long as they output a file format that libMesh reads (next slide).
- If you have a specific mesh format that you like, we can take a look at adding support for it to libMesh.

CUBIT Interface



FileMesh

- FileMesh is the default type when the “type” parameter is omitted from the Mesh section.
- MOOSE supports reading and writing a large number of formats and could be extended to read more.

.dat	Tecplot ASCII file
.e, .exd	Sandia's ExodusII format
.fro	ACDL's surface triangulation file
.gmv	LANL's GMV (General Mesh Viewer) format
.mat	Matlab triangular ASCII file (read only)
.msh	GMSH ASCII file (write only)
.n, .nem	Sandia's Nemesis format
.plt	Tecplot binary file (write only)
.poly	TetGen ASCII file (write only)
.inp	Abaqus .inp format (read only)
.ucd	AVS's ASCII UCD format
.unv	I-deas Universal format
.xda, .xdr	libMesh formats
.vtk	Visualization Toolkit

Generated Mesh

- type = GeneratedMesh
- Built-in mesh generation is implemented for lines, rectangles, and rectangular prisms ("boxes") but could be extended.
- The sides of a GeneratedMesh are named in a logical way (bottom, top, left, right, front, and back).
- The length, width, and height of the domain, and the number of elements in each direction can be specified independently.

Named Entity Support

```
...
[Mesh]
file = three_block.e

# These names will be applied on the
# fly to the mesh so that they can be
# used in the input file. In addition
# they will be written to the output
# file
block_id = '1 2 3'
block_name = 'wood steel copper'

boundary_id = '1 2'
boundary_name = 'left right'
[]

...
```

- Human-readable names can be assigned to blocks, sidesets and nodesets. These names will be automatically read in and can be used throughout the input file.
- This is typically done inside of Cubit.
- Any parameter that takes entity IDs in the input file will accept either numbers or “names”.
- Names can also be assigned to IDs on-the-fly in existing meshes to ease input file maintenance (see example).
- On-the-fly names will also be written to Exodus/XDA/XDR files.

Parallel Mesh

- Useful when the mesh data structure dominates memory usage.
- Only the pieces of the mesh “owned” by processor N are actually stored by processor N .
- If the mesh is too large to even read in on a single processor, it can be split prior to the simulation.
 - Copy the mesh to a large memory machine (Eos at the INL).
 - Use a tool to split the mesh into n pieces (SEACAS, loadbal).
 - Copy the pieces to your working directory on the cluster.
 - Use the Nemesis reader to read the mesh using n processors.
- See `nemesis_test.i` in the `moose_test` directory for a working example.

Displaced Mesh

- Calculations can take place in either the initial mesh configuration or the “displaced” configuration when requested.
- To enable displacements, simply supply a vector of variables representing displacements for each spatial dimension in the `Mesh` section:
`displacements = 'disp_x disp_y disp_z'`
- Once enabled, the parameter `use_displaced_mesh` can be set on individual MooseObjects which will enable them to use displaced coordinates during calculations:

```
template<>
InputParameters validParams<SomeKernel>()
{
    InputParameters params = validParams<Kernel>();
    params.set<bool>("use_displaced_mesh") = true;
    return params;
}
```

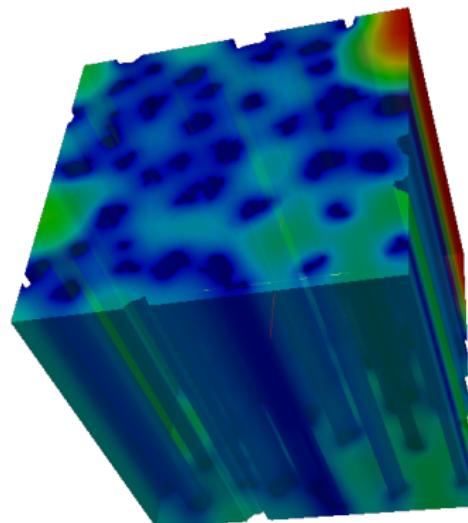
- This can also be set in the input file, but it's a good idea to do it in code if you have a pure Lagrangian formulation.

Mesh Modifiers

- [MeshModifiers] perform additional manipulations to a mesh after it has been set up.
- This can range from adding an additional node set to performing a complex set of transformations.
- MOOSE has a pluggable system for adding your own modifiers
- Inherit from MeshModifier and implement `modify()`
- A few built-ins:
 - AddExtraNodeset
 - SideSetFromNormals, SideSetFromPoints
 - Transform - **Scale, Rotate, Translate**
 - MeshExtruder

Extrusion

- type = MeshExtruder
- Takes a 1D or 2D mesh and extrudes it to 2D or 3D respectively.
- Triangles are extruded to prisms (wedges).
- Quadrilaterals are extruded to hexahedra.
- Sidesets are extruded and maintained.
- Top and bottom sidesets can be specified.
- The extrusion vector should be specified (direction and length).



Extruded Mesh result from MAMBA courtesy of Michael Short.

The Outputs Block



www.inl.gov

The Outputs Block

- The output system is like any other system in MOOSE: modular and expandable.
- It is possible to create multiple output objects each outputting at different intervals, different variables, or varying file types.
- Supports ‘short-cut’ syntax, common parameters, and sub-blocks

```
[Outputs]
  file_base = <base_file_name>
  exodus = true
  console = true
[]
```

```
[Outputs]
  output_initial = true
  vtk = true
  [./console]
    type = Console
    perf_log = true
  [../]
  [./exodus]
    type = Exodus
    output_initial = false
  [../]
  [./exodus_displaced]
    type = Exodus
    file_base = displaced
    use_displaced = true
    interval = 3
  [../]
[]
```

Supported Formats

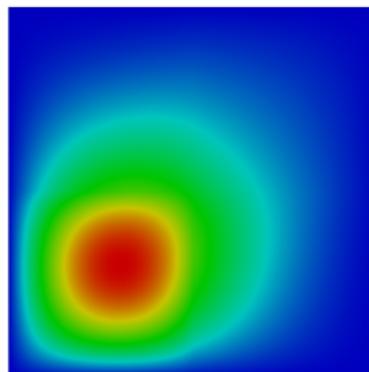
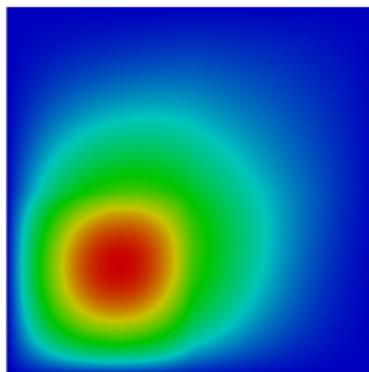
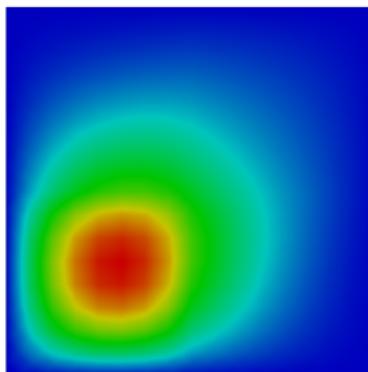
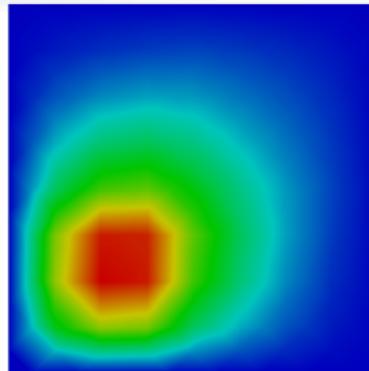
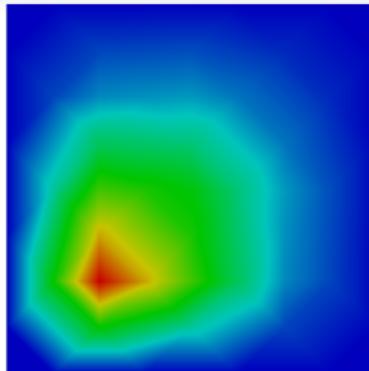
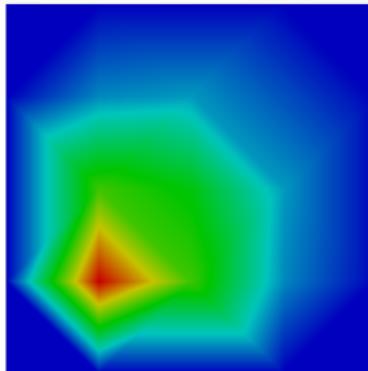
Format	Short-cut Syntax	Sub-block Type
Console (screen)	console = true	type = Console
Exodus II	exodus = true	type = Exodus
VTK	vtk = true	type = VTK
GMV	gmv = true	type = GMV
Nemesis	nemesis = true	type = Nemesis
Tecplot	tecplot = true	type = Tecplot
XDA	xda = true	type = XDA
XDR	xdr = true	type = XDR
CSV	csv = true	type = CSV
GNUpolt	gnuplot = true	type = GNUPlot
Checkpoint	checkpoint = true	type = Checkpoint

Over Sampling

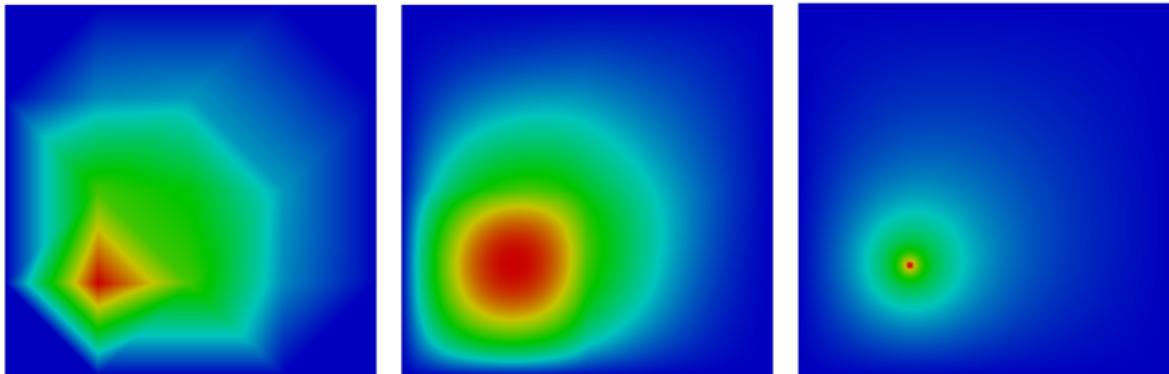
- None of the generally available visualization packages currently display higher-order solutions (quadratic, cubic, etc.) natively.
- To work around this limitation, MOOSE can “oversample” a higher-order solution by projecting it onto a finer mesh of linear elements.
- **Note:** This is not mesh adaptivity, nor does it improve the solution’s accuracy in general.
- The following slide shows a sequence of solutions oversampled from a base solution on second-order Lagrange elements.

```
[Outputs]
  console = true
  ['./exodus']
    file_base = out_oversample
    type = Exodus
    oversample = true
    refinements = 3
  [...]
[]
```

Over Sampling Refinements: 0 to 5 levels



Over Sampling (Cont.)



- It's important to note that oversampling will not *improve* the solution!
- The solution on the left is solved on a "coarse" grid.
- The solution in the center is the *same* as on the left, but has been oversampled for visualization purposes.
- The solution on the right is for the same problem, but solved on a finer mesh (and is therefore closer to the true solution).

Input Parameters and MOOSE Types



www.inl.gov

Valid Parameters

- A set of custom parameters (`InputParameters`) is used to construct every MOOSE object.
- Every MOOSE-derived object must specify a `validParams` function.
- In this function you *must* start with the parameters from your parent class (like `Kernel`) and then specify additional parameters.
- This function must return a set of parameters that the corresponding object requires in order to be constructed.
- This design allows users to control any and all parameters they need for constructing objects while leaving all C++ constructors uniform and unchanged.

Defining Valid Parameters

In the .h file:

```
class Convection;

template<>
InputParameters validParams<Convection>();

class Convection : public Kernel
...
```

In the .C file:

```
template<>
InputParameters validParams<Convection>()
{
    InputParameters params = validParams<Kernel>(); // Must get from parent
    params.addRequiredParam<RealVectorValue>("velocity", "Velocity Vector");
    params.addParam<Real>("coefficient", "Diffusion coefficient");
    return params;
}
```

On the Fly Documentation

- The parser object is capable of generating documentation from the `validParams` functions for each class that specializes that function.
- Option 1: Generate a complete tree of registered objects
 - CLI option `--dump [optional search string]`
 - The search string may contain wildcard characters
 - Searches both block names and parameters
 - All parameters are printed for a matching block
- Option 2: Generate a tree based on your input file
 - CLI option `--show-input`
- Option 3: View it online
 - <http://mooseframework.org/wiki/InputFileSyntax>

The InputParameters Object

- The parser and Peacock work with the InputParameters object to read almost any kind of parameter.
- Built-in types and `std::vector` are supported via templated methods:
 - `addRequiredParam<Real>("required_const", "doc");`
 - `addParam<int>("count", 1, "doc"); // default supplied`
 - `addParam<unsigned int>("another_num", "doc");`
 - `addRequiredParam<std::vector<int>>("vec", "doc");`
- Other supported parameter types include:
 - Point
 - RealVectorValue
 - RealTensorValue
 - SubdomainID
 - BoundaryID
- For the complete list of supported types see
`Parser::extractParams(...)`

The *InputParameters* Object (cont.)

- MOOSE uses a large number of string types to make Peacock more context-aware. All of these types can be treated just like strings, but will cause compile errors if mixed improperly in the template functions.

SubdomainName

BoundaryName

FileName

MeshFileName

OutFileName

VariableName

NonlinearVariableName

AuxVariableName

FunctionName

UserObjectName

PostprocessorName

IndicatorName

MarkerName

- For a complete list, see the instantiations at the bottom of MooseTypes.h

Default and Range-checked Parameters

- You may supply a default value for all optional parameters (not required)

```
addParam<RealVectorValue>("direction", RealVectorValue(0,1,0), "doc");
```

- The following types allow you to supply scalar defaults in place of object:
 - Any coupled variable
 - Postprocessors (PostprocessorName)
 - Functions (FunctionName)
- You may supply an expression to perform range checking within the parameter system.
- You should use the name of your parameter in the expression.

```
addRangeCheckedParam<Real>("temp", "temp>=300 & temp<=330", "doc");
```

- Function Parser Syntax

<http://warp.povusers.org/FunctionParser/fparser.html>

MooseEnum

- MOOSE includes a “smart” enum utility to overcome many of the deficiencies in the standard C++ enum type.
- It works in both integer and string contexts and is self-checked for consistency.

```
#include "MooseEnum.h"
...
// The valid options are specified in a comma separated list.
// You can optionally supply the default value as a second argument.
// MooseEnums are case preserving but case-insensitive.
MooseEnum option_enum("first=1, second, fourth=4", "second");

// Use in a string context
if (option_enum == "first")
    doSomething();

// Use in an integer context
switch(option_enum)
{
    case 1: ... break;
    case 2: ... break;
    case 4: ... break;
    default: ... ;
}
```

Using MooseEnum with InputParameters

- Objects that have a specific set of named options should use a MooseEnum so that parsing and error checking code can be omitted.

```
template<>
InputParameters validParams<MyObject>()
{
    InputParameters params = validParams<ParentObject>();

    MooseEnum component("X, Y, Z"); // No default supplied
    params.addRequiredParam<MooseEnum>("component", component,
                                         "The X, Y, or Z component");
    return params;
}

...
// Assume we've saved off our MooseEnum into a instance variable: _component
Real value = 0.0;
if (_component.isValid())
    value = _some_vector[_component];
```

- Peacock will automatically create a drop box when using MooseEnum.
- If the user supplies an invalid option, the parser will automatically catch it and throw an informative message.

Kernels



www.inl.gov

Kernel

- A “Kernel” is a piece of physics.
 - It can represent one or more operators / terms in a PDE.
- A Kernel **MUST** override `computeQpResidual()`
- A Kernel **can** optionally override:
 - `computeQpJacobian()`
 - `computeQpOffDiagJacobian()`

Anatomy of a MOOSE object

MyObject.h

```
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include "BaseClass.h"

class MyObject;

template<>
InputParameters validParams<MyObject>();

class MyObject : public BaseClass
{
public:
    MyObject(const std::string & name,
             InputParameters params);

protected:
    virtual SomeType inheritedMethod();
};

#endif // MYOBJECT_H
```

MyObject.C

```
#include "MyObject.h"

template<>
InputParameters validParams<MyObject>()
{
    InputParameters params = validParams<BaseClass>();
    return params;
}

MyObject::MyObject(const std::string & name,
                   InputParameters params):
    BaseClass(name, params)
{}

SomeType
MyObject::inheritedMethod()
{
    // Do Stuff!
}
```

(Some) Values Available to Kernels

- `_u, _grad_u`
 - Value and gradient of variable this Kernel is operating on.
- `_phi, _grad_phi`
 - Value (ϕ) and gradient ($\nabla\phi$) of the trial functions at the q-points.
- `_test, _grad_test`
 - Value (ψ) and gradient ($\nabla\psi$) of the test functions at the q-points.
- `_q_point`
 - XYZ coordinates of the current q-point.
- `_i, _j`
 - Current shape functions for test and trial functions respectively.
- `_qp`
 - Current quadrature point index.
- `_current_elem`
 - A pointer to the current element that is being operated on.
- And more!

Diffusion

$$-\nabla \cdot \nabla u - f = 0$$

$$u|_{\partial\Omega_1} = g_1$$

$$\nabla u \cdot \hat{n}|_{\partial\Omega_2} = g_2$$

- Multiply by test function, integrate

$$(-\nabla \cdot \nabla u, \psi_i) - (f, \psi_i) = 0$$

- Integrate by parts

$$(\nabla u, \nabla \psi_i) - (f, \psi_i) - \langle g_2, \psi_i \rangle = 0$$

- Jacobian

$$(\nabla \phi_j, \nabla \psi_i)$$

Diffusion

Diffusion.h

```
#ifndef DIFFUSION_H
#define DIFFUSION_H

#include "Kernel.h"

class Diffusion;

template<>
InputParameters validParams<Diffusion>();

class Diffusion : public Kernel
{
public:
    Diffusion(const std::string & name,
              InputParameters params);

protected:
    virtual Real computeQpResidual();
    virtual Real computeQpJacobian();
};

#endif // DIFFUSION_H
```

Diffusion.C

```
#include "Diffusion.h"

template<>
InputParameters validParams<Diffusion>()
{
    InputParameters params = validParams<Kernel>();
    return params;
}

Diffusion::Diffusion(const std::string & name,
                     InputParameters params):
    Kernel(name, params)
{}

Real
Diffusion::computeQpResidual()
{
    return _grad_test[_i][_qp]*_grad_u[_qp];
}

Real
Diffusion::computeQpJacobian()
{
    return _grad_test[_i][_qp]*_grad_phi[_j][_qp];
}
```

Kernel Registration

- Before a Kernel is available for use, it must be registered with the Factory inside your main application .C file (e.g., SomeApp.C).

```
#include "Diffusion.h"

int SomeApp::registerObjects(Factory & factory)
{
    ...
    registerKernel(Diffusion);
    ...
}
```

Convection Diffusion

- In Example 2, we will register and add a new Convection Kernel

- Strong Form:

$$\begin{cases} -\nabla \cdot \nabla u + \mathbf{v} \cdot \nabla u = 0 \\ \text{BCs} \end{cases}$$

- Multiply by test function ψ_i and integrate over the domain:

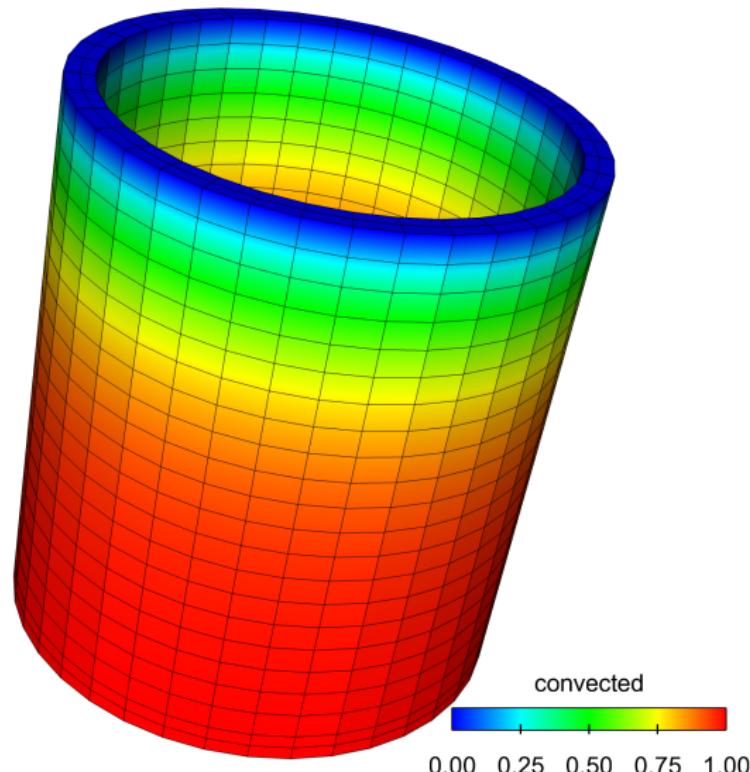
$$(-\nabla \cdot \nabla u, \psi_i) + (\mathbf{v} \cdot \nabla u, \psi_i) = 0$$

- Integrate the first term by parts to get our final weak form:

$$(\nabla u, \nabla \psi_i) - \langle \nabla u \cdot \hat{\mathbf{n}}, \psi_i \rangle + (\mathbf{v} \cdot \nabla u, \psi_i) = 0$$

Look at Example 2 (page E11)

Example 2 Output



Multiphysics Coupling



www.inl.gov

Multiphysics Coupling – Nonlinear Transient Heat Conduction

- Strong form:

$$\begin{cases} \rho C_p \frac{\partial T}{\partial t} - \nabla \cdot k \nabla T - q = 0 \\ \text{Dirichlet BCs} \end{cases}$$

- Weak form:

$$\left(\rho C_p \frac{\partial T}{\partial t}, \psi_i \right) + (k \nabla T, \nabla \psi_i) - (q, \psi_i) = 0$$

Multiphysics Coupling – Nonlinear Transient Oxygen Non-Stoichiometry

- Strong form:

$$\left\{ \begin{array}{l} \frac{\partial s}{\partial t} - \nabla \cdot \left(D \left(\nabla s + \frac{sQ^*}{FRT^2} \nabla T \right) \right) = 0 \\ \text{Dirichlet BCs} \end{array} \right.$$

- Weak form:

$$\left(\frac{\partial s}{\partial t}, \psi_i \right) + \left(D \left(\nabla s + \frac{sQ^*}{FRT^2} \nabla T \right), \nabla \psi_i \right) = 0$$

Multiphysics Coupling (Steady form)

- Coupled steady heat conduction / steady oxygen diffusion
- Fully coupled nonlinear residual

$$\mathbf{R}_i(\mathbf{T}, \mathbf{s}) = \begin{bmatrix} (k(\mathbf{T}, \mathbf{s}) \nabla \mathbf{T}, \nabla \psi_i) - (\mathbf{q}, \psi_i) \\ (D(\mathbf{T}, \mathbf{s})(\nabla \mathbf{s} + f(\mathbf{T}, \mathbf{s}) \nabla \mathbf{T}), \nabla \psi_i) \end{bmatrix} = 0$$

- Apply JFNK to this residual

Simplified Coupling Example

- Weak forms:

$$\begin{aligned}(\nabla u, \nabla \psi_i) + (\nabla \textcolor{red}{v} \cdot \nabla u, \psi_i) &= 0 \\ (\nabla \textcolor{red}{v}, \nabla \psi_i) &= 0\end{aligned}$$

- Each “physics” sums into residual
- One-way coupling: u -equation depends on v

Coupling Parameters

- To couple a Kernel (or any other MOOSE object) to other variables you must declare the coupling in the `validParams` method:

```
params.addCoupledVar("temperature", "doc_string");
```

- You may then specify your coupling in the input file:

```
[./temp]
  order = FIRST
  family = LAGRANGE
[...]
```

```
...
```

```
[./coupled_diffusion]
  type = ExampleDiffusion
  variable = u
  temperature = temp
[...]
```

- Important!**

- “`temp`” is the (arbitrary) name of the variable in the *input file*
- “`temperature`” is the name used by the kernel (always the same)

Coupling in Kernels

- The coupling of values and gradients into Kernels is done by calling the following functions in the initialization list of the constructor:
 - `coupledValue()`
 - `coupledValueOld()`
 - `coupledValueOlder()`
 - `coupledGradient()`
 - `coupledGradientOld()`
 - `coupledGradientOlder()`
 - `coupledDot()`
 - ...
- These functions return references that you hold onto in the class.
- The references are then used in `computeQpResidual()` / `computeQpJacobian()`

Default Coupling Parameters

- To aid in rapid development, debugging, and flexibility, MOOSE allows you to supply default scalar values where you might expect a coupled value.

```
...  
params.addCoupledVar("temperature", 300, "doc_string");  
...
```

- If a variable is not supplied through the input file, a properly sized variable containing the default value will be made available to you at each integration point in your domain.
- Additionally, you may also supply a Real value in the input file in lieu of a coupled variable name.
- Consider using this feature to decouple your non-linear problems for troubleshooting.

Convection.h / .C

```
#ifndef CONVECTION_H
#define CONVECTION_H

#include "Kernel.h"

class Convection;

template<>
InputParameters validParams<Convection>();

class Convection : public Kernel
{
public:

    Convection(const std::string & name,
               InputParameters parameters);

protected:
    virtual Real computeQpResidual();
    virtual Real computeQpJacobian();

private:
    VariableGradient & _grad_some_variable;
};

#endif //CONVECTION_H
```

```
#include "Convection.h"

template<>
InputParameters validParams<Convection>()
{
    InputParameters params = validParams<Kernel>();
    params.addRequiredCoupledVar("some_variable", "Doc");
    return params;
}

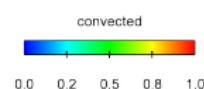
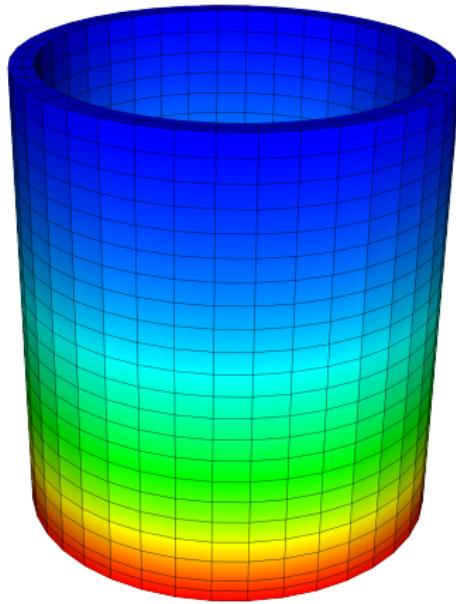
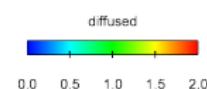
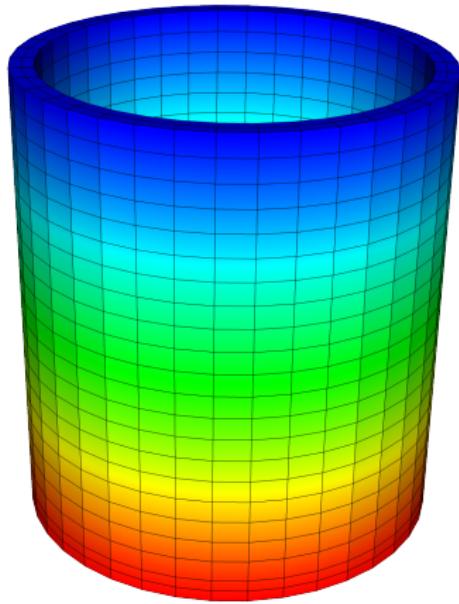
Convection::Convection(const std::string & name,
                      InputParameters parameters):
    Kernel(name, parameters),
    _grad_some_variable(coupledGradient("some_variable"))
{}

Real Convection::computeQpResidual()
{
    return _test[_i][_qp]*(_grad_some_variable[_qp]*_grad_u[_qp]);
}

Real Convection::computeQpJacobian()
{
    return _test[_i][_qp]*(_grad_some_variable[_qp]*_grad_phi[_j][_qp]);
}
```

Look at Example 3 (page E19)

Example 3 Output



Example 3: 1D exact solution

- A simplified 1D analog of this problem is given by

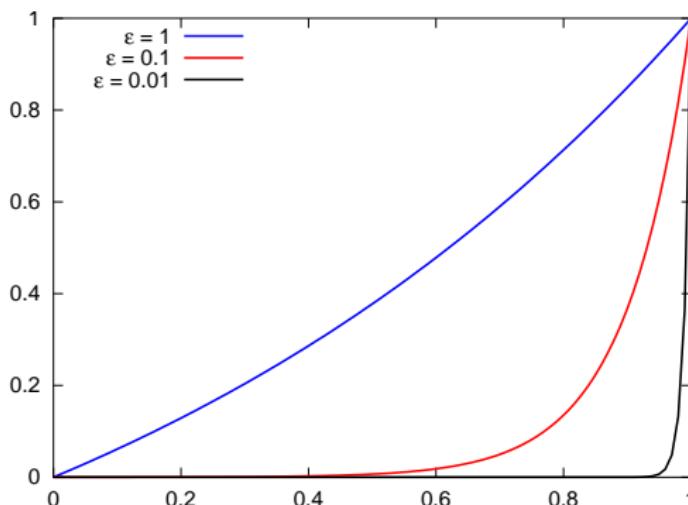
$$-\epsilon \frac{d^2 u}{dx^2} + \frac{du}{dx} = 0$$

$$u(0) = 0$$

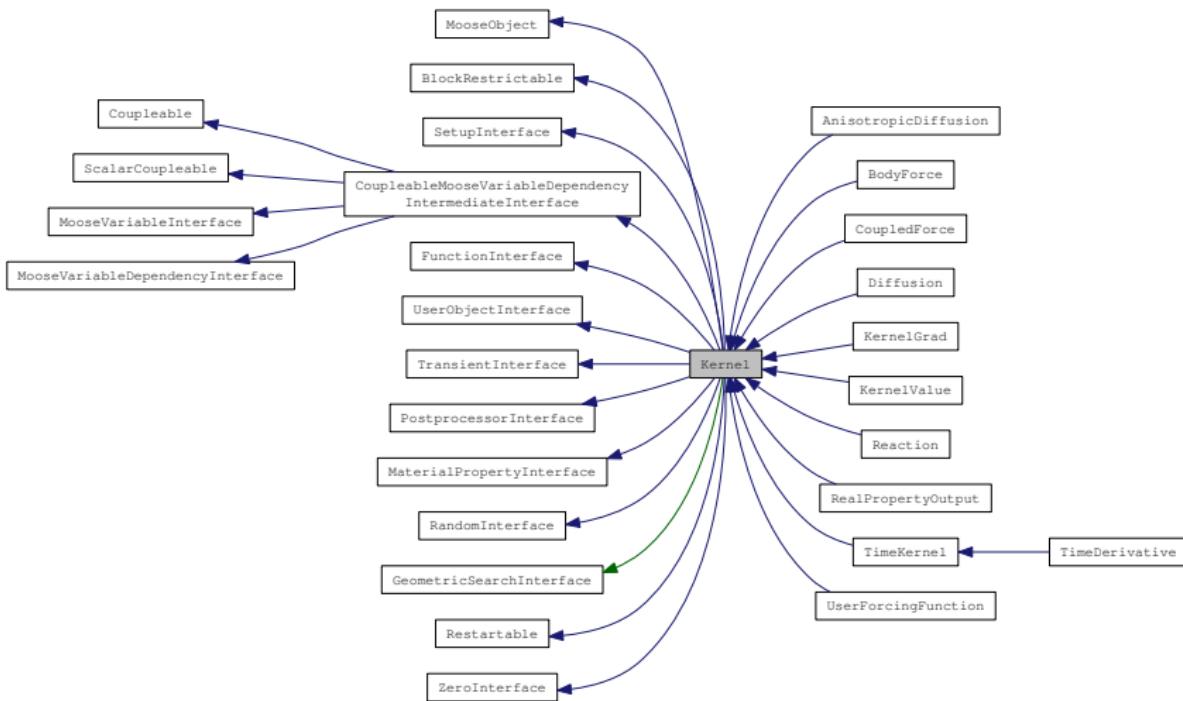
$$u(1) = 1$$

- The exact solution to this problem is

$$u = \frac{\exp\left(\frac{x}{\epsilon}\right) - 1}{\exp\left(\frac{1}{\epsilon}\right) - 1}$$



Kernel Inheritance



Boundary Conditions



www.inl.gov

Boundary Condition

- A `BoundaryCondition` provides a residual (and optionally a Jacobian) on a side of a domain.
- The structure is very similar to kernels
 - `computeQpResidual / Jacobian()`
 - Parameters
 - Coupling
- The only difference is that some BCs are NOT integrated over the boundary... and instead specify values on boundaries (Dirichlet).
- IntegratedBCs inherit from `IntegratedBC`.
- Non-integrated BCs inherit from `NodalBC`.

(Some) Values Available to BCs

Integrated BCs:

- `_u`, `_grad_u`: Value and gradient of variable this BC is operating on.
- `_phi`, `_grad_phi`: Value (ϕ) and gradient ($\nabla\phi$) of the trial functions at the q-points.
- `_test`, `_grad_test`: Value (ψ) and gradient ($\nabla\psi$) of the test functions at the q-points.
- `_q_point`: XYZ coordinates of the current q-point.
- `_i`, `_j`: Current shape function indices for test and trial functions respectively.
- `_qp`: Current quadrature point index.
- `_normals`: Normal vector at the current quadrature point.
- `_boundary_id`: The boundary ID this BC is currently computing on.
- `_current_elem`: A pointer to the element that is being computed on.
- `_current_side`: The side number of `_current_elem` that is currently being computed on.

Non-Integrated BCs:

- `_u`
- `_qp`
- `_boundary_id`
- `_current_node`: A pointer to the current node that is being operated on.
- And more!

Coupling and BCs

- The coupling of values and gradients into BCs is done the same way as in kernels and materials:
 - coupledValue()
 - coupledValueOld()
 - coupledValueOlder()
 - coupledGradient()
 - coupledGradientOld()
 - coupledGradientOlder()
 - coupledDot()

Dirichlet BCs

- Set a condition on the **value** of a variable on a boundary.
- Usually... these are NOT integrated over the boundary.
- Non-integrated BCs can't provide a Jacobian!

$$u = g_1 \quad \text{on} \quad \partial\Omega_1$$

Becomes:

$$u - g_1 = 0 \quad \text{on} \quad \partial\Omega_1$$

- In the following example:

$$u = \alpha v \quad \text{on} \quad \partial\Omega_2$$

And therefore:

$$u - \alpha v = 0 \quad \text{on} \quad \partial\Omega_2$$

CoupledDirichletBC.h / .C

```
#ifndef COUPLEDDIRICHLETBC_H
#define COUPLEDDIRICHLETBC_H

#include "NodalBC.h"

//Forward Declarations
class CoupledDirichletBC;

template<>
InputParameters validParams<CoupledDirichletBC>();

class CoupledDirichletBC : public NodalBC
{
public:
    CoupledDirichletBC(const std::string & name,
                       InputParameters parameters);

protected:
    virtual Real computeQpResidual();

private:
    Real _alpha;

    VariableValue & _some_var_val;
};

#endif //COUPLEDDIRICHLETBC_H
```

```
#include "CoupledDirichletBC.h"

template<>
InputParameters validParams<CoupledDirichletBC>()
{
    InputParameters params = validParams<NodalBC>();
    params.addParam<Real>("alpha", 0.0, "Doc");
    params.addRequiredCoupledVar("some_var", "Doc");
    return params;
}

CoupledDirichletBC::
    CoupledDirichletBC(const std::string & name,
                      InputParameters parameters) :
    NodalBC(name, parameters),
    _alpha(getParam<Real>("alpha")),
    _some_var_val(coupledValue("some_var"))
{}

Real
CoupledDirichletBC::computeQpResidual()
{
    return _u[_qp] - _alpha*_some_var_val[_qp];
}
```

Integrated BCs

- Integrated BCs (including Neumann BCs) are actually integrated over the external face of an element.
- Their residuals look similar to kernels:

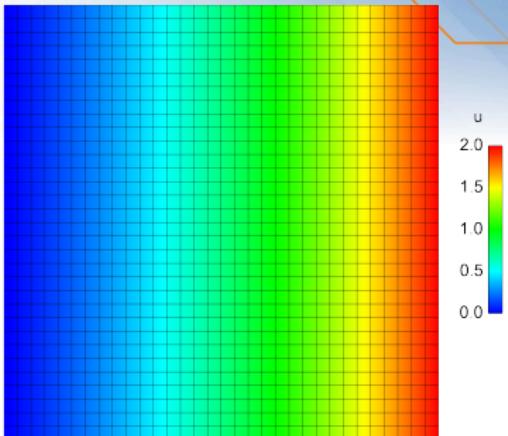
$$\begin{cases} (\nabla u, \nabla \psi_i) - (f, \psi_i) - \langle \nabla u \cdot \hat{\mathbf{n}}, \psi_i \rangle = 0 & \forall i \\ \nabla u \cdot \hat{\mathbf{n}} = g_1 \quad \text{on} \quad \partial\Omega \end{cases}$$

Becomes:

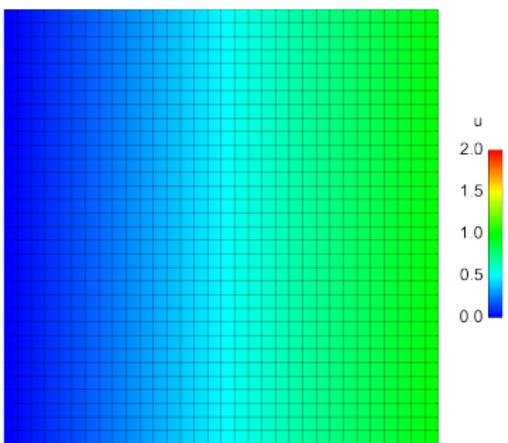
$$(\nabla u, \nabla \psi_i) - (f, \psi_i) - \langle g_1, \psi_i \rangle = 0 \quad \forall i$$

- Also note that if $\nabla u \cdot \hat{\mathbf{n}} = 0$ then the boundary integral is zero (sometimes known as the “natural boundary condition”).
- In the following example, $g_1 = \alpha v$.

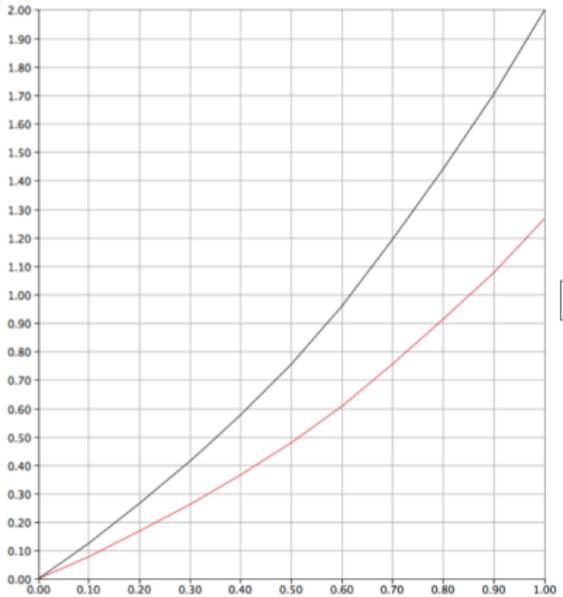
Look at Example 4 (page E25)



u
2.0
1.5
1.0
0.5
0.0



u
2.0
1.5
1.0
0.5
0.0



/u
/u

Periodic BCs

- Periodic boundary conditions are useful for modeling quasi-infinite domains and systems with conserved quantities.
- MOOSE has full support for Periodic BCs
 - 1D, 2D, and 3D.
 - With mesh adaptivity.
 - Can be restricted to specific variables.
 - Supports arbitrary translation vectors for defining periodicity.

Periodic BCs

```
[BCs]
[./Periodic]
[./all]
variable = u

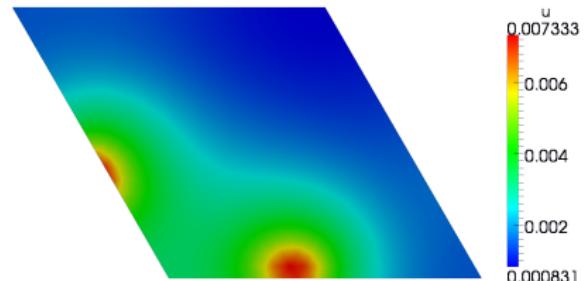
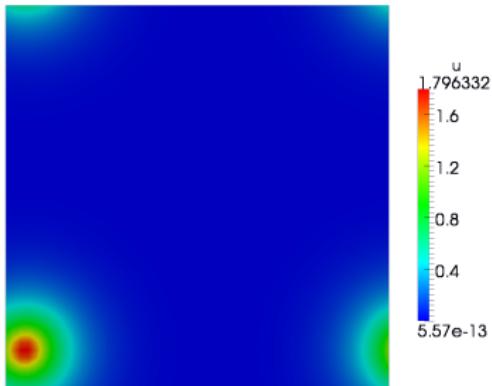
# Works for any regular orthogonal
# mesh with defined boundaries
auto_direction = 'x y'
[...]
[...]
[]
```

```
[BCs]
[./Periodic]
[./x]
primary = 1
secondary = 4
transform_func = 'tr_x tr_y'
inv_transform_func = 'itr_x itr_y'
[...]
[...]
[]
```

- Normal usage: with an axis-aligned mesh, use `auto_direction` to supply the coordinate directions to wrap.
- Advanced usage: specify a translation or transformation function.

```
[BCs]
[./Periodic]
[./x]
variable = u
primary = 'left'
secondary = 'right'
translation = '10 0 0'
[...]
[...]
[]
```

Periodic BCs Output



h-Adaptivity



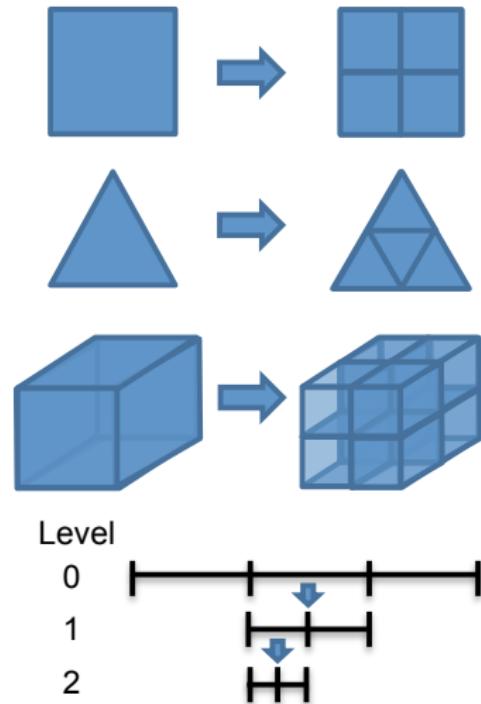
www.inl.gov

h-Adaptivity

- h-Adaptivity attempts to cluster mesh in areas of relatively high error in an effort to better capture the solution.
- The idea is to put more DOFs where the error is highest... not wasting effort anywhere the solution is already well captured.
- This is achieved through refinement (splitting cells into smaller cells) and coarsening (undoing refinement to create larger elements).
- In order to know where to refine, an **Indicator** must be employed.
 - These attempt to report the relative error in each cell.
- Next, a **Marker** is utilized to decide which cells to refine and which to coarsen.
- This process can be repeated for steady state solves, or done every time step during a transient simulation.

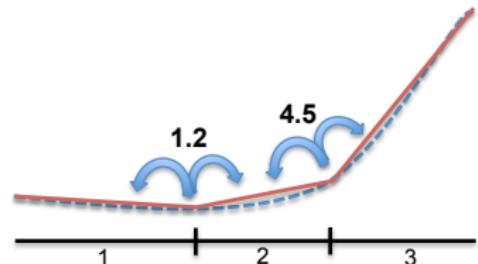
Refinement Patterns

- MOOSE employs “self-similar”, isotropic refinement patterns.
- When an element is marked for refinement it is split into elements of the same type.
- For example, when using Quad4 elements, four “children” elements are created when the element is refined.
- Coarsening happens in reverse, removing children to get back to the “parent” element.
- The original mesh starts at refinement level “0”.
- Every time an element is split, its children gain a refinement level.



Indicators

- Indicators attempt to report a relative amount of “error” for each element.
- Built-in Indicators include:
 - GradientJumpIndicator:** Jump in the gradient of a variable across element edges (pictured to the right). A good “curvature” indicator that works well over a wide range of problems.
 - FluxJumpIndicator:** Same as GradientJump except it can multiply by a scalar diffusivity (like thermal_conductivity) to compute the real “flux” jump.
 - LaplacianJumpIndicator:** Jump in the second derivative of a variable. Only useful for C^1 shape functions.
 - AnalyticIndicator:** Computes the difference between the finite element solution and a given Function that usually represents the analytic solution to the problem.
- In higher dimensions, the gradient jump is integrated around element edges to find contributions to each connected element.



Elem	Error
1	1.2
2	5.7
3	4.5

Markers

- After an Indicator has computed the error for each element, a decision to refine or coarsen elements must be made.
- The Marker class is used for this purpose. Built-in Markers include:
 - ErrorFractionMarker: Selects elements based on how much each element contributes to the total error (pictured to right).
 - ErrorToleranceMarker: Refine if error over a specified value. Coarsen if under.
 - ValueThresholdMarker: Refine if variable value is above a specific value. Coarsen if under.
 - UniformMarker: Refine or coarsen all elements.
 - BoxMarker: Refine or coarsen inside or outside a given box.
 - ComboMarker: Combine several Markers into one.
- Markers produce an element field that can be viewed in your visualization utility.
- Custom Markers are easy to create by inheriting from the Marker base class.

Elem	Error
7	27
9	20
6	12
1	8
4	6
8	5
10	5
2	4
3	2
Total:89	

Refine Fraction=0.6
(89*0.6=53.4)

Coarsen Fraction=0.1
(89*0.1=8.9)

Input File Syntax

```
[Adaptivity]
marker = errorfrac
steps = 2

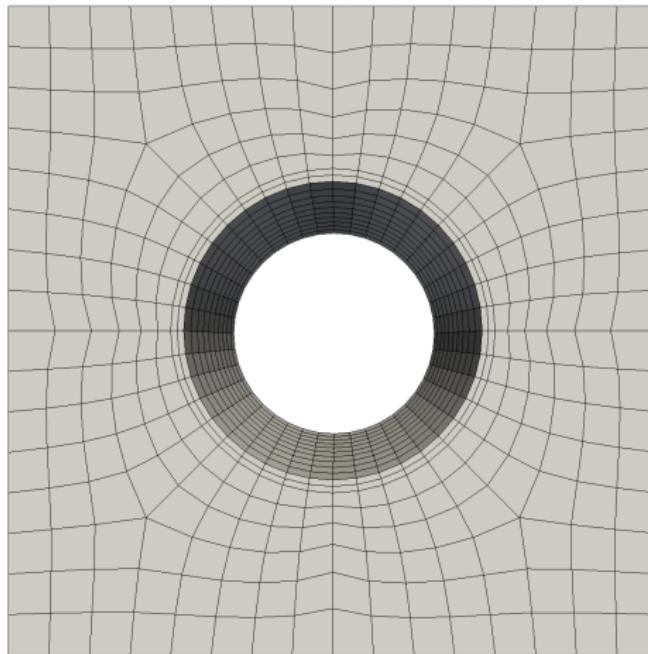
[./Indicators]
[./error]
    type = GradientJumpIndicator
    variable = convected
[../]
[../]

[./Markers]
[./errorfrac]
    type = ErrorFractionMarker
    refine = 0.5
    coarsen = 0
    indicator = error
[../]
[../]
[]
```

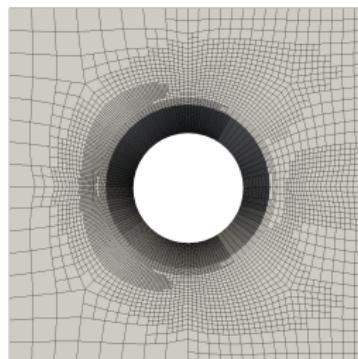
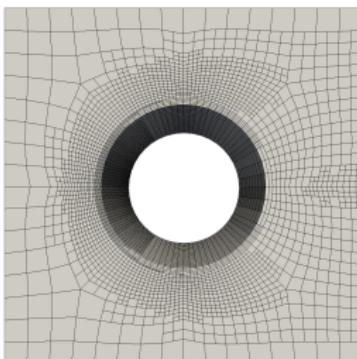
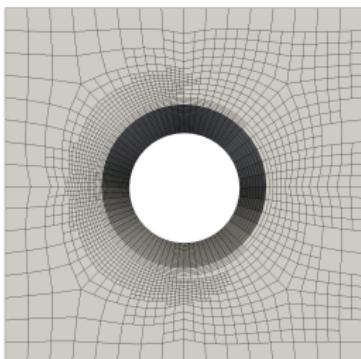
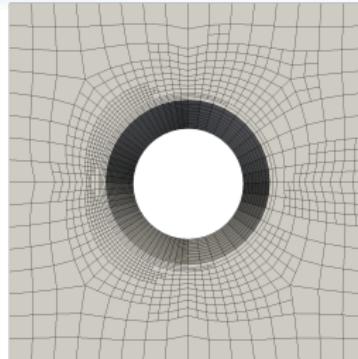
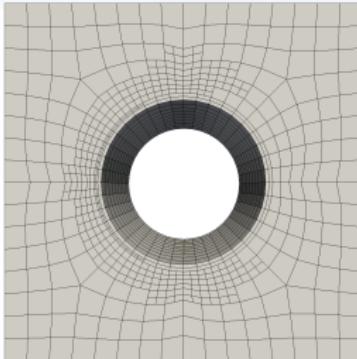
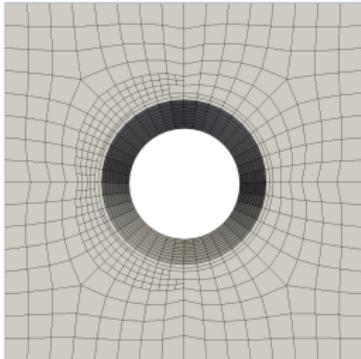
- To enable adaptivity, create an **Adaptivity block**.
- The Adaptivity block itself has several parameters:
 - **marker**: (Optional) Name of the Marker to use. *If not set, no mesh adaptivity will be done.*
 - **steps**: Number of refinement steps to do in a steady state calculation. Ignored in transient.
- Adaptivity has two sub-blocks: Indicators and Markers. Within these blocks, you can specify multiple Indicators and Markers that will be active in the simulation.

Look at Example 5 (page E39)

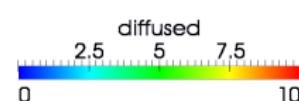
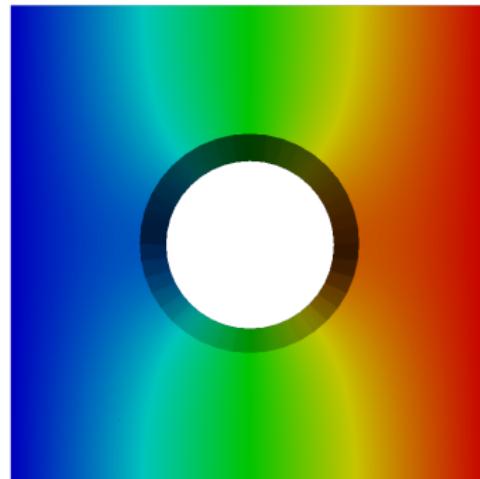
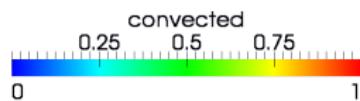
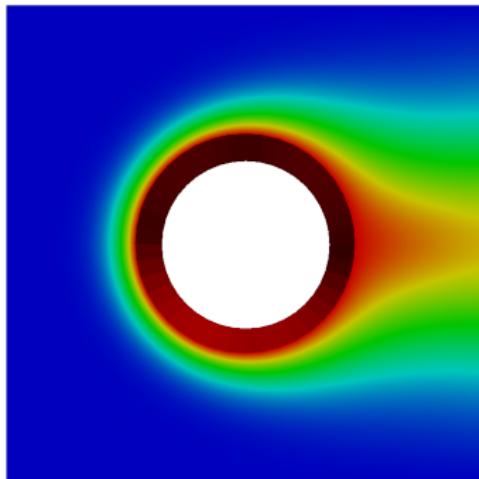
Initial grid



Adaptivity Steps



Example 5 Output



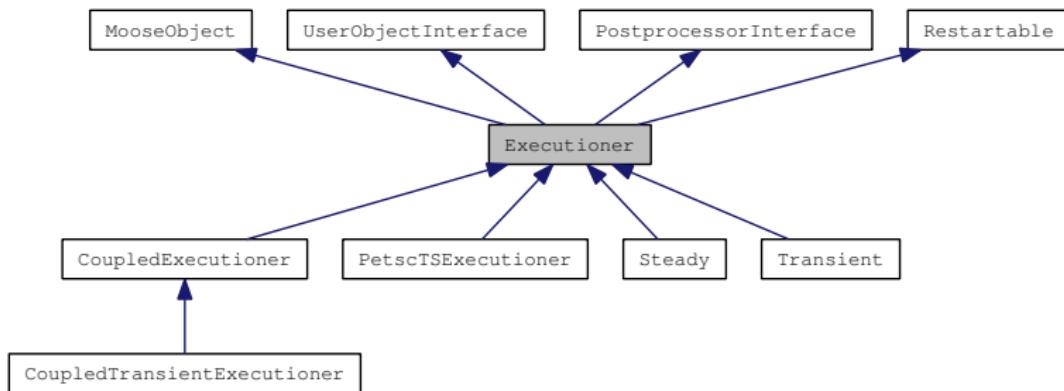
Executioners



www.inl.gov

Built-in Executioners

- There are two main types of Executioners: Steady and Transient.
- MOOSE provides a number of built-in executioners, but you can extend this system and add your own.



Steady-state Executioner

- Steady-state executioners generally solve the nonlinear system just once.
- However, our steady-state executioner can solve the nonlinear system multiple times while adapting the mesh to better fit the solution as was seen in the previous example.

Some Common Options for All Executioners

- There are a number of options that appear in the executioner block that control the solver. The best way to view these is through peacock. Here are a few common options:

l_tol	Linear Tolerance
l_max_its	Max Linear Iterations
nl_rel_tol	Nonlinear Relative Tolerance
nl_max_its	Max Nonlinear Iterations
...	

Transient Executioners

- Transient executioners solve the nonlinear system at least once per time step.
- Frequently used options Transient Executioners

dt	Starting time step size
num_steps	Number of time steps
start_time	The start time of the simulation
end_time	The end time of the simulation
scheme	Time integration scheme (discussed next)
...	

Transient Analysis

- Currently MOOSE includes support for these TimeIntegrator objects:
 1. Backward Euler
 2. BDF2
 3. Crank-Nicolson
 4. Implicit-Euler
 5. Explicit-Euler
- Each one of these supports adaptive time stepping.
- They are treated as TimeDerivative kernels for the time derivative terms that appear in PDEs:

$$\frac{\partial u}{\partial t} - \nabla \cdot \nabla u + \mathbf{v} \cdot \nabla u = 0$$

Becomes

$$\left(\frac{\partial u}{\partial t}, \psi_i \right) + (\nabla u, \nabla \psi_i) - \langle \nabla u \cdot \hat{\mathbf{n}}, \psi_i \rangle + (\mathbf{v} \cdot \nabla u, \psi_i) = 0$$

Custom Time Kernel

- If you need to provide a coefficient to the transient term (such as `density*specific_heat` for heat conduction) **inherit** from one of the `TimeDerivative` kernels that are in MOOSE.
- In `computeQpResidual/Jacobian()` multiply your coefficient by `TimeDerivative::computeQpResidual/Jacobian()`.

Custom Time Kernel

```
#include "ExampleTimeDerivative.h"

template<>
InputParameters validParams<ExampleTimeDerivative>()
{
    InputParameters params = validParams<TimeDerivative>();
    params.addParam<Real>("time_coefficient", 1.0, "Time Coefficient");
    return params;
}

ExampleTimeDerivative::ExampleTimeDerivative(const std::string & name,
                                              InputParameters parameters):
    TimeDerivative(name, parameters),
    // This kernel expects an input parameter named "time_coefficient"
    _time_coefficient(getParam<Real>("time_coefficient"))
{ }

Real
ExampleTimeDerivative::computeQpResidual()
{
    return _time_coefficient*TimeDerivative::computeQpResidual();
}

Real
ExampleTimeDerivative::computeQpJacobian()
{
    return _time_coefficient*TimeDerivative::computeQpJacobian();
}
```

Transient Input Syntax

- Add appropriate time derivative Kernels.
- Switch your executioner type to “Transient”
- Add a few required parameters to the Executioner block

Real time: The start time for the simulation.

Real dt: The initial time step size.

int t_step: The initial time step

string scheme: Time integrator scheme

- ‘crank-nicolson’

- ‘backward-euler’ – default if you do not specify the scheme

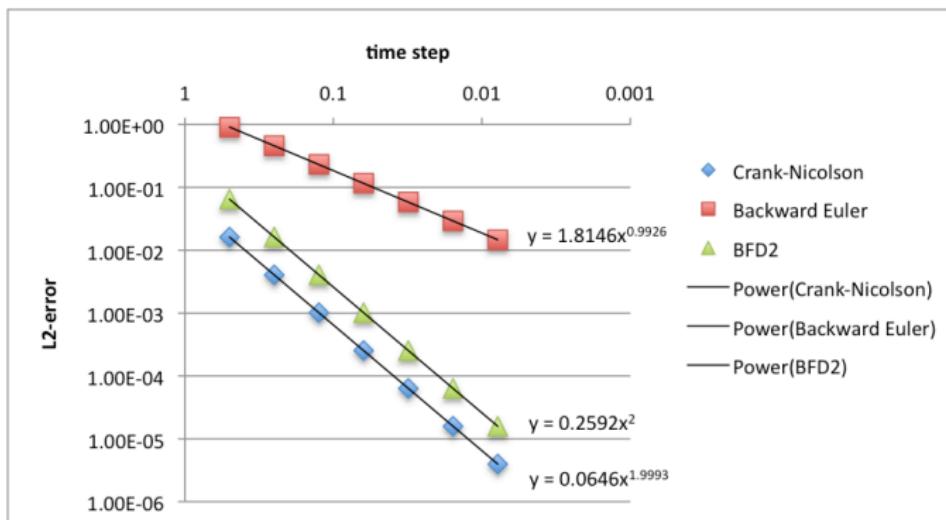
- ‘bdf2’

Convergence Rates

- Test problem:

$$\frac{\partial u}{\partial t} - \nabla \cdot \nabla u - f = 0 \text{ in } \Omega$$

where f is chosen so the exact solution is given by $u = t^3(x^2 + y^2)$.



Adaptive Time Step

- Set TimeStepper type to DT2

- Algorithm

- Takes a time step of size dt to get \hat{u}_{n+1} from u_n
- Takes two time steps of length $dt/2$ to get u_{n+1} from u_n
- Calculate local relative time discretization error estimate

$$\hat{e}_n \equiv \frac{\|u_{n+1} - \hat{u}_{n+1}\|_2}{\max(\|u_{n+1}\|_2, \|\hat{u}_{n+1}\|_2)}$$

- Obtain global relative time discretization error estimate $e_n \equiv \frac{\hat{e}_n}{dt}$
- Adaptivity is based on target error tolerance e_{TOL} and a maximum acceptable error tolerance e_{MAX} .
 - If we achieve $e_n < e_{MAX}$, then we continue with a new time step size

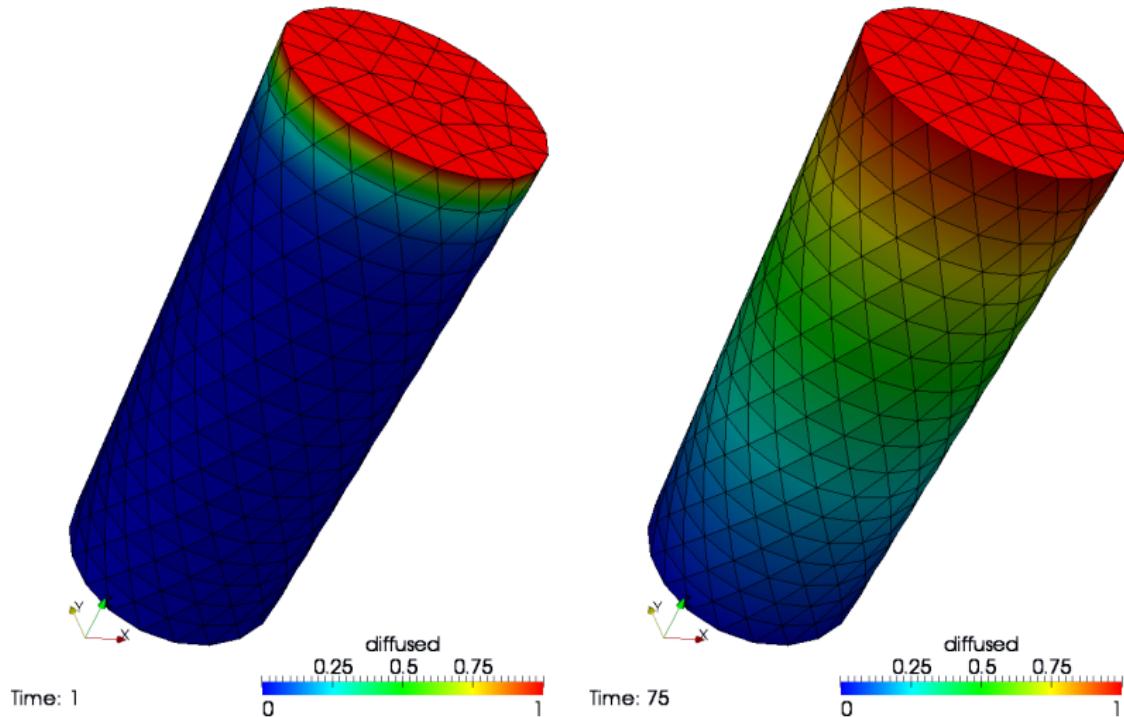
$$dt_{n+1} \equiv dt_n \cdot \left(\frac{e_{TOL}}{e_n} \right)^{1/p}$$

where p is the global convergence rate of the time stepping scheme.

- If we do not achieve $e_n < e_{MAX}$, or if the nonlinear solver fails, we reject the time step and shrink dt .
- Parameters e_{TOL} and e_{MAX} can be specified in the input file as `e_tol` and `e_max` (in the Executioner block).

Look at Example 6 (page E45)

Watch the Movie Using Paraview On a Laptop.



Initial Conditions

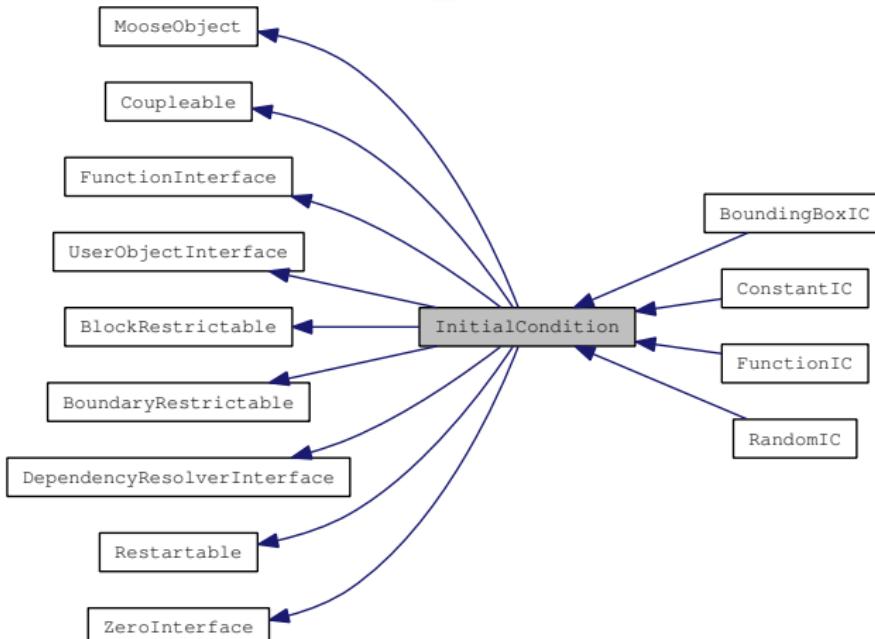


www.inl.gov

Initial Condition System

- Allows for creation of arbitrary initial conditions.
 - Can be spatially dependent.
- Allows for selection of initial condition from input file.
- If using an ExodusII mesh you can read the value of a variable from a previous simulation
- Allows initial conditions to be coupled together
 - Automatic dependency resolution

Base Class



- `value(const Point & p)`
 - Returns the value the variable should have at point `p`.
 - All `InitialCondition` classes **MUST** override this!

ExampleIC.h/.C

```
#ifndef EXAMPLEIC_H
#define EXAMPLEIC_H

#include "InitialCondition.h"

class ExampleIC;
template<>
InputParameters validParams<ExampleIC>();

class ExampleIC : public InitialCondition
{
public:
ExampleIC(const std::string & name,
          InputParameters parameters);

virtual Real value(const Point & p);

private:
  Real _coefficient;
};

#endif //EXAMPLEIC_H
```

```
#include "ExampleIC.h"

template<>
InputParameters validParams<ExampleIC>()
{
  InputParameters params = validParams<InitialCondition>();
  params.addRequiredParam<Real>("coefficient", "A coef");
  return params;
}

ExampleIC::ExampleIC(const std::string & name,
                     InputParameters parameters):
  InitialCondition(name, parameters),
  _coefficient(getParam<Real>("coefficient"))
()

Real
ExampleIC::value(const Point & p)
{
  // 2.0 * c * |x|
  return 2.0*_coefficient*std::abs(p(0));
}
```

Using ExampleIC

Now Register It:

```
#include "ExampleIC.h"  
...  
registerInitialCondition(ExampleIC);
```

And use it in an input file:

```
...  
[ICs]  
  [./mat_1]  
    type = ExampleIC  
    variable = u  
    coefficient = 2.0  
    block = 1  
  [../]  
  
  [./mat_2]  
    type = ExampleIC  
    variable = u  
    coefficient = 10.0  
    block = 2  
  [../]  
...
```

Initial Condition Shortcut Syntax

Constant Initial Conditions

```
...
[Variables]
active = 'u'

[./u]
order = FIRST
family = LAGRANGE
# For simple constant ICs
initial_condition = 10
[...]
...
```

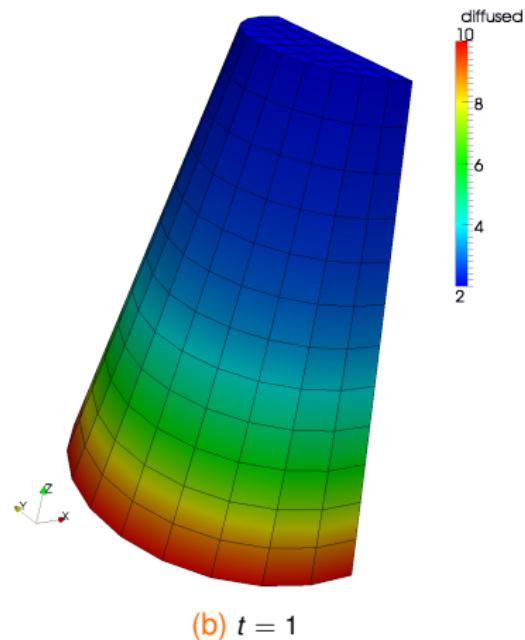
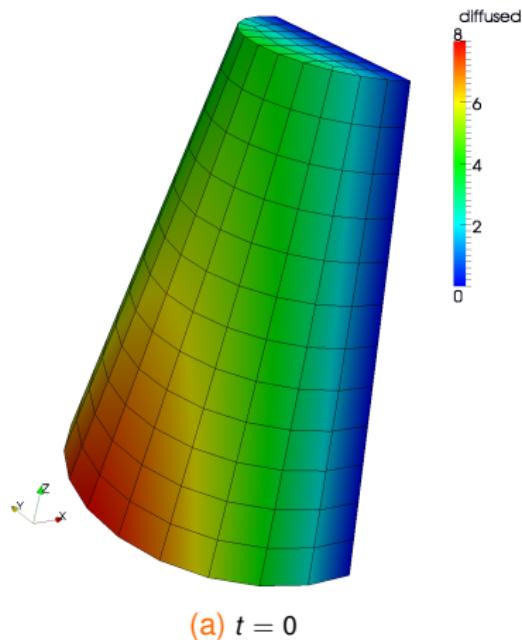
“Restart” from an existing solution

```
...
[Variables]
active = 'u'

[./u]
order = FIRST
family = LAGRANGE
# For reading a solution
# from an ExodusII file
initial_from_file_var = diffused
initial_from_file_timestep = 2
[...]
...
```

Look at Example 7 (page E53)

Example 7 Output



Materials



www.inl.gov

Materials

- Material objects declare coefficients for use by kernels.
- Use `declareProperty<TYPE>()` to declare a material property
- One virtual to override: `computeQpProperties()`
 - Compute all of the declared properties at one quadrature point.
- Can use coupled variables using the same functions that Kernel uses. (like `coupledValue()` et al.)
- To use a material property use `getMaterialProperty<TYPE>()` in a Kernel.

Example 8 Overview

- This is a convection-diffusion problem with a non-linear material property.
- The `ExampleDiffusion` kernel uses a coefficient produced from a linearly interpolated tabulated Material property.
- The `ExampleMaterial` object couples in the gradient of the “diffused” variable and uses it to make a gradient material property that will be coupled into the “convected” variable.
- The `Convection` kernel uses the gradient material property as a velocity vector.

ExampleMaterial.h/C

```
#ifndef EXAMPLEMATERIAL_H
#define EXAMPLEMATERIAL_H

#include "Material.h"
#include "LinearInterpolation.h"

//Forward Declarations
class ExampleMaterial;

template<>
InputParameters validParams<ExampleMaterial>();

class ExampleMaterial : public Material
{
public:
    ExampleMaterial(const std::string & name,
                   InputParameters parameters);

protected:
    virtual void computeQpProperties();

private:
    MaterialProperty<Real> & _diffusivity;
    MaterialProperty<RealGradient> & _conv_vel;
    // Nonlinear coupling
    VariableGradient & _diffusion_gradient;
    // Table data property
    LinearInterpolation _piecewise_func;
};

#endif //EXAMPLEMATERIAL_H
```

```
#include "ExampleMaterial.h"

template<>
InputParameters validParams<ExampleMaterial>()
{
    InputParameters params = validParams<Material>();
    params.addRequiredParam<std::vector<Real>>("independent_vals", "");
    params.addRequiredParam<std::vector<Real>>("dependent_vals", "");
    params.addCoupledVar("diffusion_gradient", "Doc");
    return params;
}

ExampleMaterial::ExampleMaterial(const std::string & name,
                                 InputParameters parameters):
    Material(name, parameters),
    // Declare that this material is going to provide a Real prop
    _diffusivity(declareProperty<Real>("diffusivity")),
    // Declare that this material is going to provide a RealGradient
    _conv_vel(declareProperty<RealGradient>("convection_velocity")),
    // Get the reference to the variable coupled into this Material
    _diffusion_gradient(isCoupled("diffusion_gradient") ?
        coupledGradient("diffusion_gradient") : _grad_zero),

    _piecewise_func(getParam<std::vector<Real>>("independent_vals"),
                    getParam<std::vector<Real>>("dependent_vals"))
()

void
ExampleMaterial::computeQpProperties()
{
    // sample the function using the z coordinate
    _diffusivity[_qp] = _piecewise_func.sample(_q_point[_qp](2));
    _conv_vel[_qp] = _diffusion_gradient[_qp];
}
```

ExampleDiffusion.h/C with Material Property

```
#ifndef EXAMPLEDIFFUSION_H
#define EXAMPLEDIFFUSION_H

#include "Diffusion.h"

//Forward Declarations
class ExampleDiffusion;

template<>
InputParameters validParams<ExampleDiffusion>();

class ExampleDiffusion : public Diffusion
{
public:

    ExampleDiffusion(const std::string & name,
                    InputParameters parameters);

protected:
    virtual Real computeQpResidual();
    virtual Real computeQpJacobian();

    MaterialProperty<Real> & _diffusivity;
};

#endif //EXAMPLEDIFFUSION_H
```

```
#include "ExampleDiffusion.h"

template<>
InputParameters validParams<ExampleDiffusion>()
{
    InputParameters params = validParams<Diffusion>();
    return params;
}

ExampleDiffusion::ExampleDiffusion(const std::string & name,
                                  InputParameters parameters)
:_diffusivity(getMaterialProperty<Real>("diffusivity"))
{ }

Real
ExampleDiffusion::computeQpResidual()
{
    return _diffusivity[_qp]*Diffusion::computeQpResidual();
}

Real
ExampleDiffusion::computeQpJacobian()
{
    return _diffusivity[_qp]*Diffusion::computeQpJacobian();
}
```

Convection.h/C with Material Property

```
#ifndef CONVECTION_H
#define CONVECTION_H

#include "Kernel.h"

class Convection;

template<>
InputParameters validParams<Convection>();

class Convection : public Kernel
{
public:

    Convection(const std::string & name,
               InputParameters parameters);

protected:
    virtual Real computeQpResidual();
    virtual Real computeQpJacobian();

private:

    MaterialProperty<RealGradient> & _velocity;
};

#endif //CONVECTION_H
```

```
#include "Convection.h"

template<>
InputParameters validParams<Convection>()
{
    InputParameters params = validParams<Kernel>();
    return params;
}

Convection::Convection(const std::string & name,
                      InputParameters parameters) :
    Kernel(name, parameters),

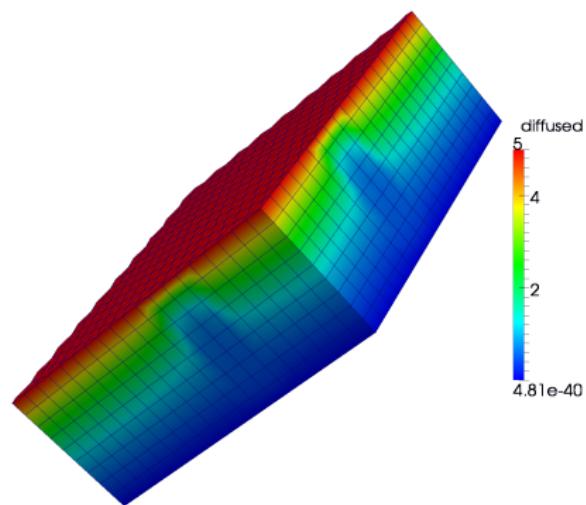
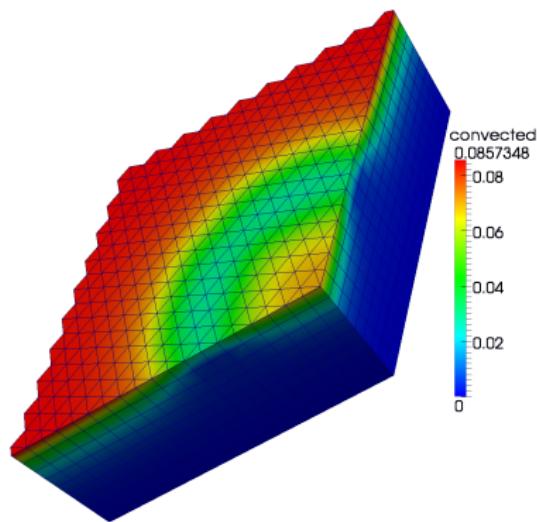
    // Retrieve a gradient material property to use for
    // the convection velocity
    _velocity(getMaterialProperty<RealGradient>
              ("convection_velocity"))

Real Convection::computeQpResidual()
{
    return _test[_i][_qp]*(_velocity[_qp]*_grad_u[_qp]);
}

Real Convection::computeQpJacobian()
{
    return _test[_i][_qp]*(_velocity[_qp]*_grad_phi[_j][_qp]);
}
```

Look at Example 8 (page E59)

Example 8 Output



Material Property Output

- Material properties may be automatically output via the input file.
 - outputs = none
Disables outputting of all properties to all possible outputs (default)
 - outputs = all
Enables outputting of all properties to all possible outputs
 - outputs = 'exodus'
Enables outputting of all properties to the output named "exodus" from the "[Outputs]" block
 - output_properties = 'property1 property2'
Limits the output to the properties listed
- Supported types include: Real, RealVectorValue, and RealTensorValue

Stateful Material Properties

- It can sometimes be useful to have properties at quadrature points that have old values.
- This situation often arises in solid mechanics for constitutive models like plasticity.
- Traditionally this type of value is called a “state variable”.
- In MOOSE they are “Stateful Material Properties”... i.e. material properties that have old values.
- To provide a material property with an old state just use
`declarePropertyOld<TYPE>()` and
`declarePropertyOlder<TYPE>()` passing in the same name as you did with `declareProperty<TYPE>()`.
- This will use more memory.

Stateful ExampleMaterial

```
#ifndef EXAMPLEMATERIAL_H
#define EXAMPLEMATERIAL_H

#include "Material.h"

//Forward Declarations
class ExampleMaterial;

template<>
InputParameters validParams<ExampleMaterial>();

/** 
 * Example material class that defines a few properties.
 */
class ExampleMaterial : public Material
{
public:
    ExampleMaterial(const std::string & name,
                   InputParameters parameters);

protected:
    virtual void computeQpProperties();
    virtual void initQpStatefulProperties();

private:
    Real _initial_diffusivity;

    /**
     * Create two MooseArray Refs to hold the current
     * and previous material properties respectively
     */
    MaterialProperty<Real> & _diffusivity;
    MaterialProperty<Real> & _diffusivity_old;
};

#endif //EXAMPLEMATERIAL_H
```

```
#include "ExampleMaterial.h"

template<>
InputParameters validParams<ExampleMaterial>()
{
    InputParameters params = validParams<Material>();
    params.addParam<Real>("initial_diffusivity", 1.0, "Doc");
    return params;
}

ExampleMaterial::ExampleMaterial(const std::string & name,
                                 InputParameters parameters):
    Material(name, parameters),
    // Get a parameter value for the diffusivity
    _initial_diffusivity(getParam<Real>("initial_diffusivity")),

    // Declare that this material is going to have a Real
    // valued property named "diffusivity" that Kernels can use.
    _diffusivity(declareProperty<Real>("diffusivity")),

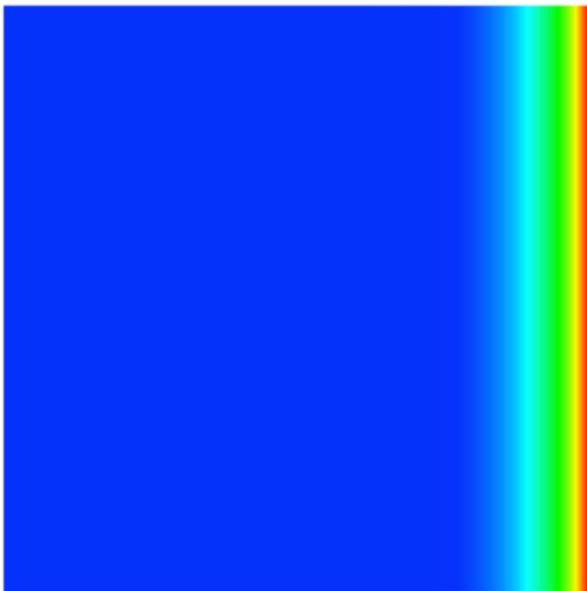
    // Declare that we are going to have an old value of diffusivity
    // Note: this is _expensive_. Only do this if you REALLY need it!
    _diffusivity_old(declarePropertyOld<Real>("diffusivity"))

void
ExampleMaterial::initQpStatefulProperties()
{
    // init the diffusivity property (this will become
    // _diffusivity_old in the first call of computeProperties)
    _diffusivity[_qp] = _initial_diffusivity;
}

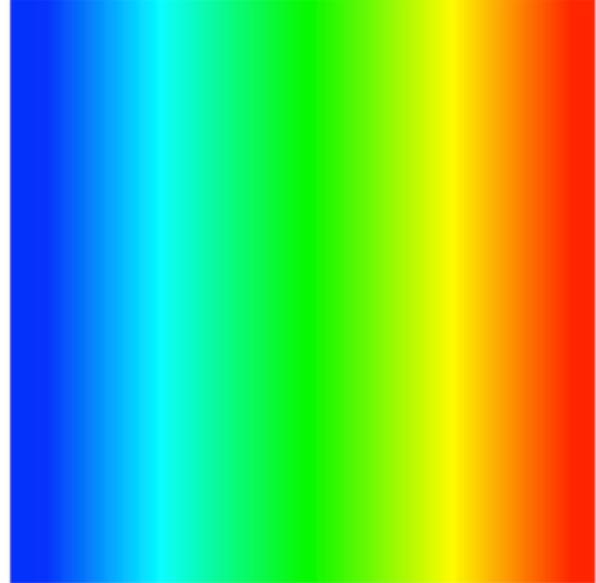
void
ExampleMaterial::computeQpProperties()
{
    _diffusivity[_qp] = _diffusivity_old[_qp] + 2;
}
```

Look at Example 9 (page E69)

Beginning



End



Peacock: The MOOSE GUI



www.inl.gov

Input File Syntax

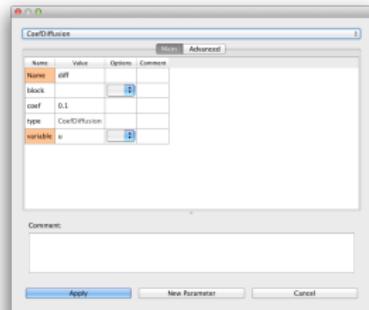
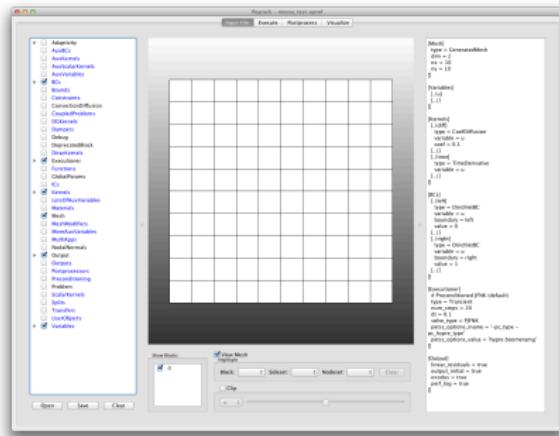
- As MOOSE grows, so does the amount of input file syntax.
- Remembering it all is definitely out of the question.
- Capabilities do exist to help you out:
 - Any MOOSE-based application can be run using `./app-opt --dump` to dump out the input file syntax.
 - You can pass another optional argument to `--dump` and the syntax will be searched for that string.
 - On the Wiki an up to date set of searchable syntax is automatically rebuilt for every application we know about.
- Each of those solutions requires that you go out of your way to find out about a piece of input file syntax.

Peacock

- Peacock is an integrated Graphical User Interface (GUI) for building input files and interacting with a MOOSE-based application.
- It is built using PyQt and will look like a native application on Mac, Linux (and even Windows!).
- It is located under `moose/gui/`.
- We recommend you add that path to your PATH environment variable.
- Peacock will automatically mold itself to the MOOSE-based application you are trying to use.
- It will intelligently try to pick up the executable you are trying to use, but it can also be specified with `-e`.
- Peacock caches the input file syntax for the application the first time you run it.
- To re-cache use (in case the syntax has changed for the app) use `-r`.

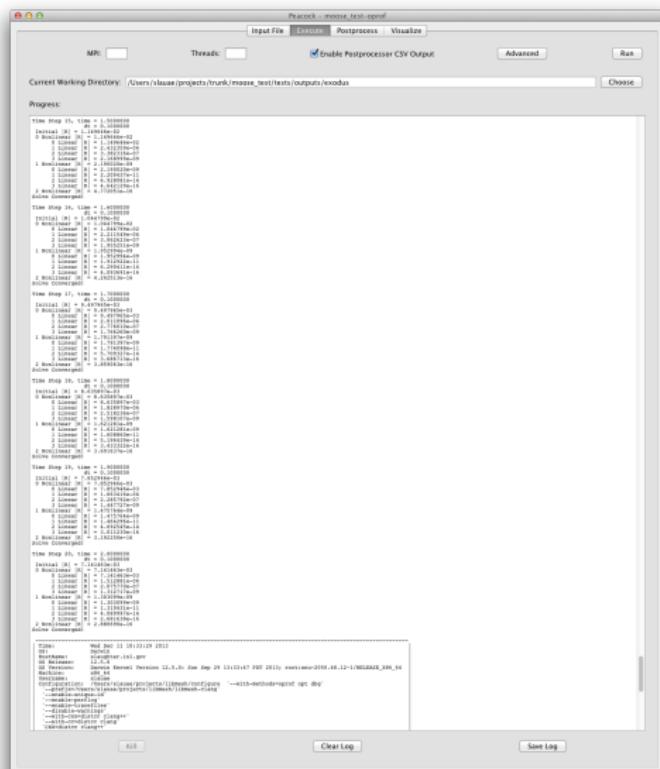
Input File Editor

- The main “tab” is the input file editor.
- You can Create/Open/Edit/Save any MOOSE-based application’s input file.
- Add a “subsection” (for instance, a Kernel) by double-clicking the main section to open the Parameter editor.
- To edit an existing subsection double-click it.
- Main blocks with parameters use a “ParentParams” subsection holding the parameters for that block.
- Check boxes indicate “active” status.
- The box on the right updates to show the changing input file.



Execution

- The second tab allows you to execute the input file.
- You don't need to save the input file.
- MPI procs and threads are easily set.
- Other command-line options can be passed.
- The output of the run is captured and can be saved.
- A progress bar will appear during the run.
- The “Kill” button will stop the currently running job.



Auxiliary Variables



www.inl.gov

Aux Variables

- The auxiliary system's purpose is to allow explicit calculations using nonlinear variables.
- These values can be used by kernels, BCs and material properties.
 - Just couple to them as if they were a nonlinear variable.
- They will also come out in the output file... useful for viewing things you don't solve for (e.g. velocity).
- Auxiliary variables currently come in two flavors:
 - Element (constant monomials)
 - Nodal (linear Lagrange)
- When using element auxiliary variables:
 - You are computing average values per element.
 - You can couple to nonlinear variables and both element and nodal auxiliary variables.
- When using nodal auxiliary variables:
 - You are computing values at nodes.
 - You can **only** couple to nonlinear variables and other nodal auxiliary variables.
- Auxiliary variables have “old” states just like nonlinear variables.

Aux Kernels

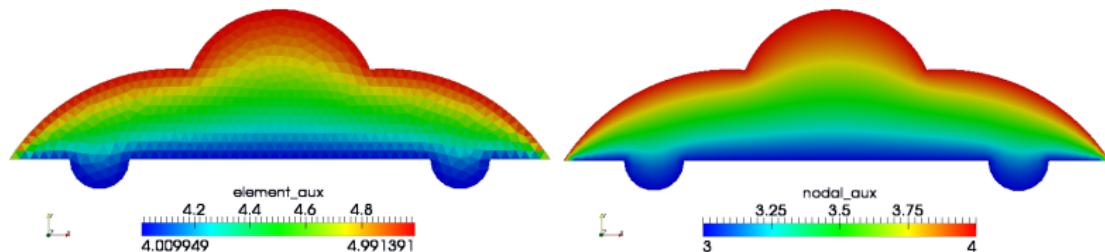
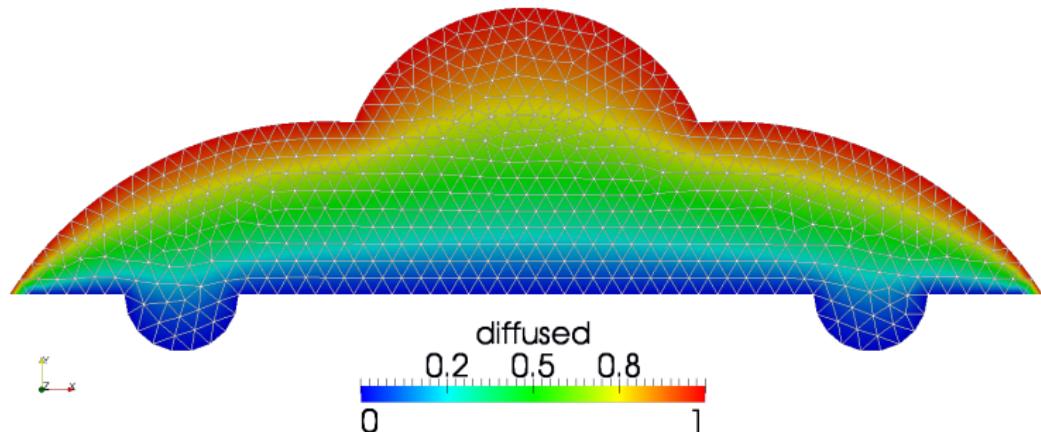
- Aux kernels should go under `include/auxkernels` and `src/auxkernels`.
- They are similar to regular kernels except that they override `computeValue()` instead of `computeQpResidual()`.
- They don't have Jacobians.
- Note that there is no physical difference between a nodal auxiliary kernel and an elemental.
- The difference is only in the input file.

(Some) Values Available to AuxKernels

- `_u`, `_grad_u`
 - Value and gradient of variable this AuxKernel is operating on.
- `_q_point`
 - XYZ coordinates of the current q-point.
 - Only valid for element AuxKernels!
- `_qp`
 - Current quadrature point.
 - Used even for nodal AuxKernels! (Just for consistency)
- `_current_elem`
 - A pointer to the current element that is being operated on.
 - Only valid for element AuxKernels!
- `_current_node`
 - A pointer to the current node that is being operated on.
 - Only valid for nodal AuxKernels!
- And more!

Go Look at Example 10 (page E75)

Example 10 Output



Preconditioning



www.inl.gov

Preconditioning

- Krylov methods need preconditioning to be efficient (or even effective!).
- Even though the Jacobian is never formed, JFNK methods still require preconditioning.
- MOOSE's automatic (without user intervention) preconditioning is fairly minimal.
- Many options exist for implementing improved preconditioning in MOOSE.

Preconditioned JFNK

- Using right preconditioning, solve

$$\mathbf{R}'(\mathbf{u}_i) \mathbf{M}^{-1} (\mathbf{M} \delta \mathbf{u}_{i+1}) = -\mathbf{R}(\mathbf{u}_i)$$

- \mathbf{M} symbolically represents the preconditioning matrix or process
- Inside GMRES, we only apply the action of \mathbf{M}^{-1} on a vector
- Right preconditioned matrix free version

$$\mathbf{R}'(\mathbf{u}_i) \mathbf{M}^{-1} \mathbf{v} \approx \frac{\mathbf{R}(\mathbf{u}_i + \epsilon \mathbf{M}^{-1} \mathbf{v}) - \mathbf{R}(\mathbf{u}_i)}{\epsilon}$$

Preconditioning Matrix vs Process

- On the previous slide \mathbf{M} represented the “Preconditioning Matrix”.
- The action of \mathbf{M}^{-1} on a vector represents the “Preconditioner” or “Preconditioning Process”.
- In MOOSE the “matrix to build” and the “process to apply” with that matrix are separated.
- There are four different ways to build preconditioning matrices:
 - Default: Block Diagonal Preconditioning
 - Single Matrix Preconditioner (SMP)
 - Finite Difference Preconditioner (FDP)
 - Physics Based Preconditioner (PBP)
- After selecting how to build a preconditioning matrix you can then use solver options to select how to apply the Preconditioner.

Solve Type

- The default `solve_type` for MOOSE is “Preconditioned JFNK”.
- An alternative `solve_type` can be set through either the `[Executioner]` or `[Preconditioner/*]` block.
- Valid options include:
 - PJFNK (default)
 - JFNK
 - NEWTON
 - FD (Finite Difference)

PETSc Preconditioning Options

- For specifying the preconditioning process we use solver options directly (i.e. PETSc options).
- Currently the options for preconditioning with PETSc are exposed to the applications.
- This will change in the future... there will be more generic ways of specifying preconditioning parameters.
- The best place to learn about all of the preconditioning options with PETSc is the user manual.
- We use the command-line syntax, but provide places to enter it into the input file.

<http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf>

PETSc Specific Options (for Executioner)

<code>petsc_options</code>	Description
<code>-snes_ksp_ew</code>	Variable linear solve tolerance – useful for transient solves
<code>-help</code>	Show PETSc options during the solve

<code>petsc_options_iname</code>	<code>petsc_options_value</code>	Description
<code>-pc_type</code>	<code>ilu</code>	Default for serial
	<code>bjacobi</code>	Default for parallel with <code>-sub_pc_type ilu</code>
	<code>asm</code>	Additive Schwartz with <code>-sub_pc_type ilu</code>
	<code>lu</code>	Full LU, serial only!
	<code>gamg</code>	PETSc Geometric AMG Preconditioner
	<code>hypre</code>	Hypre, usually used with <code>boomeramg</code>
<code>-sub_pc_type</code>	<code>ilu, lu, hypre</code>	Can be used with <code>bjacobi</code> or <code>asm</code>
<code>-pc_hypre_type</code>	<code>boomeramg</code>	Algebraic multigrid
<code>-pc_hypre_boomeramg (cont.)</code>		"Information Threshold" for AMG process
<code>_strong_threshold</code>	<code>0.0 - 1.0</code>	(Use 0.7 for 3D!)
<code>-ksp_gmres_restart</code>	<code>#</code>	Number of Krylov vectors to store

Default Preconditioning Matrix

- Consider the fully coupled system of equations:

$$\begin{aligned}-\nabla \cdot k(s, T) \nabla T &= 0 \\ -\nabla \cdot D(s, T) \nabla s &= 0\end{aligned}$$

- Fully coupled Jacobian approximation

$$\mathbf{R}'(s, T) = \begin{bmatrix} (\mathbf{R}_T)_T & (\mathbf{R}_T)_s \\ (\mathbf{R}_s)_T & (\mathbf{R}_s)_s \end{bmatrix} \approx \begin{bmatrix} (\mathbf{R}_T)_T & \mathbf{0} \\ \mathbf{0} & (\mathbf{R}_s)_s \end{bmatrix}$$

- For our example:

$$\mathbf{M} \equiv \begin{bmatrix} (k(s, T) \nabla \phi_j, \nabla \psi_i) & \mathbf{0} \\ \mathbf{0} & (D(s, T) \nabla \phi_j, \nabla \psi_i) \end{bmatrix} \approx \mathbf{R}'(s, T)$$

- This simple style of throwing away the off-diagonal blocks is the way MOOSE will precondition when using the default `solve_type`.

The Preconditioning Block

```
[Preconditioning]
active = 'my_prec'

[./my_prec]
type = SMP
# SMP Options Go Here!
# Override PETSc Options Here!
[../]

[./other_prec]
type = PBP
# PBP Options Go Here!
# Override PETSc Options Here!
[../]

[]
```

- The Preconditioning block allows you to define which type of preconditioning matrix to build and what process to apply.
- You can define multiple blocks with different names, allowing you to quickly switch out preconditioning options.
- Each sub-block takes a `type` parameter to specify the type of preconditioning matrix.
- Within the sub-blocks you can also provide other options specific to that type of preconditioning matrix.
- You can also override PETSc options here.
- Only one block can be active at a time.

Single Matrix Preconditioning (SMP)

- The Single Matrix Preconditioner (SMP) builds one matrix for preconditioning.
- You enable SMP with:

```
type = SMP
```

- You specify which blocks of the matrix to use with:

```
off_diag_row      = 's'  
off_diag_column = 'T'
```

- Which would produce an \mathbf{M} like this:

$$\mathbf{M} \equiv \begin{bmatrix} (k(s, T) \nabla \phi_j, \nabla \psi_i) & \mathbf{0} \\ \left(\frac{\partial D(s, T)}{\partial T_j} \nabla s, \nabla \psi_i \right) & (D(s, T) \nabla \phi_j, \nabla \psi_i) \end{bmatrix} \approx \mathbf{R}'$$

- In order for this to work you must provide a `computeQpOffDiagJacobian()` function in your Kernels that computes the required partial derivatives.
- To use *all* off diagonal blocks, you can use the following input file syntax:

```
full = true
```

Finite Difference Preconditioning (FDP)

- The Finite Difference Preconditioner (FDP) allows you to form a “Numerical Jacobian” by doing direct finite differences of your residual statements.
- This is extremely slow and inefficient, but is a great debugging tool because it allows you to form a nearly perfect preconditioner.
- You specify it by using:

```
type = FDP
```

- You can use the same options for specifying off-diagonal blocks as SMP.
- Since FDP allows you to build the perfect approximate Jacobian it can be useful to use it directly to solve instead of using JFNK.
- The finite differencing is sensitive to the differencing parameter which can be specified using:

```
petsc_options_iname = '-mat_fd_coloring_err -mat_fd_type'  
petsc_options_value = '1e-6'                                ds'
```

- NOTE: FDP currently works in serial only! This might change in the future, but FDP will always be meant for debugging purposes!

Examples

```
[Executioner]
...
petsc_options_iname = '-pc_type -pc_hypre_type -ksp_gmres_restart'
petsc_options_value = 'hypre      boomeramg      101'
...
[]
```

- Default Preconditioning Matrix, Preconditioned JFNK, monitor linear solver, variable linear solver tolerance.
- Use Hypre with algebraic multigrid and store 101 Krylov vectors.

```
[Preconditioning]
active = 'SMP_jfnk'

[./SMP_jfnk]
type = SMP

off_diag_row    = 'forced'
off_diag_column = 'diffused'

petsc_options_iname = '-pc_type'
petsc_options_value = 'lu'
[.../]
[]
```

- Single Matrix Preconditioner, Fill in the (forced, diffused) block, Preconditioned JFNK, Full inverse with LU

Look at Example 11 (page E81)

Physics Based Preconditioning

- Physics based preconditioning is an advanced concept used to more efficiently solve using JFNK.
- The idea is to create a preconditioning process that targets each physics individually.
- In this way you can create a more effective preconditioner... while also maintaining efficiency.
- In MOOSE there is a `PhysicsBasedPreconditioner` object.
- This object allows you to dial up a preconditioning matrix and the operations to be done on the different blocks of that matrix on the fly from the input file.

What the PBP Does

- The PBP works by partially inverting a preconditioning matrix (usually an approximation of the true Jacobian) by partially inverting each block row in a Block-Gauss-Seidel way.

$$\mathbf{R}(u, v) = \begin{bmatrix} \mathbf{R}_u \\ \mathbf{R}_v \end{bmatrix}$$

$$\mathbf{M} \equiv \begin{bmatrix} (\mathbf{R}_u)_u & \mathbf{0} \\ (\mathbf{R}_v)_u & (\mathbf{R}_v)_v \end{bmatrix} \approx \mathbf{R}'$$

$$\mathbf{M}\mathbf{q} = \mathbf{p} \quad \Rightarrow \quad \begin{cases} (\mathbf{R}_u)_u \mathbf{q}_u &= \mathbf{p}_u \\ (\mathbf{R}_v)_v \mathbf{q}_v &= \mathbf{p}_v - (\mathbf{R}_v)_u \mathbf{q}_u \end{cases}$$

Using the PBP

```
[Variables]
...
[]

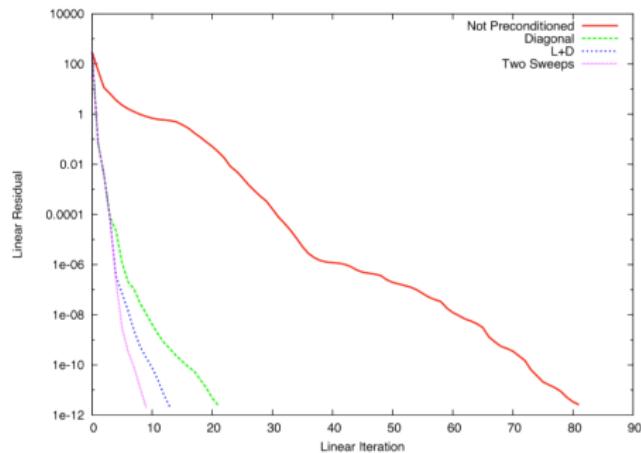
[Preconditioning]
active = 'myPBP'

[./myPBP]
type = PBP
solve_order = 'u v'
preconditioner = 'ILU AMG'
off_diag_row = 'v'
off_diag_column = 'u'
[...]
[]
```

- Set up a PBP object for a two variable system (consisting of variables “u” and “v”).
- Use ILU for the “u” block and AMG for “v”.
- Use the lower diagonal (v,u) block.
- When using ‘type=PBP’, MOOSE will set solve_type = JFNK automatically.

Applying PBP

- Applying these ideas to a coupled thermo-mechanics problem:



Look at Example 12 (page E91)

Functions



www.inl.gov

Functions

- Function objects allow you to evaluate analytic expressions based on x, y, z and time.
- You can create your own custom Function object in the general MOOSE way of inheriting from Function and overriding `value()` and optionally `gradient()`.
- Functions can be used in any MOOSE-based object by calling `getFunction("name")` where "name" matches a name from the input file.
- In this way you can create generic BCs, ICs, Forcing Functions, Materials and more where you can plug and play different Functions to change behavior.
- A number of these have already been created that you can use including:
 - `FunctionDirichletBC`, `FunctionNeumannBC`, `FunctionIC`, `UserForcingFunction`...
- Each of these objects takes a "function" parameter from the input file to know what Function object to use.
- In addition to creating your own objects that inherit from Function there is also a `ParsedFunction` object. This object will parse arbitrary expressions directly from the input file to define its value and gradient like:
 - `value = x*x+sin(y*t)`

Default Functions

- Whenever a Function object is requested through an InputParameter object, a default can be provided.
- Constant values or parsed functions can be supplied as the default function.

```
...
// Adding a Function with a default constant
params.addParam<FunctionName>("pressure_grad", "0.5", "doc");

// Adding a Function with a default parsed function
params.addParam<FunctionName>("power_history", "t+100*sin(y)", "doc");
...
```

- A ParsedFunction or ConstantFunction object is automatically constructed based on the default value if a function name is not supplied in the input file.

Input File Syntax

```
...
[Functions]
active = 'bc_func'

[./bc_func]
type = ParsedFunction
value = 'sin(alpha*pi*x)'
vars = 'alpha'
vals = '16'
[../]
[]
...
[BCs]
active = 'all'

[./all]
type = FunctionDirichletBC
variable = u
boundary = '1 2'
function = bc_func
[../]
[]
```

- Functions are declared in the Function block.
- ParsedFunction allows you to provide a string specifying the function.
- You can use constants (like alpha), and define their value. Common expressions like sin() and pi are built in.
- After you have declared your functions you can use them in objects like FunctionDirichletBC.

Look at Example 13 (page E97)

Postprocessors and Code Verification

www.inl.gov



Postprocessors

- A Postprocessor is an explicit calculation based on the values of your simulation variables.
- They are computed when needed using the `execute_on` option in the input file.
 - `execute_on = timestep`
 - `execute_on = initial`
 - `execute_on = jacobian`
 - `execute_on = residual`
 - `execute_on = timestep_begin`
- They can be restricted to specific blocks, sidesets, and nodesets in your domain.
- The output from a Postprocessor is **one** scalar number.
- Think of Postprocessors as “Reductions” or “Aggregations”.

Types of Postprocessors

- Element
 - Operate on each element.
 - Can be restricted to subdomains by specifying one or more `block` ids.
 - Inherit from `ElementPostprocessor`.
- Nodal
 - Operate on each node.
 - Can be restricted to nodesets by specifying one or more `boundary` ids.
 - Inherit from `NodalPostprocessor`.
- Side
 - Operate on boundaries.
 - **Requires** specification of one or more `boundary` ids.
 - Inherit from `SidePostprocessor`.
- General
 - Does whatever it wants.
 - Inherit from `GeneralPostprocessor`.

Postprocessor Anatomy

Postprocessor virtual functions for implementing your aggregation operation:

- `void initialize()`
 - Clear or initialize your data structures before execution.
- `void execute()`
 - Called on each geometric entity for the type of this Postprocessor.
- `void threadJoin(const UserObject & uo)`
 - Aggregation across threads.
 - Called to join the passed in Postprocessor **with this** Postprocessor.
 - You have local access to the data structures in both Postprocessors.
- `void finalize()`
 - Aggregation across MPI.
 - One of the only places in MOOSE where you might need to use MPI!
 - Several Aggregation routines are available in libMesh's `parallel.h` file.
- `Real getValue()`
 - Retrieve the final scalar value.

Helpful Aggregation routines

We provide several utility routines to perform common aggregation operations:

- MOOSE convenience functions:
 - `gatherSum(scalar)` – returns the sum of `scalar` across all processors.
 - `gatherMin(scalar)` – returns the min of `scalar` from all processors.
 - `gatherMax(scalar)` – returns the max of `scalar` from all processors.
 - `gatherProxyValueMax(scalar, proxy)` – returns `proxy` based on max `scalar`.
- LibMesh convenience functions (from `parallel.h`):
 - `_communicator.max(...)`
 - `_communicator.sum(...)`
 - `_communicator.min(...)`
 - `_communicator.gather(...)`
 - `_communicator.send(...)`
 - `_communicator.receive(...)`
 - `_communicator.set_union(...)`
- LibMesh functions work with a wide variety of types (scalars, vectors, sets, maps, ...)

ThreadJoin (Advanced)

- You do not need to implement this function initially to run in parallel. Start with `finalize()` and use MPI only.
- You generally need to cast the base class reference to the current type so that you can access the data structures within.
- Finally, you can perform your custom aggregation operation.

```
void
PPSum::threadJoin(const UserObject & y)
{
    // Cast UserObject into a PPSum object so that we can access member variables
    const PPSum & pps = static_cast<const PPSum &>(y);

    _total_value += pps._total_value;
}
```

Postprocessors

- A few types of built in Postprocessors:
 - ElementIntegral, ElementAverageValue
 - SideIntegral, SideAverageValue
 - ElementL2Error, ElementH1Error
 - And others...
- In all of these cases you can inherit from these objects and extend them. For instance, if you want the average flux on one side you can inherit from SideAverageValue and override computeQpIntegral to compute the flux at every quadrature point.
- For the Element and Side Postprocessors, you can use material properties (and functions).
- By default, Postprocessors will output to a formatted table on the screen, but they can also write to a CSV or Tecplot file.
- They are also written to Exodus format when using that as an output option.

Default Postprocessor Values

- It is possible to set default values for Postprocessors.
- This allows a MooseObject (e.g., Kernel) to operate without creating or specifying a Postprocessor.
- Within the `validParams` function for your object, declare a Postprocessor parameter with a default value.

```
params.addParam<PostprocessorName>("postprocessor", 1.2345, "My optional postprocessor documentation")
```

- When you use the `getPostprocessorValue()` interface, MOOSE provides the user-defined value, or the default if no PostProcessor has been specified.

```
const PostprocessorValue & value = getPostprocessorValue("postprocessor");
```

- Additionally, users may supply a real value in the input file in lieu of a postprocessor name

Input File Syntax and Output

- Postprocessors are declared in the Postprocessors block.
- The name of the sub-block (like side_average and integral) is the “name” of the Postprocessor, and will be the name of the column in the output.
- Element and Side Postprocessors generally take a variable argument to work on, but can also be coupled to other variables just like Kernels, BCs, etc.

```
[Postprocessors]
[./dofs]
  type = NumDOFs
[]

[./h1_error]
  type = ElementH1Error
  variable = forced
  function = bc_func
[]

[./l2_error]
  type = ElementL2Error
  variable = forced
  function = bc_func
[]
[]
```

Postprocessor Values:

time	dofs	h1_error	l2_error
0.000000e+00	9.000000e+00	2.224213e+01	9.341963e-01
1.000000e+00	9.000000e+00	2.224213e+01	9.341963e-01
2.000000e+00	2.500000e+01	6.351338e+00	1.941240e+00
3.000000e+00	8.100000e+01	1.983280e+01	1.232381e+00
4.000000e+00	2.890000e+02	7.790486e+00	2.693545e-01
5.000000e+00	1.089000e+03	3.995459e+00	7.130219e-02
6.000000e+00	4.225000e+03	2.010394e+00	1.808616e-02
7.000000e+00	1.664100e+04	1.006783e+00	4.538021e-03

Code Verification Using MMS

- Method of Manufactured Solutions (MMS) is a useful tool for code verification (making sure that your mathematical model is being properly solved).
- MMS works by assuming a solution, substituting it into the PDE, and obtaining a “forcing term”.
- The modified PDE (with forcing term added) is then solved numerically; the result can be compared to the assumed solution.
- By checking the norm of the error on successively finer grids you can verify your code obtains the theoretical convergence rate (i.e. that you don't have any code bugs).
- For example:

$$\text{PDE:} \quad -\nabla \cdot \nabla u = 0$$

$$\text{Assumed solution:} \quad u = \sin(\alpha\pi x)$$

$$\text{Forcing function:} \quad f = \alpha^2\pi^2 \sin(\alpha\pi x)$$

$$\text{Need to solve:} \quad -\nabla \cdot \nabla u - f = 0$$

Error Analysis

- To compare two solutions (or a solution and an analytical solution) f_1 and f_2 , the following expressions are frequently used:

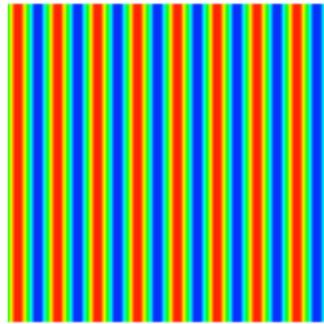
$$\|f_1 - f_2\|_{L_2(\Omega)}^2 = \int_{\Omega} (f_1 - f_2)^2 \, d\Omega$$

$$\|f_1 - f_2\|_{H_1,\text{semi}}^2 = \int_{\Omega} |\nabla(f_1 - f_2)|^2 \, d\Omega$$

- From finite element theory, we know the convergence rates of these quantities on successively refined grids.
- They can be computed in a MOOSE-based application by utilizing the `ElementL2Error` or `ElementH1SemiError` Postprocessors, respectively, and specifying the analytical solution with Functions.

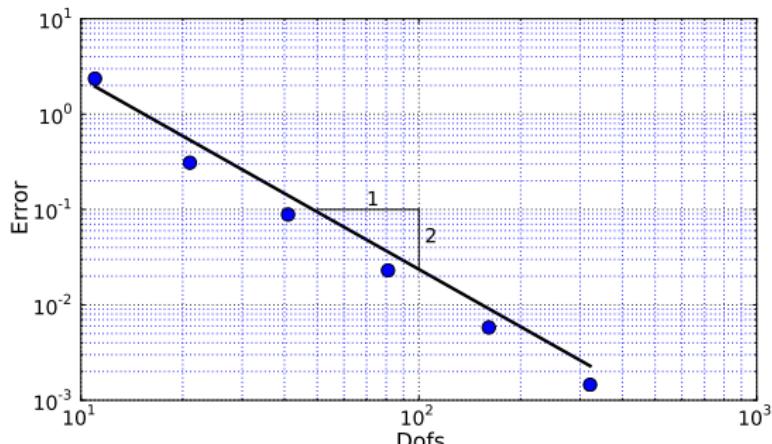
Look at Example 14 (page E103)

Example 14 Output



Postprocessor Values:

time	dofs	integral
0.000000e+00	1.210000e+02	7.071068e-01
1.000000e+00	1.210000e+02	2.359249e+00
2.000000e+00	4.410000e+02	3.093980e-01
3.000000e+00	1.681000e+03	8.861951e-02
4.000000e+00	6.561000e+03	2.297902e-02
5.000000e+00	2.592100e+04	5.797875e-03
6.000000e+00	1.030410e+05	1.452813e-03



Parallel-Agnostic Random Number Generation



www.inl.gov

Pseudo-Random Number Generation (PRNG)

- Most MOOSE objects include an interface for generating pseudo-random numbers consistently during serial, parallel, and threaded runs.
- This consistency enables more robust development and debugging without sacrificing PRNG quality.
- Users have control over the reset frequency of the PRNG.
- Helps avoid convergence issues due to excessive PRNG noise during linear or non-linear iterations.
- The PRNG system avoids the repetition of “patterns” in subsequent executions of an application.

Using Random Numbers

- Make a call to `setRandomResetFrequency()` in your object's constructor.

```
// Options include EXEC_RESIDUAL, EXEC_JACOBIAN, EXEC_TIMESTEP, EXEC_INITIAL  
// Note: EXEC_TIMESTEP == EXEC_TIMESTEP_BEGIN for the purpose of reset  
  
setRandomResetFrequency(EXEC_RESIDUAL);
```

- Obtain Random Numbers (Real or Long)

```
// anywhere inside your object (except the constructor)  
  
unsigned long random_long = getRandomLong();  
Real random_real = getRandomReal();
```

More Details

- Each MooseObject has its own “seed” value.
- The seed is used to generate different random sequences from run to run.
- The “reset frequency” specifies how often the random number generators should be reset.
- If you reset on `EXEC_RESIDUAL`, you will get the same random number sequence each residual evaluation for a given timestep.
- You can also reset less often, e.g. Jacobian, timestep, or simulation initialization only.
- Generators are advanced every time step unless you explicitly set the reset frequency to `EXEC_INITIAL`.
- A multi-level random seeding scheme is used to avoid patterning from mesh entity to mesh entity, timestep to timestep, and run to run.

MooseApp and main()



www.inl.gov

MooseApp Base Class

- A MOOSE Application (`MooseApp`) is just another plugable system.
- Your application will contain a `MooseApp` derived class.
- Each `MooseApp` is responsible for holding on to a number of essential objects such as the Factories where its objects are built, and the Warehouses where its objects are stored.
- In this way, `MooseApp` objects can be combined (coupled) together to form larger simulations. (See the MultiApp System)
- `MooseApp` objects also supply `validParams()` like any other MOOSE object.
 - Parameters for `MooseApps` are extracted from the Command Line with special functions on `InputParameters`.
 - `InputParameters::addRequiredCommandLineParam()`
 - `InputParameters::addCommandLineParam()`

Your AnimalApp.h/C

- Your application will contain a MooseApp derived class which will contain a couple of key static functions:
 - registerApps()
 - registerObjects()
 - associateSyntax() // optional
- This is the “go-to” place to see all of the applications and objects that this application depends on.

Object Registration

- The Application object is responsible for calling the static function `registerObjects()` where you have registered all of the objects created in your application.
- Strong coupling to other applications is handled by calling `registerObjects()` in those applications making those objects available in your application.
- By default, your application will register all MOOSE and MOOSE module objects, but can be made to link to any other application.
- When creating a new application with `Stork`, your application object will be completely setup and should not require any editing for normal use.

main.C

- This is your program entry point!
- By default it only does a couple of very specific things:
 - Initializes MPI through an object Wrapper (`MooseInit`)
 - Calls `registerApps()` which makes your application objects available for construction in the `AppFactory`.
 - Builds your application object
 - Runs your application object
 - Clean-up and exit
- See main.C in Example 1 (page E5)
- While it is possible to add additional code here, it's not recommended.

Testing



www.inl.gov

Test Harness

- MOOSE provides an extendable Test Harness for executing your code with different input files.
- Each kernel (or logical set of kernels) you write should have test cases that verify the code's correctness.
- The test system is very flexible. You can perform many different operations, for example: testing for expected error conditions.
- Additionally, you can create your own “Tester” classes which extend the Test Harness.

Tests setup

- Related tests should be grouped into an individual directory and have a consistent naming convention.
- We recommend organizing tests in a hierarchy similar to your application source (i.e. kernels, BCs, materials, etc).
- Tests are found dynamically by matching patterns (highlighted below)

```
tests/
  kernels/
    my_kernel_test/
      my_kernel_test.e      [input mesh]
      my_kernel_test.i      [input file]
      tests                 [test specification file]
      gold/                 [gold standard folder - validated solution]
      out.e                 [solution]
```

A quick look at the test specification file

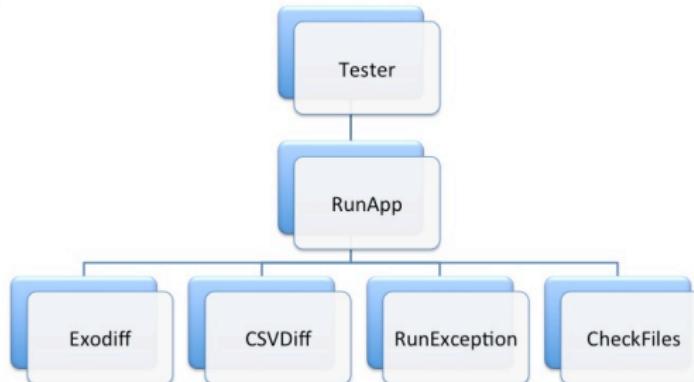
- Same format as the standard MOOSE input file

```
[Tests]
[./my_kernel_test]
  type      = Exodiff
  input     = my_kernel_test.i
  exodiff  = my_kernel_test_out.e
[.../]

[./kernel_check_exception]
  type      = RunException
  input     = my_kernel_exception.i
  expect_err = 'Bad stuff happened with variable \w+'
[.../]

[]
```

Testers provided in MOOSE



- RunApp: Runs a MOOSE-based application with specified options.
- Exodiff: Checks Exodus files for differences within specified tolerances.
- CSVDiff: Checks CSV files for differences within specified tolerances.
- RunException: Tests for various error conditions.
- CheckFiles: Checks for the existence of specific files after a completed run.

Adding Additional Testers (Advanced)

- Inherit from Tester and override:
 - checkRunnable()
 - prepare(): Method to run right before a test starts
 - getCommand(): Command to run (in parallel with other tests)
 - processResults(): Process the results to check whether the test has passed
- NO REGISTRATION REQUIRED!
 - Drop the Tester object in “<Your App>/scripts/TestHarness/testers”

Options available to each Tester

Run: `./run_tests --dump`

- `input`: The name of the input file
- `exodiff`: The list of output filenames to compare
- `abs_zero`: Absolute zero tolerance for exodiff
- `rel_err`: Relative error tolerance for exodiff
- `prereq`: Name of the test that needs to complete before running this test
- `min_parallel`: Minimum number of processors to use for a test (default: 1)
- `max_parallel`: Maximum number of processors to use for a test
- ...

Running your tests

```
./run_tests [options]
-j <n>                      run 'n' jobs at a time
--dbg                         run tests in debug mode (debug binary)
FOLDER NAME                   run just one set of tests

-h                            help
--heavy                       run regular tests and those marked 'heavy'
-a                            write separate log file for each failed test
--group=GROUP                 all the tests in a user defined group
--not_group=GROUP              opposite of --group option
-q                            quiet (don't print output of FAILED tests
-p <n>                        request to run each test with 'n' procs
```

Other Notes on Tests

- Individual tests should run relatively quickly (~2 second rule)
- Outputs or other generated files should not be checked into the subversion repository.
 - Do not check in the solution that is created in the test directory when running the test!
- The MOOSE developers rely on application tests when refactoring to verify correctness
 - Poor test coverage = Higher code failure rate

MOOSE Modules



www.inl.gov

MOOSE Modules

- MOOSE comes with a library of community-developed physics modules.
- The purpose of the modules is to encapsulate common kernels, boundary conditions, etc. to prevent code duplication.
- Examples include: heat conduction, solid mechanics, Navier-Stokes, and others.
- *No* export controlled physics (i.e. neutronics) should be put into the MOOSE modules.
- The modules are organized so that your application can link against only those which it requires.

MOOSE Modules Anatomy

```
$ ls moose/modules
chemical_reactions
combined
contact
fluid_mass_energy_balance
heat_conduction
linear_elasticity
navier_stokes
phase_field
richards
solid_mechanics
tensor_mechanics
water_steam_eos
```

- All applications are set up to use the MOOSE modules.
- If you need assistance adding or removing unneeded modules from your application, please contact us and we'll be happy to help.

MOOSE Modules Anatomy

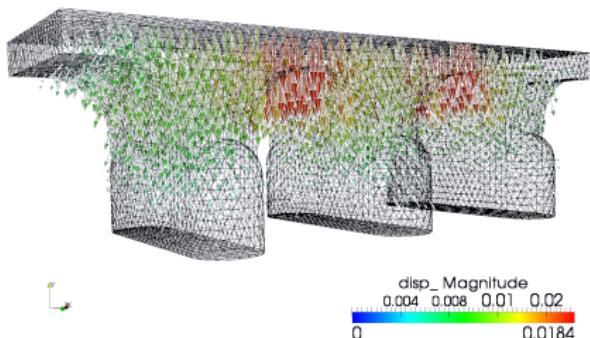
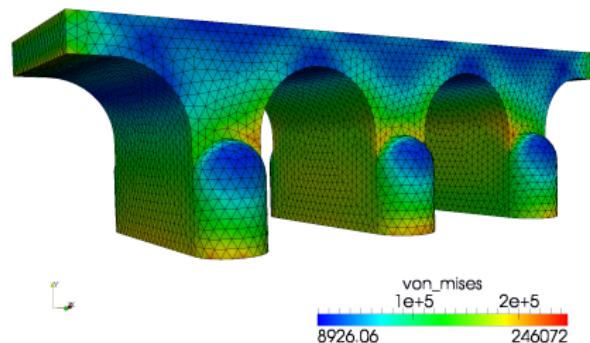
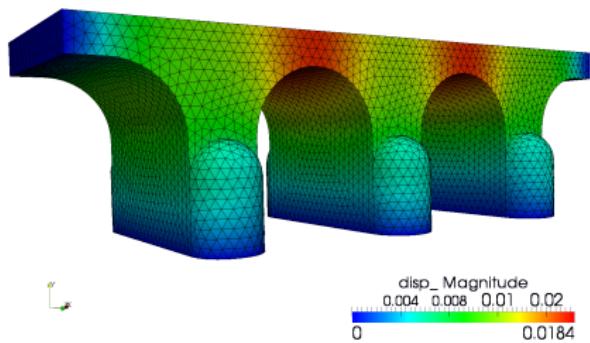
- The contents of each module are the same as any MOOSE application:

```
$ ls moose/modules/solid_mechanics
Makefile
doc
include
lib
plugins
run_tests
src
tests
```

- Application codes specify the modules they want to use in their Makefiles with the following syntax:

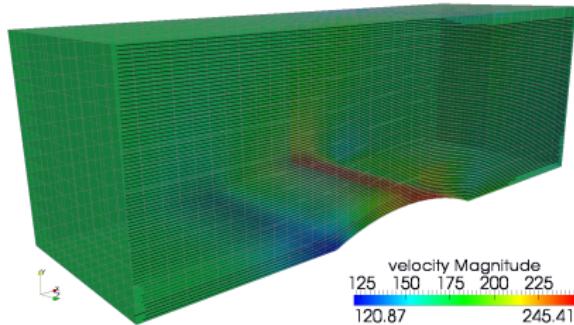
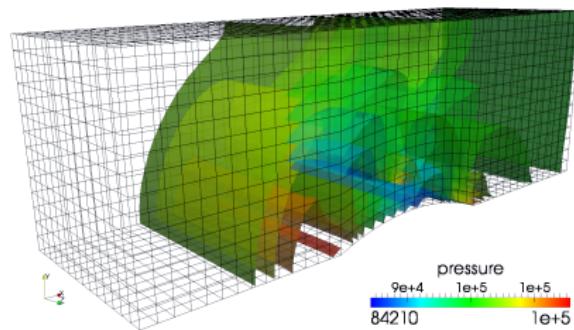
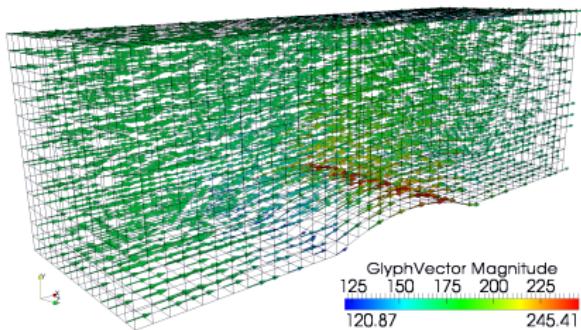
```
#####
# MODULES #####
#####
SOLID_MECHANICS    := yes
LINEAR_ELASTICITY  := yes
```

MOOSE modules: Solid Mechanics Example



- Available in
modules/solid_mechanics
- Stats:
 - 127,650 elements, 25,227 nodes
- Features:
 - Large displacement formulation
 - Plasticity
 - Creep

MOOSE modules: Flow Example



- Available in
modules/navier_stokes
- Subsonic Test Case:
 - Mach 0.5 over a circular arc
 - Euler equations
 - 8,232 elements, 9,675 nodes

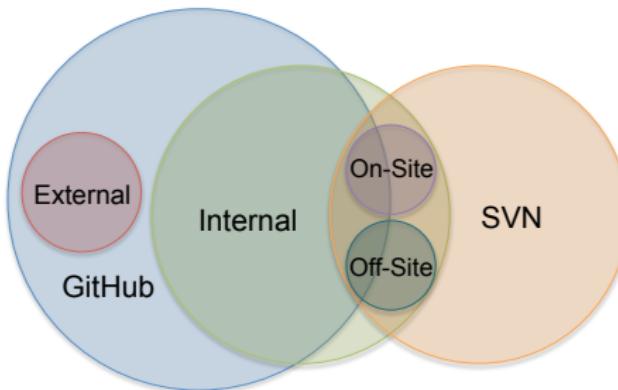
MOOSE Internal SVN Development

www.inl.gov



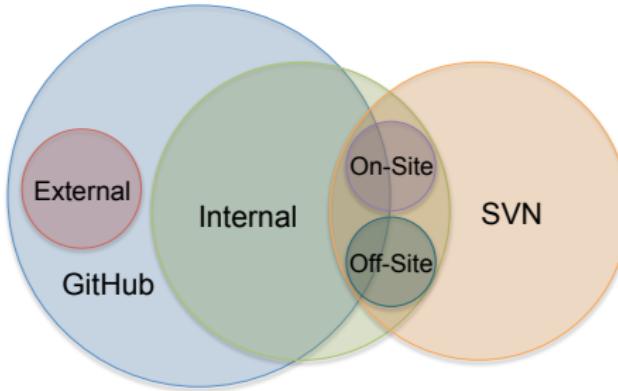
Developer Audiences

- The MOOSE team currently supports *several* distinct developer audiences:



- Internal = Developer *with* access to INL servers.
- External = Developer *without* access to INL servers.
- SVN = Developer working with an SVN checkout of INL MOOSE applications.
- GitHub = Developer working with a git checkout of MOOSE from GitHub.

Developer Audiences



- These training slides focus primarily on
 - Internal SVN users, both on- and off-site.
 - External GitHub users.
- Detailed instructions¹ for Internal and External GitHub *developers* (people who will commit to MOOSE) are also available.

¹https://mooseframework.org/static/media/uploads/docs/moose_github.pdf

www.mooseframework.org

Instructions, support, and tools for users of MOOSE.

- **Getting Starting:**

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

- **Documentation:**

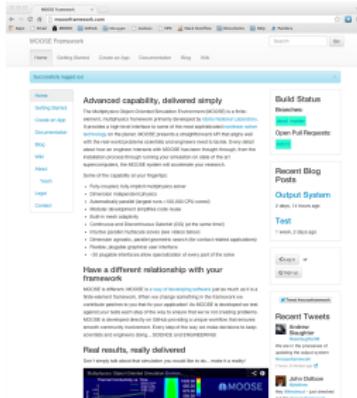
<http://mooseframework.org/documentation>

- **Wiki:**

<http://mooseframework.org/wiki>

- **Blog:**

<http://mooseframework.org/blog>



Off-site SVN Access

- Apply for an account
 - <https://secure.inl.gov/claims>
- Set up Port Forwarding
 - Linux/OS X
 - Set up a `~/.ssh/config` profile
 - Windows (Really?)
 - Configure forwarding in PuTTY or some other SSH tool
- Create an ssh connection to `hpclogin.inl.gov`, and leave it open.
- Translate internal links as follows:

<https://hpcsc/moose> → <https://localhost:4443/moose>

<https://hpcsc/svn/herd/trunk> → <https://localhost:4443/svn/herd/trunk>

Sample SSH config file for off-site users

```
Host *
    ControlMaster auto
    ControlPath ~/.ssh/master-%r@%h:%p

# General Purpose HPC Machines
Host flogin1 flogin2 eos quark hpcsc hpcweb
    User <username>
    ProxyCommand ssh <username>@hpclogin.inl.gov netcat %h %p

# Forwarding
Host hpclogin hpclogin.inl.gov
    User <username>
    HostName hpclogin.inl.gov
    LocalForward 8080 hpcweb:80
    LocalForward 4443 hpcsc:443
```

Getting Started On Your Own Machine

- On the internal Trac Wiki, there are “Getting Started” links on the front page to directions for getting your environment set up for use with MOOSE.
- In general the steps are:
 1. Download and install redistributable package.
 2. Check out MOOSE.
 3. Build libmesh.
 4. Build your application.
 5. Run tests!
- We currently have redistributable packages for Linux (Ubuntu and others) and Mac OSX.
- Under “Getting Started” there are also “advanced” directions for special situations (like compiling MOOSE on a cluster).
- No matter what directions you follow, make sure to *read every step carefully!*

Basic Subversion workflow

- The INL internal MOOSE applications are currently kept in a Subversion repository.
- Subversion (or SVN) allows multiple people to work on source code simultaneously.
- The basic workflow is:
 1. Check out source from server.
 2. Make changes.
 3. Add files.
 4. Update.
 5. TEST!
 6. Commit changes back to server.
- Create an SVN checkout by typing:

```
svn co https://hpcsc/svn/herd/trunk
```

Building

- Build libMesh by running:

```
~/projects/trunk/moose/scripts/update_and_rebuild_libmesh.sh
```

- After libMesh is compiled, compile MOOSE by typing:

```
make -j8
```

- The 8 after `-j` should be replaced with the number of processors in your machine. This enables parallel compilation.
- To build a debug version of the code, type `METHOD=dbg make -j8`
- Another option is just to go into your application and do `make -j8` this will automatically recompile your application, MOOSE, and MOOSE modules if necessary.

Running Tests

- After building your application, it can be a good idea to make sure everything is working by running the tests:

```
./run_tests
```

- If you would like to run the MOOSE test suite you can type:

```
cd ~/projects/trunk/moose/test  
make -j8  
./run_tests
```

- Remember to add tests to your test suite whenever possible. This will help you to know when something is broken, and will help the MOOSE developers know when they've broken your application!

Basic Subversion Usage

- At any time you can see what you have modified in the current directory by doing:

```
svn stat
```

- This will generally print a lot of extra files with "?" next to them. These are usually just generated files like ".o" and ".d". To get them out of the way do:

```
svn stat -q
```

- Some of those "?" files might be new files you need to tell subversion about:

```
svn add filename
```

- To see the actual modifications you have made to a file you can do:

```
svn diff filename
```

- When you are ready to commit your changes you first need to update your copy of the code:

```
svn update
```

- Note that this can fail if someone else modified the same piece of code you did. You will get a conflict that you will have to resolve. Just ask us the first time this happens.

- The next step before committing is to rerun the tests to make sure you haven't broken anything for your application:

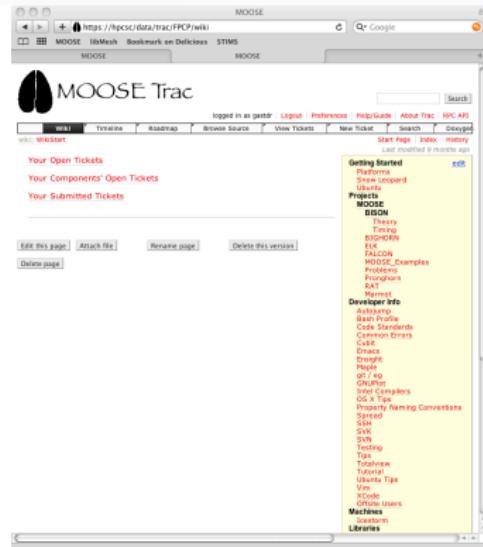
```
./run_tests
```

- Finally, when you are happy with the changes you have made and you've svn added all the new files, you can commit your changes using:

```
svn commit -m "Some descriptive message."
```

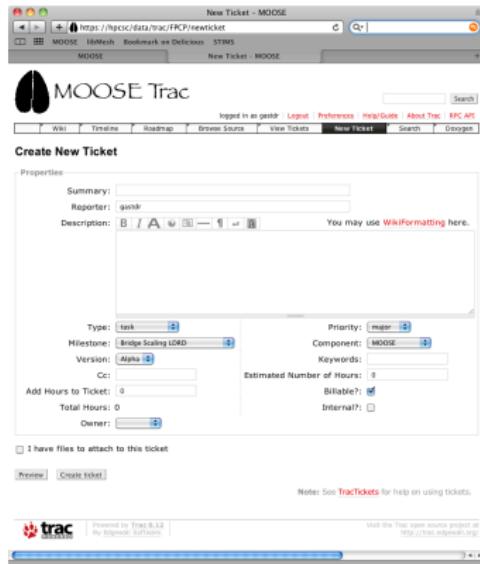
Internal Wiki

- The MOOSE Wiki is a one stop shop for information on how to develop applications using MOOSE.
- To get to the wiki just point your browser to: <https://hpcsc/moose>
- Log in using your INL HPC username and password.
- The first thing you will see is the Wiki.
- Anyone can add or edit any page on the Wiki (in fact it's encouraged!)
- In the middle of the front page are links to "Tickets" that are relevant to you (explained in a moment).
- On the right side is a menu with a lot of links to info about developing applications using MOOSE.
- The top menu allows you to navigate to different sections of the site.
- The first time you login you will need to go to "Preferences" and set your name and email address.



Trac Tickets

- Tickets are “bug reports” or feature requests for your application.
- If you run into a bug or think of a feature you would like, just click on “New Ticket” and fill out the form.
- When a new ticket is submitted and assigned an “Owner,” he or she will automatically receive email regarding it.
- If you don’t know who the owner should be, just leave it blank.
- To view tickets you can either use the links provided on the front page of the Wiki or click on “View Tickets” and select a report.
- When submitting bug reports, try to be as thorough as possible:
 - Attach relevant input files.
 - Reference a specific test or example in the repository.
- “Tickets” are “bug reports” for MOOSE must be submitted via www.github.com



Trac Build Status

- Any time a change is made to MOOSE, its modules, or any application, our regression test suite runs to ensure everything is still working.
- You can view the current status of the tests by clicking on “Build Status” in the upper right.
 - Yellow = In progress
 - Green = All tests passed
 - Red = Something (compilation, testing, etc.) failed.
- You can click on individual builds to get more information and diagnose failures.

Screenshot of the Trac Build Status interface for MOOSE. The top navigation bar shows "Build Status - MOOSE". Below it is a search bar and a menu with links like Wiki, Timeline, Roadmap, Browse Source, View Tickets, New Ticket, Search, and Help/Guide.

The main area is titled "MOOSE Trac" and displays "Build Status". It shows 10 pending builds and 1 in-progress build. A checkbox labeled "Show In-progress configurations" is checked, with a "Update" button next to it.

A section titled "All Applications because of MOOSE" lists 4 pending builds (Helios, IceStorm, Snout), 1 in-progress build (Helios-stable), and a "Rebuilds all the applications because of a checkin to Moose." button.

The "Latest builds" section shows several builds for Helios, Helios-stable, Helios-ubuntu, and Snout. The Helios-stable build is highlighted in yellow and labeled "In-progress". Other builds are shown in green ("Success") or red ("Failure").

A "Applications because of ELK" section shows rebuilds for various applications due to a checkin to Elk. This section also includes a "Rebuilds applications that rely on Elk because of a checkin to Elk" button.

MOOSE External GitHub Development



www.inl.gov

Working With a GitHub Clone

- Users (non-developers) of MOOSE should follow the instructions at <http://mooseframework.org/getting-started>. After setting up your environment, the steps are:

1. Clone MOOSE

```
mkdir projects
cd projects
git clone https://github.com/idaholab/moose.git
git checkout master
```

2. Compile libMesh

```
cd moose
scripts/update_and_rebuild_libmesh.sh
```

3. Compile and test MOOSE all in one step:

```
cd test
make
./run_tests
```

- If you plan to develop in MOOSE, alternate instructions² are available.

²https://mooseframework.org/static/media/uploads/docs/moose_github.pdf

Basic Git Usage

- To see a brief summary of all current (not yet committed) modifications:

```
git status
```

- If you've added files, git will list them as being "untracked". Add each one you'd like to commit:

```
git add filename
```

- To see changes you've made to an existing file, type:

```
git diff filename
```

- When you are ready to (locally) commit your changes, type:

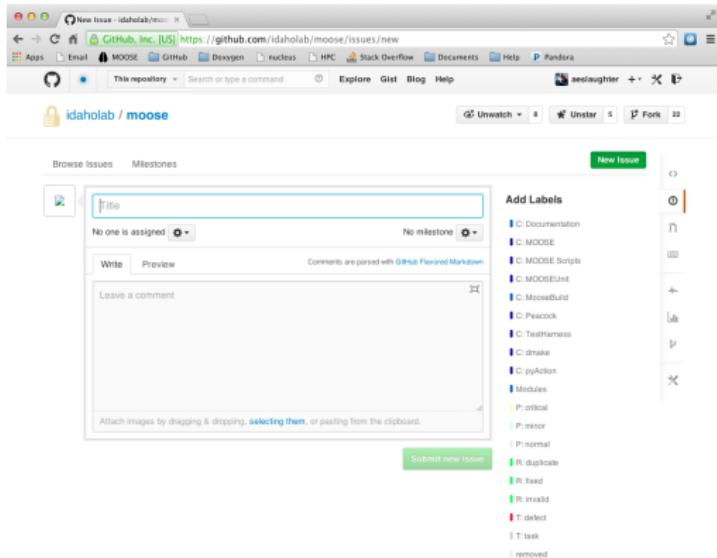
```
git commit -a -m"An informative message about your work."
```

- To see a list of recent commits:

```
git log
```

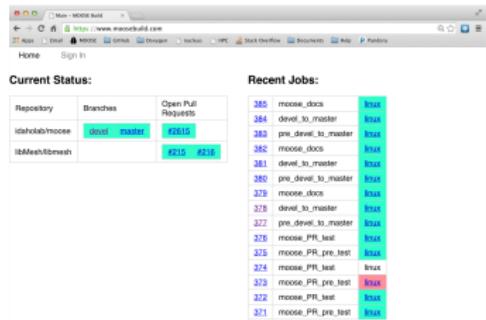
GitHub Issues

- Issues are bug reports or feature requests for MOOSE
- If you run into a bug or think of a feature you would like, just click on “New Issue” and fill out the form.
- When submitting bug reports, try to be as thorough as possible:
 - Attach relevant input files.
 - Reference a specific test or example in the repository.



Build Status

- Any time a change is made to MOOSE or its modules our regression test suite runs to ensure everything is still working.
- The current build status is shown on mooseframework.org
- Additional details are available on moosebuild.com
 - Yellow = In progress
 - Green = All tests passed
 - Red = Something (compilation, testing, etc.) failed.
- The status of “Pull Requests” are also included



© Copyright 2014

Visualization Tools

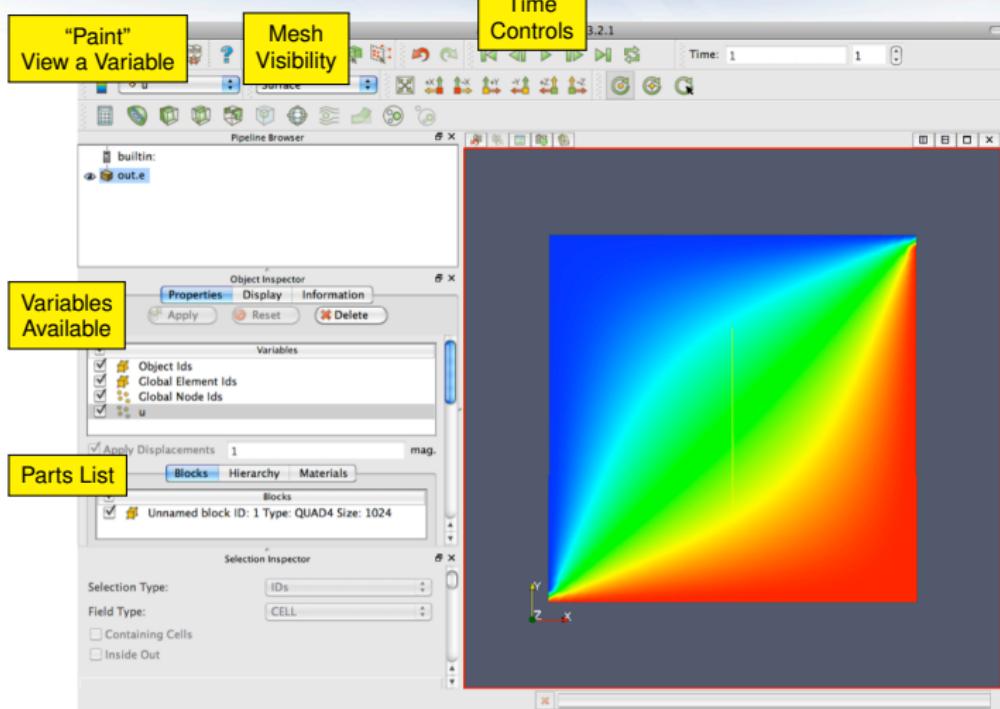


www.inl.gov

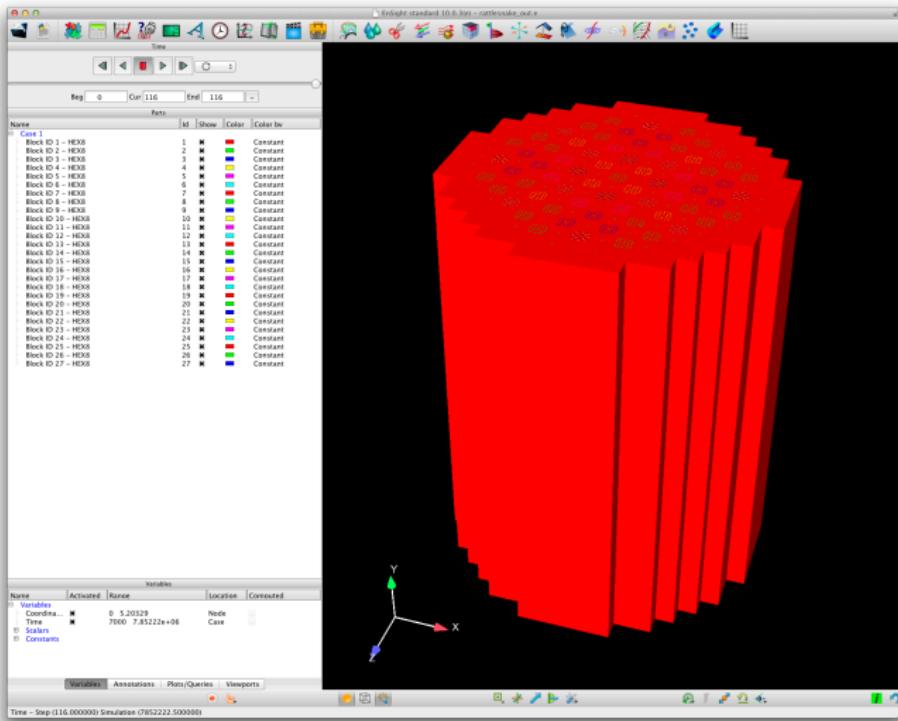
Visualization

- MOOSE can write results in several formats:
 - ExodusII
 - GMV
 - Tecplot
 - Miscellaneous others...
- The INL has a network Tecplot license. For installation instructions, see:
 - <http://hpcweb.inel.gov/home/Software/Tecplot>
- Paraview is a free visualization and data analysis program that reads ExodusII files:
 - <http://www.paraview.org>

Paraview Interface



Ensight Interface



Stork



www.inl.gov

Using Stork with an SVN checkout

- Stork is a template application for “delivering” new applications.
- To create a new internal application, run the following commands:

```
svn co https://hpcsc/svn/herd/trunk
cd trunk/stork
./make_new_application.py <new animal name>
```

- This will create a new barebones application that contains:
 - The standard app directory layout.
 - The standard main.C file.
 - A basic Application Object where you can register new physics.
 - A Makefile that will link against MOOSE and its modules.

Using Stork on GitHub

- Go to <http://github.com/idaholab/stork> and click the Fork button.
- Rename your fork
 - Click on your Repository, then the Settings button.
 - Rename your repository.
- Clone your fork and run `make_new_application.py`

```
cd ~/projects
git clone https://github.com/<username>/<app_name>.git
cd <app_name>
./make_new_application.py
```

- Commit and push

```
git commit -a -m"Starting my new application."
git push
```

The Actions System

Advanced Topic

www.inl.gov



The Actions System

- Responsible for performing work to set up a MOOSE problem or problems
- MOOSE has a large set of common tasks that are performed for every problem
- Supports complete dependency resolution
- Customizable and extendable

Actions System Model (tasks)

- The setup of a problem in MOOSE is very order sensitive.
- The required order for these steps has already been defined for all MOOSE-based applications and can be viewed in `Moose.C`.
- Each step of the setup phase is labeled as a `task`.
- MOOSE looks for Actions that will satisfy the current `task`.
- Some of these steps are optional, but they still have to be acted on in the correct order.
- The list of steps can be augmented by registering and injecting a new `task` into the setup order:
 - `registerTask("task", is_required);`
 - `addTaskDependency("task", "depends_on");`

Actions System Model (Action Object)

- An Action is another MOOSE object type.
- When building a new Action, inherit from Action and override `act()`.
- Typically you will set up other MOOSE objects with your Action:
 - Set up and add a group of Kernels...
 - Set up a group of Variables, each with their own Kernel...
 - Inspect other Actions in the warehouse, and add objects appropriately...
- A single Action `class` can be registered with more than one task.
- A single Action `instance` can only satisfy one task, however the system may create several instances of any one Action to satisfy multiple tasks.

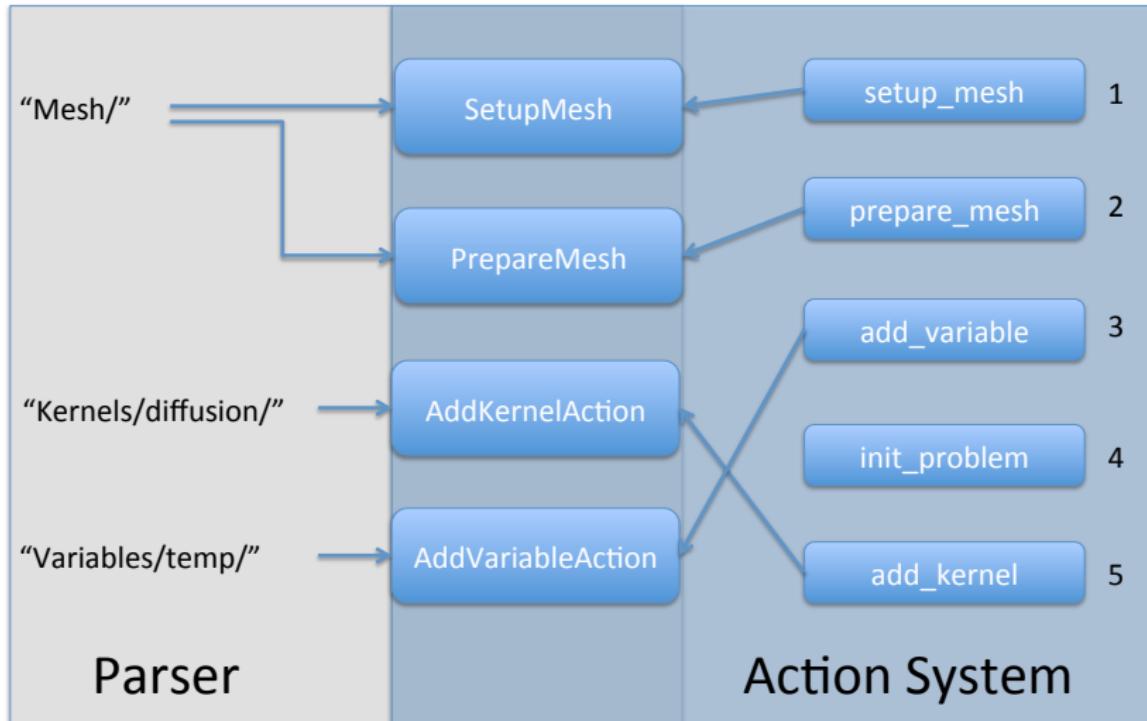
Some built in MOOSE “tasks”

"setup_mesh"	SetupMeshAction
"add_variable"	AddVariableAction
"add_aux _variable"	AddVariableAction
"add_kernel"	AddKernelAction
"add_bc"	AddBCAction
"add_postprocessor"	AddPostprocessorAction
"setup_executioner"	SetupExecutionerAction
"setup_mesh_complete"	
"init_problem"	InitProblemAction
"check_integrity"	CheckIntegrityAction

The Parser (briefly)

- MOOSE comes with an input file parser that will automatically create Actions for you based on the syntax it sees.
- Using the MOOSE Parser (or any parser) is not required. MOOSE operates on Actions which can be built in any way the developer chooses.
- To use the Parser, Actions must first be associated with syntax blocks (this works much like object registration).
- The Parser:
 1. Encounters a new syntax block in the input file
 2. Checks to see if there is an Action associated with that block
 3. Gets the InputParameters for your object using the appropriate validParams function
 4. Parses and fills in the InputParameters object with options from the input file.
 5. Builds and stores the appropriate Action in MOOSE's ActionWarehouse.

Actions System Model



Look at Example 15 (page E107)

TimeSteppers

Advanced Topic

www.inl.gov



TimeSteppers

- TimeSteppers are lightweight objects used for computing suggested time steps for transient executioners.
- Use by extending `TimeStepper` and overriding `computeDT()`
- Using a `TimeStepper` is easier than extending the `Transient` class if all you want to do is provide a custom method for picking the time step.
- TimeSteppers have access to current and old values of `time` and `dt` as well as access to the `Problem` and `Executioner`

Built-in TimeSteppers

- MOOSE has several built-in TimeSteppers
 - ConstantDT
 - SolutionTimeAdaptiveDT
 - FunctionDT
 - PostprocessorDT
 - DT2
- Example 16 creates a custom `TimeStepper` that multiplies the current time step size by a fixed ratio each time step until it reaches a user-specified minimum value.

Look at Example 16 (page E115)

Dirac Kernels

Advanced Topic

www.inl.gov



Point Source

- Often in engineering applications you will want to apply a load or source function at a single point in the domain.
- This happens when the load needs to be applied over a much smaller area than the domain of interest.
 - e.g. an injection well in a geothermal reservoir or a tiny defect in a mesoscale simulation
- For MOOSE this is just another term in your PDE:

$$-\nabla \cdot \nabla u - f = 0,$$

- where f is some forcing function that you want to apply at a point.
- Unfortunately $\int f = 0$ since f only has a value at a single point.
- To deal with that we're going to use a Dirac Delta Distribution...

Dirac Deltas To The Rescue

- A Dirac Delta Distribution (commonly referred to as a Dirac Delta Function and denoted by δ) is a generalized function that is zero except at one point (usually zero) and for which $\int_{-\infty}^{\infty} \delta = 1$. (From Wikipedia).
- Dirac Deltas also have another unique property under integration when multiplied by another function, $r(x)$:

$$\int_{-\infty}^{\infty} \delta(x - x_0) r(x) dx = r(x_0)$$

- The delta function “picks out the value of r at the point x_0 ”
- The idea is that instead of just using f we’re going to use $f\delta$, with the delta function non-zero at the point where f is applied.
- Now we can form the weak form of that piece of the PDE:

$$(-f\delta(x - x_0), \psi_i) = \sum_e \int_{\Omega_e} -f\delta(x - x_0)\psi_i = -f\psi_i(x_0)$$

- That is: if the point at which δ is nonzero is within an element, we get a contribution to each DOF with a shape function active on that element
- We need to evaluate the shape function at x_0 and multiply the result by the point load.
- This is what a `DiracKernel` achieves.

Dirac Kernels

- A DiracKernel provides a residual (and optionally a Jacobian) at a set of points in the domain.
- The structure is very similar to kernels
 - computeQpResidual / computeQpJacobian()
 - Parameters
 - Coupling
 - Material Properties
 - etc.
- The only difference is that DiracKernel *must override* addPoints() to tell MOOSE the points at which DiracKernel is active.
- Inside of addPoints() there are two different ways to notify MOOSE of the points:
 - addPoint(Point p)
 - Adds the physical point p that lies inside the domain of the problem.
 - addPoint(Elem* e, Point p)
 - Adds the physical point p that lies inside the element e .
- The second version is *much* more efficient if you know, a-priori, the element in which the point is located.

(Some) Values Available to DiracKernels

- `_u, _grad_u`
 - Value and gradient of variable this DiracKernel is operating on.
- `_phi, _grad_phi`
 - Value (ϕ) and gradient ($\nabla \phi$) of the trial functions at the q-points.
- `_test, _grad_test`
 - Value (ψ) and gradient ($\nabla \psi$) of the test functions at the q-points.
- `_q_point`
 - XYZ coordinates of the current q-point.
- `_i, _j`
 - Current shape functions for test and trial functions respectively.
- `_qp`
 - Current quadrature point.
- `_current_elem`
 - A pointer to the current element that is being operated on.
- `_current_point`
 - The Point where the DiracKernel is currently being asked to compute.
- And more!

ExampleDirac.h / .C

```
#ifndef EXAMPLEDIRAC_H
#define EXAMPLEDIRAC_H

// MOOSE Includes
#include "DiracKernel.h"

//Forward Declarations
class ExampleDirac;

template<>
InputParameters validParams<ExampleDirac>();

class ExampleDirac : public DiracKernel
{
public:
    ExampleDirac(const std::string & name,
                 InputParameters parameters);

    virtual void addPoints();
    virtual Real computeQpResidual();

protected:
    Real _value;
    std::vector<Real> _point_param;
    Point _p;
};

#endif //EXAMPLEDIRAC_H
```

```
#include "ExampleDirac.h"
template<>
InputParameters validParams<ExampleDirac>()
{
    InputParameters params = validParams<DiracKernel>();
    params.addRequiredParam<Real>("value", "");
    params.addRequiredParam<std::vector<Real> >("point", "");
    return params;
}

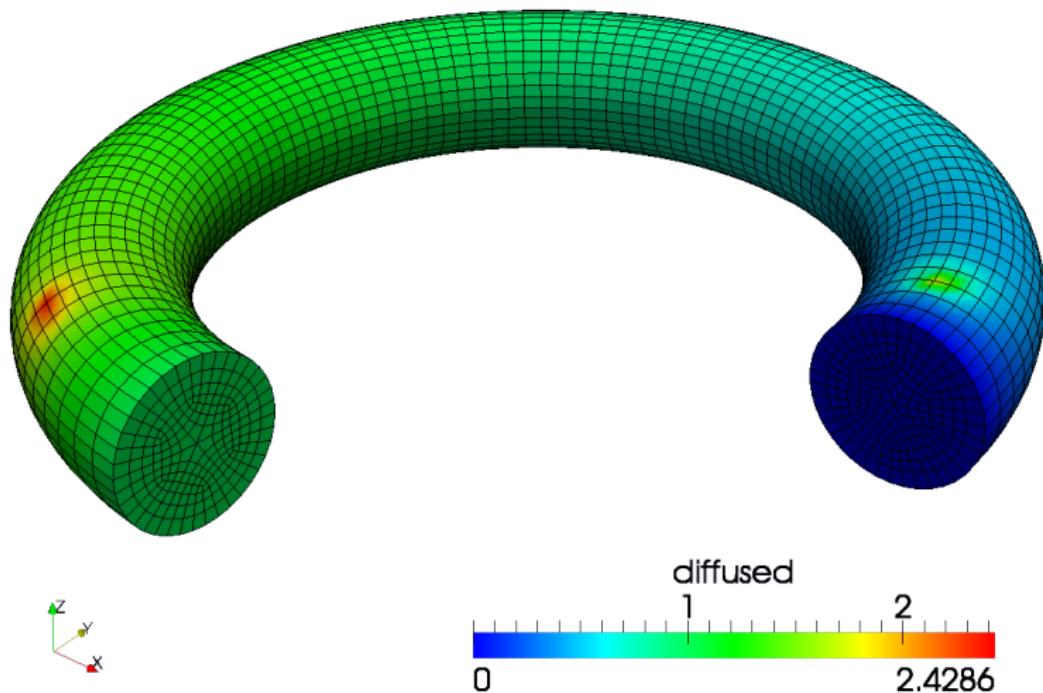
ExampleDirac::ExampleDirac(const std::string & name,
                           InputParameters parameters) :
    DiracKernel(name, parameters),
    _value(getParam<Real>("value")),
    _point_param(getParam<std::vector<Real> >("point"))
{
    _p(0) = _point_param[0];
    if(_point_param.size() > 1)
    {
        _p(1) = _point_param[1];
        if(_point_param.size() > 2)
            _p(2) = _point_param[2];
    }
}

void ExampleDirac::addPoints()
{
    addPoint(_p);
    addPoint(Point(4.9, 0.9, 0.9));
}

Real ExampleDirac::computeQpResidual()
{
    return -_test[_i][_qp]*_value;
}
```

Look at Example 17 (page E121)

Example 17 Output



Scalar Kernels

Advanced Topic

www.inl.gov



Scalar Kernels

- Scalar Kernels:
 - Operate on *scalar* variables (`family = SCALAR`).
 - Are defined in the `[ScalarKernels]` section of your input file.
- Use them for:
 - Solving ODEs (see example 18).
 - Formulations with Lagrange multipliers.
 - Contact
 - Other applications...
- Notes:
 - Mesh-specific data such as `_qp` and `_current_elem` are not available to `ScalarKernels`.

Scalar Kernels

- Problem being solved:

$$\begin{aligned} \frac{\partial u}{\partial t} &= \nabla^2 u + f && \text{in } \Omega = [-1, 1] \\ u &= X(t) && \text{on } \Gamma_{\text{left}} \\ u &= Y(t) && \text{on } \Gamma_{\text{right}} \end{aligned}$$

- Where the boundary conditions are governed by the ODEs:

$$\frac{dX}{dt} = 3X + 2Y$$

$$\frac{dY}{dt} = 4X + Y$$

plus suitable initial conditions.

ImplicitODE.h / .C

```
#ifndef IMPLICITODEX_H
#define IMPLICITODEX_H

#include "ODEKernel.h"

class ImplicitODEx;

template<>
InputParameters validParams<ImplicitODEx>();

class ImplicitODEx : public ODEKernel
{
public:
    ImplicitODEx(const std::string & name,
                 InputParameters parameters);

protected:
    virtual Real computeQpResidual();
    virtual Real computeQpJacobian();
    virtual Real computeQpOffDiagJacobian(unsigned int jvar);

    unsigned int _y_var;
    VariableValue & _y;
};


```

```
#include "ImplicitODEx.h"

template<>
InputParameters validParams<ImplicitODEx>()
{
    InputParameters params = validParams<ODEKernel>();
    params.addCoupledVar("y", "coupled variable Y");
    return params;
}

ImplicitODEx::ImplicitODEx(const std::string & name,
                           InputParameters parameters) :
    ODEKernel(name, parameters),
    _y_var(coupledScalar("y")),
    _y(coupledScalarValue("y"))
{
}

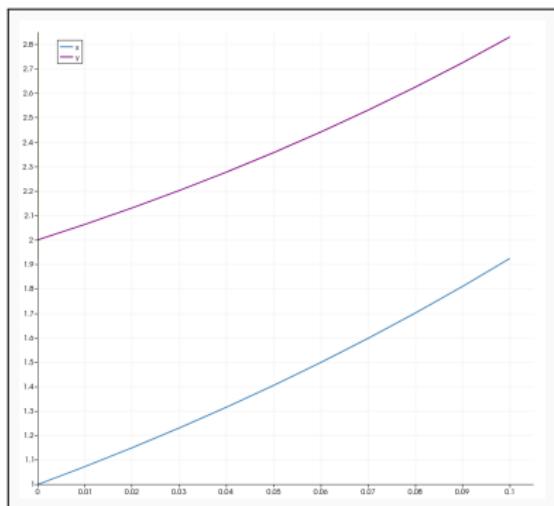
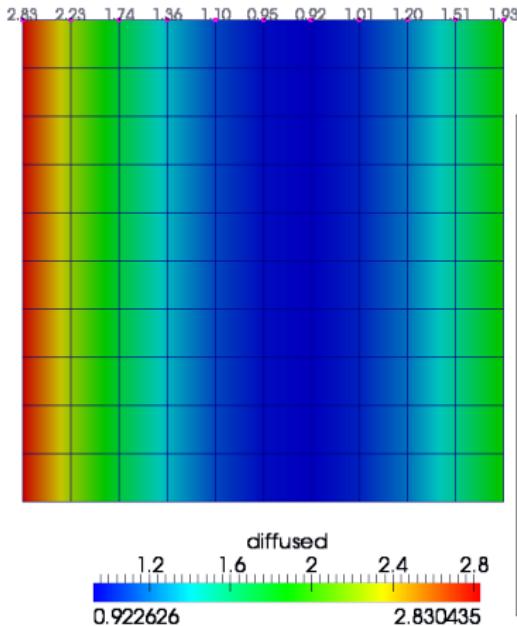
Real
ImplicitODEx::computeQpResidual()
{
    return -3. * _u[_i] - 2. * _y[_i];
}

Real
ImplicitODEx::computeQpJacobian()
{
    return -3.;
}

Real
ImplicitODEx::computeQpOffDiagJacobian(unsigned int jvar)
{
    if (jvar == _y_var)
        return -2.;
    else
        return 0.;
}
```

Look at Example 18 (page E127)

Example 18 Output



Geometric Search

Advanced Topic

www.inl.gov

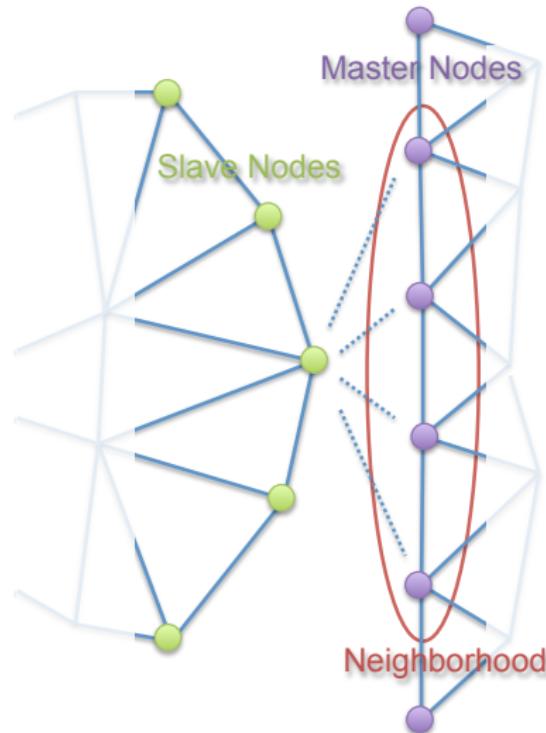


Geometric Search

- Sometimes information needs to be exchanged between disconnected pieces of mesh.
- Examples include:
 - Mechanical Contact
 - Gap Heat Conduction
 - Radiation
 - Constraints
 - Mesh Tying
- The Geometric Search system allows an application to track evolving geometric relationships.
- Currently, this entails two main capabilities: `NearestNodeLocator` and `PenetrationLocator`.
- Both of the capabilities work in parallel and with both Parallel- and Serial-Mesh.

NearestNodeLocator

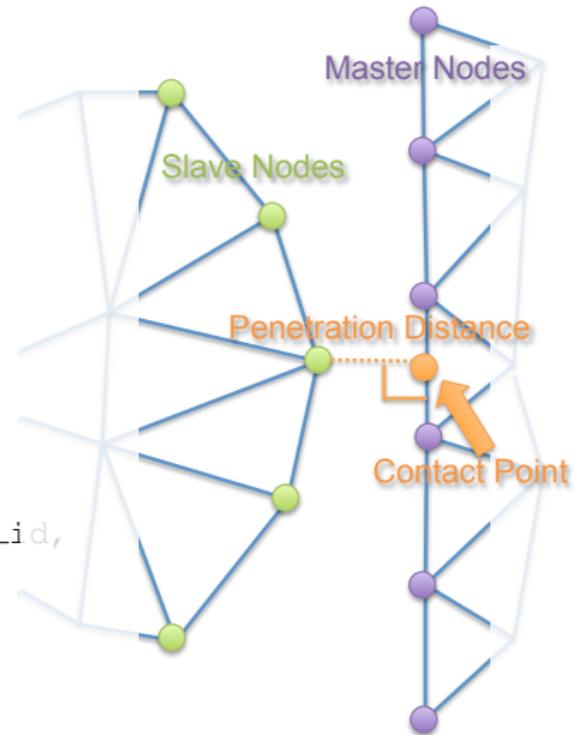
- NearestNodeLocator provides the nearest node on a “Master” boundary for each node on a “Slave” boundary (and the other way around).
- The distance between the two nodes is also provided.
- It works by generating a “Neighborhood” of nodes on the Master side that are close to the Slave node.
- The size of the Neighborhood can be controlled in the input file by setting the `patch_size` parameter in the Mesh section.
- To get a `NearestNodeLocator` `#include "NearestNodeLocator.h"` and call `getNearestNodeLocator(master_id, slave_id)` to create the object.
- The functions `distance()` and `nearestNode()` both take a node ID and return either the distance to the nearest node or a `Node` pointer for the nearest node respectively.



PenetrationLocator

- A PenetrationLocator provides the perpendicular distance from a Slave node to a Master side and the “contact point” on the Master side.
- The distance returned is negative if penetration hasn’t yet occurred and positive if it has.
- To get a NearestNodeLocator


```
#include
"PenetrationLocator.h" and
call
getPenetrationLocator(master_id,
slave_id) to create the object.
```
- The algorithm in PenetrationLocator utilizes a NearestNodeLocator so patch_size is still important.



Dampers

Advanced Topic



www.inl.gov

Dampers

- Dampers allow computation of the Newton damping (or scaling) parameter alpha:

Regular Newton

$$\mathbf{J}\delta\mathbf{u}_{n+1} = -\mathbf{R}(\mathbf{u}_n)$$

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \delta\mathbf{u}_{n+1}$$

Damped Newton

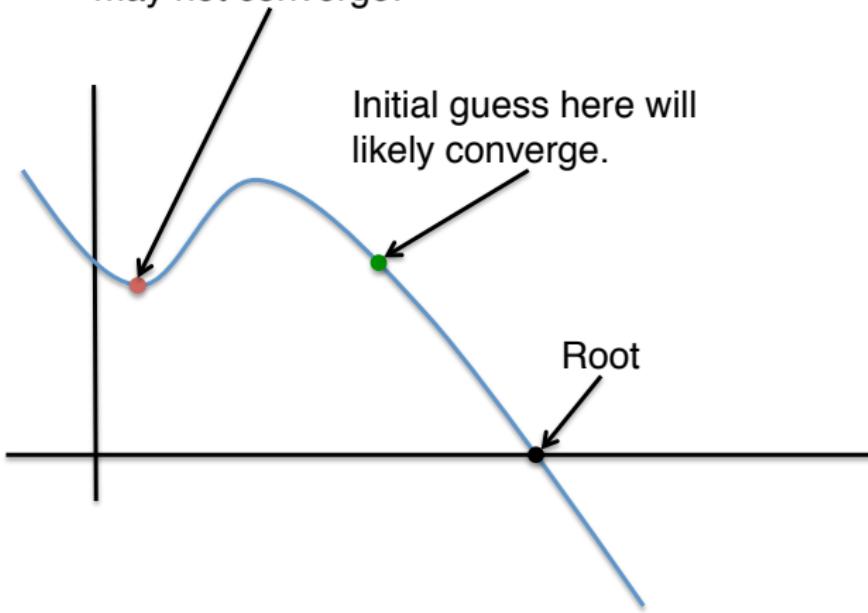
$$\mathbf{J}\delta\mathbf{u}_{n+1} = -\mathbf{R}(\mathbf{u}_n)$$

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \alpha\delta\mathbf{u}_{n+1}$$

- Where α is a number between 0 and 1.
 - Essentially taking some fraction of the step you just solved for.
- A Damper is created by inheriting from Damper and overriding computeQpDamping().
- computeQpDamping() computes a damping parameter at each quadrature point throughout the mesh.
 - The smallest damping parameter computed is then actually used.
- Dampers have access to variable values, gradients, material properties and functions just like a kernel.
 - In addition they have access to the Newton Increment.

Initial guess here
may not converge!

Initial guess here will
likely converge.



Look at Example 19 (page E139)

Example 19 Output

Damping = 1.0

Outputting Initial Condition

```
True Initial Nonlinear Residual: 10.4403
NL step 0, |residual|_2 = 1.044031e+01
NL step 1, |residual|_2 = 6.366756e-05
NL step 2, |residual|_2 = 3.128450e-10
0: 3.128450e-10
```

Damping = 0.9

Outputting Initial Condition

```
True Initial Nonlinear Residual: 10.4403
NL step 0, |residual|_2 = 1.044031e+01
NL step 1, |residual|_2 = 1.044031e+00
NL step 2, |residual|_2 = 1.044031e-01
NL step 3, |residual|_2 = 1.044031e-02
NL step 4, |residual|_2 = 1.044031e-03
NL step 5, |residual|_2 = 1.044031e-04
NL step 6, |residual|_2 = 1.044031e-05
NL step 7, |residual|_2 = 1.044031e-06
NL step 8, |residual|_2 = 1.044031e-07
NL step 9, |residual|_2 = 1.044031e-08
0: 1.044031e-08
```

Discontinuous Galerkin

Advanced Topic

www.inl.gov



Discontinuous Galerkin

- Discontinuous Galerkin is a set of methods that rely on a discontinuous set of shape functions across the domain.
- Essentially, the idea is to allow discontinuities in the value of the function you are solving for along element boundaries.
- DG then relies on penalizing the jump in the integral of quantities along element boundaries.
- MOOSE has full support for DG.
- You can use DG with MOOSE by utilizing a discontinuous set of shape functions (like the MONOMIALS) and creating / specifying DGKernels to run along with regular Kernels.
- DGKernels are responsible for residual and Jacobian contributions coming from the integral of jump terms along inter element edges.
- DG is beyond the scope of this training class, if you want to utilize this capability just ask!

UserObjects

Advanced Topic



www.inl.gov

UserObjects

- The `UserObject` system provides data and calculation results to other MOOSE objects.
- All Postprocessors are UserObjects that compute a single scalar value.
- Therefore, a `UserObject` can often be thought of as a more generic Postprocessor with more functionality.
- UserObjects define their own interface, which other MOOSE objects can call to retrieve data.
- Just like Postprocessors, there are 4 types of UserObjects:
 - `ElementUserObject`
 - `NodalUserObject`
 - `SideUserObject`
 - `GeneralUserObject`
- The first three perform a calculation on the associated geometric entity, and then provide an interface to the result.
- The `GeneralUserObject` can do “anything,” and then provide an interface to the result.
- For example, a `GeneralUserObject` might read in a large data set, hold it in memory, and provide an interface for Material classes to access the data.

UserObject Anatomy

All UserObjects must override the following functions:

1. `virtual void initialize();`
 - Called just one time before beginning the `UserObject` calculation.
 - Useful for resetting data structures.
2. `virtual void execute();`
 - Called once on each geometric object (element, node or side) or just one time per calculation for a `GeneralUserObject`.
 - This is where you actually do your calculation, read data, etc.
3. `virtual void threadJoin(const UserObject & y);`
 - During threaded execution this function is used to “join” together calculations generated on different threads.
 - In general you need to cast `y` to a `const` reference of your type of `UserObject`, then extract data from `y` and add it to the data in “this” object.
 - Note, this is not required for a `GeneralUserObject` because it is *not* threaded.
4. `virtual void finalize();`
 - The very last function called after all calculations have been completed.
 - In this function, the user must take all of the small calculations performed in `execute()` and do some last operation to get the final values.
 - Be careful to do parallel communication where necessary to ensure all processors compute the same values.

UserObject Anatomy (Cont.)

- A UserObject defines its own interface by defining `const` accessor functions.
- When another MOOSE object uses a UserObject, they do so by calling these accessor functions.
- For example, if a UserObject is computing the average value of a variable on every block in the mesh, it might provide a function like:

```
Real averageValue(SubdomainID block) const;
```

- Another MOOSE object using this UserObject would then call `averageValue()` to get the result of the calculation.
- Take special note of the `const` at the end of the function declaration!
- This `const` means the function cannot modify any member variables of the object, and is *required* for UserObject accessor functions.

Using UserObjects

- Any MOOSE object can retrieve a `UserObject` in a manner similar to retrieving a `Function`.
- Generally, it is a good idea to take the name of the `UserObject` to use from the input file:

```
template<>
InputParameters validParams<BlockAverageDiffusionMaterial>()
{
    InputParameters params = validParams<Material>();
    params.addRequiredParam<UserObjectName>("block_average_userobject", "Doc");
    return params;
}
```

- A `UserObject` comes through as a `const` reference of the `UserObject` type. So, in your class:

```
const BlockAverageValue & _block_average_value;
```

- Set the reference in the initialization list of your object by calling the templated `getUserObject()` function:

```
BlockAverageDiffusionMaterial::BlockAverageDiffusionMaterial(const std::string & name,
                                                               InputParameters parameters) :
    Material(name, parameters),
    _block_average_value(getUserObject<BlockAverageValue>("block_average_userobject"))
{}
```

- Use the reference by calling some of the interface functions defined by the `UserObject`:

```
_diffusivity[_qp] = 0.5 * _block_average_value.averageValue(_current_elem->subdomain_id());
```

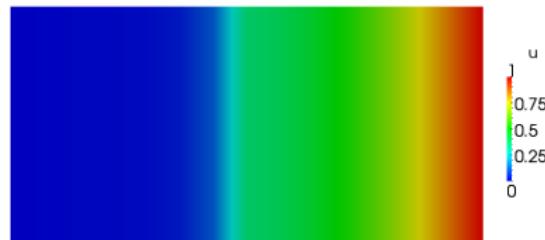
Example 20 Overview

- The problem is time-dependent diffusion with Dirichlet boundary conditions of 0 on the left and 1 on the right.
- The diffusion coefficient being calculated by the `Material` is dependent on the average value of a variable on each block.
- This means that as the concentration diffuses, the diffusion coefficient is getting larger, but the coefficient is different for each block (based on the average value of the variable on that block).
- To achieve this we need 3 objects working together:
 - `BlockAverageValue`: A `UserObject` that computes the average value of a variable on each block of the domain and provides `averageValue()` for retrieving the average value on a particular block.
 - `BlockAverageDiffusionMaterial`: A `Material` that computes diffusivity based on the average value of a variable as computed by a `BlockAverageValue` `UserObject`.
 - `ExampleDiffusion`: The same Kernel we've seen before that uses a diffusivity `material` property.

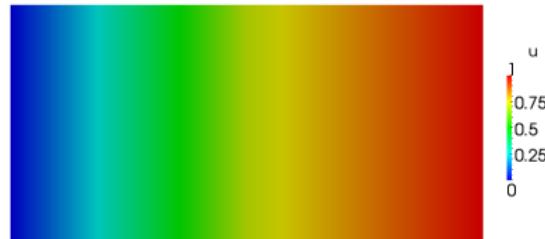
Look at Example 20 (page E143)

Example 20 Output

After 4 time steps:



After 10 time steps:



MultiApps

Advanced Topic

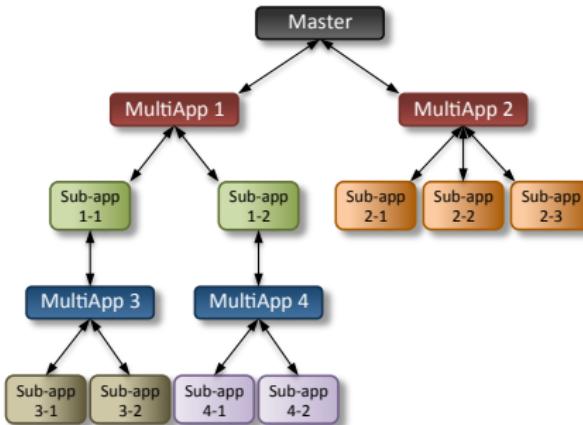


www.inl.gov

MultiApps

- MOOSE was originally created to solve fully-coupled systems of PDEs.
- Not all systems need to be / are fully coupled:
 - Multiscale systems are generally loosely coupled between scales.
 - Systems with both fast and slow physics can be decoupled in time.
 - Simulations involving input from external codes might be solved somewhat decoupled.
- To MOOSE these situations look like loosely-coupled systems of fully-coupled equations.
- A MultiApp allows you to simultaneously solve for individual physics systems.

MultiApps (Cont.)



- Each “App” is considered to be a solve that is independent.
- There is always a “master” App that is doing the “main” solve.
- A “master” App can then have any number of MultiApps.
- Each MultiApp can represent many (hence Multi!) “sub-apps”.
- The sub-apps can be solving for completely different physics from the main application.
- They can be other MOOSE applications, or might represent external applications.
- A sub-app can, itself, have MultiApps... leading to multi-level solves.

Input File Syntax

```
[MultiApps]
[./some_multi]
  type = TransientMultiApp
  app_type = SomeApp
  execute_on = timestep
  positions = '0.0 0.0 0.0
               0.5 0.5 0.0
               0.6 0.6 0.0
               0.7 0.7 0.0'
  input_files = 'sub.i'
[]
[...]
```

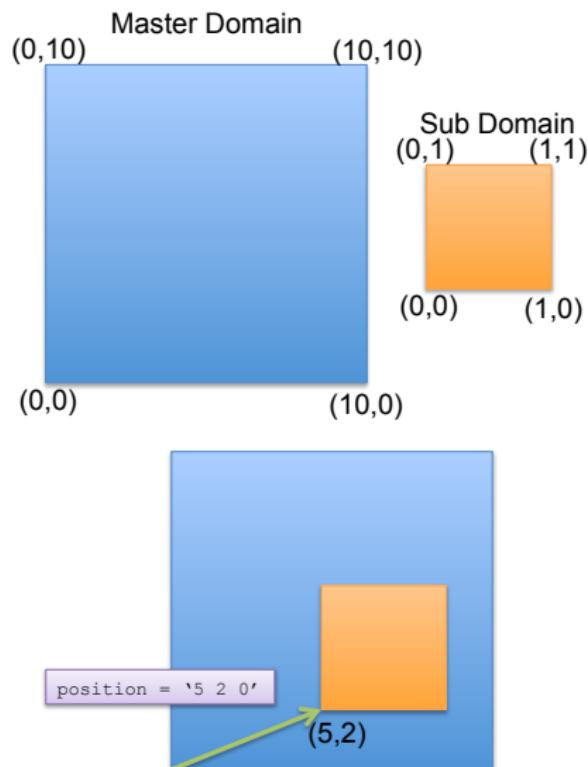
- MultiApps are declared in the MultiApps block.
- They require a type just like any other block.
- app_type is required and is the name of the MooseApp derived App that is going to be run. Generally this is something like AnimalApp.
- A MultiApp can be executed at any point during the master solve. You set that using execute_on to one of: initial, residual, jacobian, timestep_begin, or timestep.
- positions is a list of 3D coordinate pairs describing the offset of the sub-application into the physical space of the master application. More on this in a moment.
- You can either provide one input file for all the sub-apps... or provide one per position.

TransientMultiApp

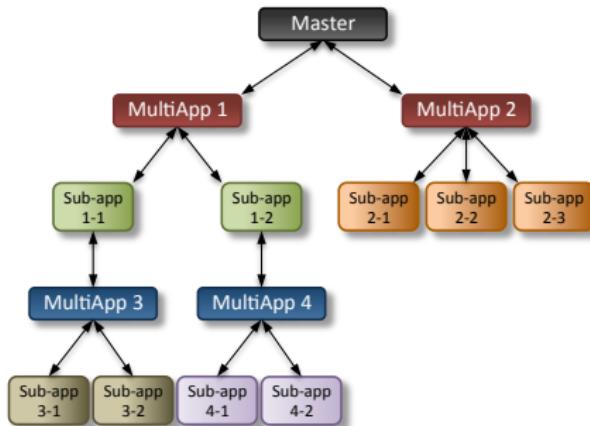
- The only currently-available MultiApp is TransientMultiApp, but that will change.
- A TransientMultiApp requires that your “sub-apps” use an Executioner derived from Transient.
- A TransientMultiApp will be taken into account during time step selection inside the “master” Transient executioner.
- Currently, the minimum `dt` over the master and all sub-apps is used.
- That situation will change when we add the ability to do “sub-cycling.”

Positions

- The `positions` parameter allows you to define a “coordinate offset” of the sub-app’s coordinates into the master app’s domain.
- You must provide one set of (x, y, z) coordinates for each sub-app.
- The number of coordinate sets determines the actual number of sub-applications.
- If you have a large number of positions you can read them from a file using `positions_file = filename`.
- You can think of the (x, y, z) coordinates as a vector that is being added to the coordinates of your sub-app’s domain to put that domain in a specific spot within the master domain.
- If your sub-app’s domain starts at $(0, 0, 0)$ it is easy to think of moving that point around using `positions`.
- For sub-apps on completely different scales, `positions` is the point in the master domain where that App is.



Parallel



- The MultiApp system is designed for efficient parallel execution of hierarchical problems.
- The master application utilizes all processors.
- Within each MultiApp, all of the processors are split among the sub-apps.
- If there are more sub-apps than processors, each processor will solve for multiple sub-apps.
- All sub-apps of a given MultiApp are run simultaneously in parallel.
- Multiple MultiApps will be executed one after another.

Transfers

Advanced Topic



www.inl.gov

Transfers

- While a MultiApp allows you to execute many solves in parallel, it doesn't allow for data to flow between those solves.
- A Transfer allows you to move fields and data both to and from the "master" and "sub" applications.
- There are three main kinds of Transfers:
 - Field interpolation.
 - UserObject interpolation (volumetric value evaluation).
 - Value transfers (like Postprocessor values).
- Most Transfers put values into AuxVariable fields.
- The receiving application can then couple to these values in the normal way.
- Idea: each application should be able to solve on its own, and then, later, values can be injected into the solve using a Transfer, thereby coupling that application to the one the Transfer came from.

Field Interpolation

```
[Transfers]
[./from_sub]
  type = MultiAppMeshFunctionTransfer
  direction = from_multiapp
  multi_app = sub
  source_variable = sub_u
  variable = transferred_u
[...]
[./to_sub]
  type = MultiAppMeshFunctionTransfer
  direction = to_multiapp
  multi_app = sub
  source_variable = u
  variable = from_master
[...]
[]
```

- An “interpolation” Transfer should be used when the domains have some overlapping geometry.
- The source field is evaluated at the destination points (generally nodes or element centroids).
- The evaluations are then put into the receiving AuxVariable field named variable.
- All MultiAppTransfers take a direction parameter to specify the flow of information. Options are: from_multiapp or to_multiapp.

UserObject Interpolation

```
[Transfers]
[./layered_transfer]
type = MultiAppUserObjectTransfer
direction = from_multiapp
multi_app = sub_app
user_object = layered_average
variable = multi_layered_average
[...]
[]
```

- Many UserObjects compute spatially-varying data that isn't associated directly with a mesh.
- Any UserObject can override Real spatialValue(Point &) to provide a value given a point in space.
- A UserObjectTransfer can sample this spatially-varying data from one App, and put the values into an AuxVariable in another App.

Single Value Transfers

```
[Transfers]
[./sample_transfer]
type = MultiAppVariableValueSampleTransfer
direction = to_multiapp
multi_app = sub
execute_on = timestep
source_variable = u
variable = from_master
[...]
[./pp_transfer]
type = MultiAppPostprocessorInterpolationTransfer
direction = from_multiapp
multi_app = sub
postprocessor = average
variable = from_sub
[...]
[]
```

- A single value transfer will allow you to transfer scalar values between applications.
- This is useful for Postprocessor values and sampling a field at a single point.
- When transferring to a MultiApp, the value can either be put into a Postprocessor value or can be put into a constant AuxVariable field.
- When transferring from a MultiApp to the master, the value can be interpolated from all the sub-apps into an auxiliary field.

Debugging

Advanced Topic



www.inl.gov

Debugging

- During development, you will reach points where your code is not running correctly.
- Sometimes simple print statements will inform you of what is going wrong.
- In cases where your program is actually “crashing” a “debugger” can help pinpoint the problem.
- Many debuggers exist: GDB, Totalview, ddd, Intel Debugger, etc.
- We are going to focus on GDB because of its ubiquity and simplicity.
- A “Segmentation fault” or “Segfault” or “Signal 11” is a common error, and denotes a memory bug (often array access out of range).
- In your terminal you will see a message like:

```
Segmentation fault: 11
```

- A segfault is a “good” error to have, because a debugger can easily pinpoint the problem.

Example 21 (page E155)

- Example 21 is exactly like Example 8, except a common error has been introduced.
- In `ExampleDiffusion.h`, a `MaterialProperty` that should be declared as a reference is not:

```
MaterialProperty<Real> _diffusivity;
```

- Not storing this as a reference will cause a *copy* of the `MaterialProperty` to be made.
- That copy will never be resized, nor will values ever be inserted into it.
- Attempting to access that `MaterialProperty` results in a segfault:

```
Solving time step  1, time=0.100000...
Segmentation fault: 11
```

- We can use a debugger to help us find the problem...

Debug Executable

- To use a debugger with a MOOSE-based application, you must do the following:
 1. Compile your application in debug mode:

```
cd ~/projects/moose/examples/ex21_debugging  
METHOD=dbg make -j8
```

- You will now have a “debug version” of your application called appname-dbg.
- Next, you need to run your application using GDB:

```
gdb --args ./ex21-dbg -i ex21.i
```

- --args tells GDB that any arguments after the executable should be passed to the executable.
- This will start GDB, load your executable, and leave you at a GDB command prompt.

Using GDB or LLDB

- At any prompt in GDB or LLDB, you can type `h` and hit enter to get help.
- We need to set a breakpoint at `MPI_Abort` in order to catch any crashes:

```
b MPI_abort
```

- To run your application, type `r` and hit enter.
- After your application has crashed, type `where` (or `bt`) to see a “backtrace”:

```
#0 0x0101e2baff in MPI_Abort ()
#1 0x0100008d89 in MooseArray<double>::operator[] (this=0x1028c6200, i=1) at MooseArray.h:256
#2 0x0100007cf0 in ExampleDiffusion::computeQpResidual (this=0x1028c5a00) at ExampleDiffusion.C:44
#3 0x01008ed71a in Kernel::computeResidual (this=0x1028c5a00) at Kernel.C:133
....
```

- This backtrace shows that, in `ExampleDiffusion::computeQpResidual` at line 44 of the file `ExampleDiffusion.C`, we attempted to use `operator[]` to index into something, and failed.
- If we look at this line, we see:

```
return _diffusivity[_qp]*Diffusion::computeQpResidual();
```

- There is only one thing we’re indexing into on that line: `_diffusivity`.
- Therefore, we can look at how `_diffusivity` was declared, realize that we forgot an ampersand (&), and fix it!

THE END