Multi-physics Object Oriented Simulation Environment (MOOSE)

Casey Icenhour, Shane Keniley, Corey DeChant, Cody Permann, Alex Lindsay, Richard Martineau, Davide Curreli, Steven Shannon

November 2018

Multi-physics Object Oriented Simulation Environment (MOOSE)

Casey Icenhour, Shane Keniley, Corey DeChant, Cody Permann, Alex Lindsay, Richard Martineau, Davide Curreli, Steven Shannon

November 2018

Idaho National Laboratory
Idaho Falls, Idaho 83415

http://www.inl.gov

MOOSE Workshop

Enabling Plasma Simulation in the MOOSE Framework

The 71st Annual Gaseous Electronics Conference

MOOSE Workshop Contents I

MOOSE Workshop Contents II

MOOSE Workshop Contents III

MOOSE Overview

MOOSE Overview Contents

MOOSE Team (INL)

General MOOSE Framework content in this workshop has been developed by the MOOSE development team:

- Derek Gaston
  - derek.gaston@inl.gov
  - @friedmud
- Cody Permann
  - cody.permann@inl.gov
  - @permcody
- David Andrš
  - david.andrs@inl.gov
  - @andrsdave
- John Peterson
  - jw.peterson@inl.gov
  - @peterson512

- Jason Miller
  - jason.miller@inl.gov
  - @jmiller96
- Andrew Slaughter
  - andrew.slaughter@inl.gov
  - @aeslaughter98
- Fande Kong
  - fande.kong@inl.gov
- Robert Carlsen
  - robert.carlsen@inl.gov
- Alex Lindsay
  - alexander.lindsay@inl.gov

Plasma/EM Collaborators

Plasma-specific tutorial content has been developed by contributors from NCSU and UIUC:

- ▶ Prof. Steven Shannon (NCSU)
    - ▶ scshanno@ncsu.edu
- ▶ Prof. Davide Curreli (UIUC)
    - ▶ dcurreli@illinois.edu
- ▶ Casey Icenhour (INL/NCSU)
    - ▶ casey.icenhour@inl.gov
    - ▶ cticenho@ncsu.edu
- ▶ Shane Keniley (UIUC)
    - ▶ keniley1@illinois.edu
- ▶ Corey DeChant (NCSU)
    - ▶ csdechan@ncsu.edu

MOOSE: Multiphysics Object Oriented Simulation Environment

- ▶ A framework for solving computational engineering problems in a well-planned, managed, and coordinated way
- ▶ **Designed to significantly reduce the expense and time required to develop new applications**
  - ▶ *Maximize Science*
  - ▶ *Designed to be easily extended and maintained*
  - ▶ *Efficient on both a few and many processors*
  - ▶ *Provides an object-oriented, pluggable system for defining all aspects of a simulation tool.*
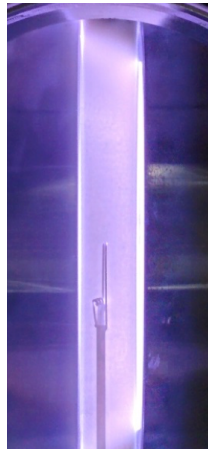
Ar CCP discharge with Hairpin Resonator Probe

(*Credit: Kris Ford, NCSU*)

Capabilities

- ▶ 1D, 2D, and 3D
  - ▶ User code agnostic of dimension
- ▶ Finite Element Based
  - ▶ Continuous and Discontinuous Galerkin (and Petrov Galerkin)
  - ▶ Nodal and Vector Basis Functions
- ▶ 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 . . .

Code Platform

- Uses well-established libraries
- Implements robust and state-of-the-art solution methods
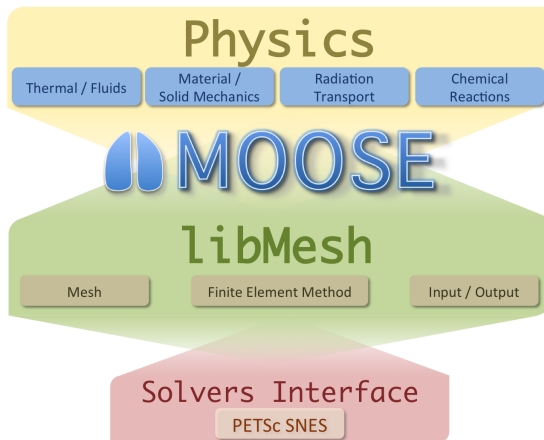
Rapid Development

Application	Physics	Results	Lines
BISON	Thermo-mechanics, Chemical, diffusion, coupled mesoscale	4 mo.	3,000
PRONGHORN	Neutronics, Porous flow, eigenvalue	3 mo.	7,000
MARMOT	4-th order phase-field meso-scale	1 mo.	6,000
RAT	Porous ReActive Transport	1 mo.	1,500
FALCON	Geo-mechanics, coupled meso-scale	3 mo.	3,000
MAMBA	Chemical reactions, precipitation, and porous flow	5 wks.	2,500
HYRAX	Phase-field, ZrH precipitation	3 mo.	1,000
PIKA	Multi-scale heat and mass transfer with phase-change	3 mo.	2,900
ZAPDOS	Plasma water interactions	3 mo.	3,000
ELK	Radio-frequency electromagnetics	3 mo.	2,500
CRANE	Multi-species plasma chemistry	4 mo.	3,000

MOOSE Application Architecture
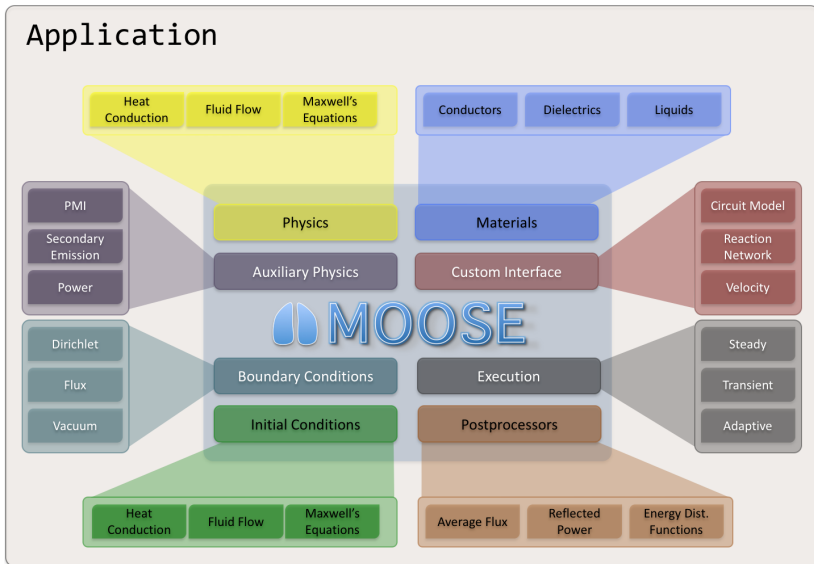
MOOSE Code Example

Strong Form

$$\frac{\partial C}{\partial t} - D\nabla^2 C = R$$

Weak Form

$$\int_\Omega \frac{\partial C}{\partial t}\psi_i + \int_\Omega D\nabla C \cdot \nabla\psi_i - \int_{\partial\Omega} D(\nabla C \cdot \hat{n})\psi_i - \int_\Omega R\psi_i = 0$$

Kernel   Kernel   BoundaryCondition   Kernel

Actual Code

```
return _D[_qp]*_grad_u[_qp]*_grad_test[_i][_qp];
```

MOOSE Software Quality Practices

- ▶ MOOSE follows an NQA-1 (Nuclear Quality Assurance Level 1) development process.
- ▶ 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 (see CodeStandards).
- ▶ We process code comments with third-party tools to generate documentation.
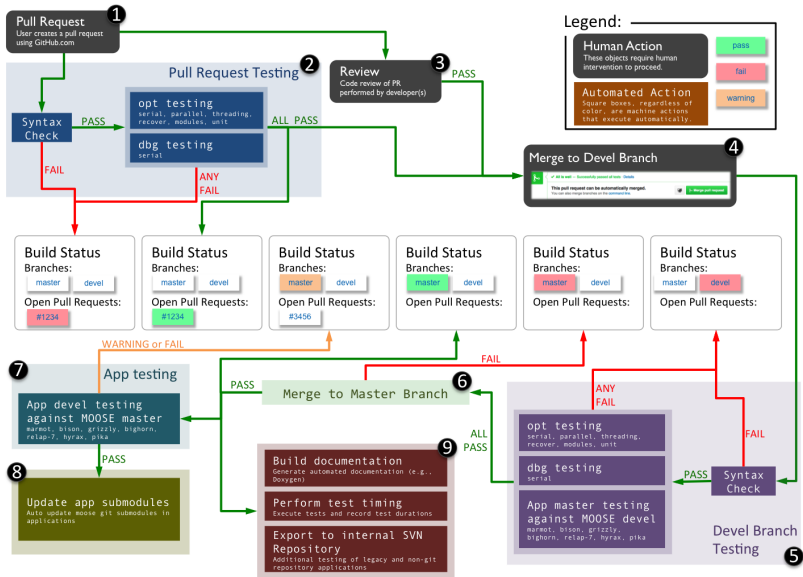
idaholab / moose ○

Current status

○ moose   devel   master

○	#6646	Two tests demonstrating the dependency issues by YaqiWang
○	#6817	Reinitialize geometric search when mesh changes, test with XFEM by bwspenc
○	#7070	Begin moving Jacobian graph augmentation into the Constraints by friedmud
○	#7118	Fix for variable dependency bug when initializing stateful properties. by aeslaughter
○	#7151	Allow coupling aux variables in aux kernels on displaced system by YaqiWang
○	#7245	Enable SolutionUserObject and SolutionFunction to evaluate gradients by snschune
○	#7288	Deprecate DiracKernel contact by bwspenc
○	#7776	executioner: move solver related params into FEProblem by rwcarlsen
○	#7896	Allow time stepper to control Transient output by zachmprince
○	#8017	clang-format for discussion by rwcarlsen
○	#8041	Full Makefile control of disabling/enabling modules! by permcody
○	#8044	Error out for multiply declared material property by bwspenc
○	#8063	Further cleaning in chemical_reactions by cpgr
○	#8069	Fix misleading-indentation warning in PorousFlowJoiner by cpgr
○	#8080	Add --redirect-stdout for select tests by milljm
○	#8098	Temperature-dependent thermal eigenstrain functions by bwspenc
○	#8108	Remove deprecated options (ParallelMesh) by permcody
○	#8127	Stepper System by friedmud
○	#8133	Re-factor MooseDocs to remove nested navigation by aeslaughter
○	#8153	add a master rand generator to each app/problem by rwcarlsen
○	#8155	Revamp automatic output in TensorMechanicsAction by dschwen
○	#8160	TestHarness output color modifications by milljm
○	#8180	Replacing several more raw pointers with smart pointers by permcody
○	#8183	Add missing tasks for DynamicTensorMechanicsAction by dschwen
○	#8184	Strip deprecated method by permcody
○	#8192	Reformatted "slider" and associated template web page: refs #6699 by tolman42
○	#8193	Implement a monotonic cubic interpolator with at least C1 continuity by lindsayad
○	#8198	Emit a warning when using a private parameter in an input file. by brianmoose
○	#8203	Added test for slider and fixed bibtex tests in moosedocs.py: refs #6699 by tolman42
○	#8206	Phase field fracture displacement vector by zhangsf2015
○	#8208	Make eigen test more robust by fdkong
○	#8209	Add missing quotes by aeslaughter

Latest 30 events

moose master : Merge commit ef4b75	Update app submodules 0:22:29	TBB 64bit Test 2:22:27	Mac Test 0:11:46	Valgrind 0:22:15	Infrastructure 0:22:10	Documentation 0:10:08 Build Docs	Test Clang 0:22:20	Test Trilinos 0:22:19	Test timings 0:04:26	Examples 0:06:31	Tutorial 0:22:24
○ moose #8209 : Add missing quotes	Precheck 0:00:32	→	Modules debug 0:33:13	Test debug 0:33:04	Pthreads Test 0:33:01	TBB Test 1:13:35	App tests 0:33:09	External App tests 0:32:59	Test Intel 0:33:07		
○ moose #8155 : Revamp automatic	Precheck	→	Modules debug	Test debug	Pthreads Test	TBB Test	App tests	External App tests			

MOOSE Publications

80+ Journal Publications, Dissertations, and Theses
Plasma-related MOOSE Publications/Dissertations:

- A. Lindsay, "Coupling of Plasmas and Liquids.", Ph.D. Dissertation, North Carolina State University, 2016.

- A. Lindsay, D. Graves and S. Shannon, "Fully coupled simulation of the plasma liquid interface and interfacial coefficient effects", Journal of Physics D: Applied Physics, vol. 49 (23),pp. 235204(1-9), May 2016.

- J. Haase, D. Go, "Designing microscale gas discharges to enhance thermionic energy conversion", Proceedings of the 2017 30th International Vacuum Nanoelectronics Conference (IVNC), July 2017.

- J. Haase, "Enhanced Thermionic Energy Conversion Using Microplasmas and Diamond Electrodes", Ph.D. Dissertation, University of Notre Dame, 2018.

Support Resources

- For general MOOSE documentation, visit:
  http://www.mooseframework.org
- For MOOSE Technical Q&A, please visit:
  https://groups.google.com/forum/!forum/moose-users
- For Zapdos (MOOSE-based low-temperature plasma fluid code) Technical Q&A, please visit:
  https://groups.google.com/forum/!forum/zapdos-users

Parallel Plate CCP Discharge: Problem Statement

Problem Overview

Consider a system containing two parallel electrodes, separated by distance L. The left electrode is powered by a voltage source of frequency f and peak voltage V_{rf} while the right is grounded. Argon gas exists between the electrodes with gas density N, pressure P, and gas temperature T_{gas}.
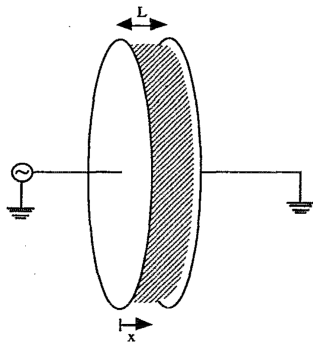
Source: Lymberopoulos and Economou, 1993

Problem Overview (cont.)

This problem loosely follows the "no metastables" case described by Lymberopoulos and Economou (1993).

D. P. Lymberopoulos and D. J. Economou, "Fluid simulations of glow discharges: Effect of metastable atoms in argon," J. Appl. Phys. **73** (8), 3668 - 3679 (1993).

Governing Equations: Electron Continuity and Momentum Balance

$$\frac{\partial n_e}{\partial t} + \nabla \cdot \mathbf{\Gamma}_e = k_i N n_e$$

and

$$\mathbf{\Gamma}_e = -D_e \nabla n_e + \mu_e n_e \nabla V$$

where:

- k_i is the reaction rate coefficient for the ground state ionization reactions in the plasma
- N is the background gas density
- V is voltage
- D_e is the electron diffusivity
- μ_e is the electron mobility

Governing Equations: Ion Continuity and Momentum Balance

$$\frac{\partial n_+}{\partial t} + \nabla \cdot \mathbf{\Gamma}_+ = k_i N n_e$$

and

$$\mathbf{\Gamma}_+ = -D_+ \nabla n_+ - \mu_+ n_+ \nabla V$$

where:

- D_+ is the electron diffusivity
- μ_+ is the electron mobility

Governing Equations: Poisson's Equation

$$\nabla^2 V = -\frac{e}{\epsilon_0}(n_+ - n_e)$$

where:

- e is the elementary charge
- ϵ_0 is the vacuum permittivity

Governing Equations: Energy Balance for Electrons

$$\frac{\partial}{\partial t}\left(\frac{3}{2}n_e T_e\right) + \nabla \cdot \mathbf{q}_e - e\mathbf{\Gamma}_e \cdot \nabla V + \sum_j H_j R_j = 0$$

$$\mathbf{q}_e = -\frac{5}{2}D_e n_e \nabla T_e + \frac{5}{2}T_e \mathbf{\Gamma}_e$$

and

$$\sum_j H_j R_j = 11.56(\text{eV})k_{ex}Nn_e + 15.7k_iNn_e$$

where:

- T_e is the electron temperature
- $k_e x$ is the reaction rate coefficient for the ground state excitation reactions in the plasma

Assuming that $T_+ = T_{gas}$, we do not need a balance equation for ions.

Getting Started

- First, download the sample code we'll be using for the Workshop.
  - https://github.com/shannon-lab/tutorial
  - In your "projects" directory (where you placed MOOSE), run
    `git clone https://github.com/shannon-lab/tutorial`
- The tutorial application also includes a `docs` directory - there you will find this presentation in PDF format!

Peacock: The MOOSE GUI

▶ Peacock allows a user to build or modify an input file, execute the application and view the results all in a single package.

▶ First, add Peacock to your system path:
  ▶ Assuming that $MOOSE_DIR$ is the location of your MOOSE directory, navigate to $MOOSE_DIR$/python/peacock
  ▶ Type `pwd` - this is the path to Peacock.
  ▶ Add the following line to your ~/.bash_profile (Mac) or ~/.bashrc (Ubuntu) file: `export PATH=pwd:\$PATH`

▶ Peacock can now be launched from any directory by simply typing `peacock` into your Terminal.

▶ To run Peacock with a particular input file, you should do
```
cd ~/projects/<your application directory>/<your input file directory>
peacock -i your_input_file.i
```

▶ See the MOOSE webpage (Application Usage section, under Documentation) for more general info

Summary of Steps

Summary of Steps Contents

Step 1: Geometry and Laplace's Equation

- The first step is to solve a simple Laplace's equation problem, which doesn't require any new code:

$$-\nabla \cdot \nabla V = 0$$

Step 2: Custom Diffusion Kernel

► In order to implement diffusion terms in the ion and electron continuity equations, a `Kernel` object is needed to represent:

$$-\nabla \cdot D\nabla n = 0$$

Step 3: Poisson's Equation and Materials

- Let's turn to Poisson's Equation.
- Instead of passing constant parameters to our CoeffDiffusion `Kernel`, we can use the `Material` system to supply the values.
  - This allows for properties that vary in space and can be coupled to variables in the simulation.

Step 4: Transient Analysis

▶ With the basis for Poisson's Equation and basic particle diffusion handled for a steady case, let's set up time dependence for our applied voltage, and discuss general Transient analysis.

Step 5: Fluid Equations

► Now that we have a handle on Custom Kernels, Boundary
  Conditions, and Steady and Transient Analysis, we now have
  enough knowledge to tackle the electron fluid equation.

Step 1: Geometry and Laplace's Equation

Step 1: Geometry and Laplace's Equation Contents

Step 1: Geometry and Laplace's Equation

- The first step is to solve a simple Laplace's equation problem, which doesn't require any code:

$$-\nabla \cdot \nabla V = 0$$

Step 1: Geometry and Laplace's Equation (cont.)

- This will have a weak form of:

$$(\nabla V, \nabla \psi_i) - \langle \nabla V \cdot \mathbf{n}, \psi_i \rangle = 0$$

- For now, ignore the boundary term (set $\nabla V \cdot \mathbf{n} = 0$)
- This is similar to the potential equation we need for Poisson's equation (as well as the diffusion terms we need in the continuity equations for ions and electrons)
- Use the `Diffusion` object that already exists in MOOSE (no new code necessary!)
- Need to specify the geometry and set up a 1D Cartesian Problem
- Further, define boundary conditions to impose the potential on each boundary
- All of this can be accomplished using the input file!

Laplace's Equation Input File

```
[Mesh]
  type = GeneratedMesh # Generates lines, rectangles, and boxes
  dim = 1 # Mesh dimension
  nx = 100 # Number of elements in the x direction
  xmax = 2.54 # (cm) Distance between the plates
[]

[Variables]
  [./potential]
    # Adds a Linear Lagrange variable by default
  [../]
[]

[Kernels]
  [./laplace]
    type = Diffusion      # A Laplacian operator
    variable = potential  # Operate on the "potential" variable
  [../]
[]
```

Laplace's Equation Input File (cont.)

```
[BCs]
  [./left]
    type = DirichletBC   # u = value BC
    variable = potential
    boundary = left # Name of a sideset from the generated mesh
    value = 100 # (V) Peak voltage from paper.
  [../]
  [./right]
    type = DirichletBC
    variable = potential
    boundary = right
    value = 0 # (V) Voltage of grounded surface.
  [../]
[]
```

Laplace's Equation Input File (cont.)

```
[Problem]
  type = FEProblem   # The "normal" type of Finite Element
                     # Problem in MOOSE
  coord_type = XYZ   # Cartesian coordinate system
[]

[Executioner]
  type = Steady # Steady state problem
  solve_type = 'PJFNK' # Preconditioned Jacobian Free
                       # Newton Krylov
  # PetSc command line options that match with the values below
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]

[Outputs]
  exodus = true # Output data in Exodus file format
[]
```

Create a Test

- Create a simple test that mimics the behavior of the problem above, but use a smaller mesh to increase the speed
- Place this test in the tests directory of your app
- See MOOSE examples in test/tests/

```
[Tests]
  [./test]
    type = 'Exodiff'
    input = 'laplace.i'
    exodiff = 'laplace_out.e'
  [../]
[]
```

Create a Test (cont.)

```
[Mesh]
  type = GeneratedMesh # Generates lines, rectangles, and boxes
  dim = 1 # Mesh dimension
  nx = 20 # Number of elements in the x direction
  xmax = 2.54 # (cm) Distance between the plates
[]

[Variables]
  [./potential]
    # Adds a Linear Lagrange variable by default
  [../]
[]

[Kernels]
  [./laplace]
    type = Diffusion       # A Laplacian operator
    variable = potential  # Operate on the "potential" variable
  [../]
[]
```

Create a Test (cont.)

```
[BCs]
  [./left]
    type = DirichletBC  # u = value BC
    variable = potential
    boundary = left # Name of a sideset from the generated mesh
    value = 100 # (V) Peak voltage from paper.
  [../]
  [./right]
    type = DirichletBC
    variable = potential
    boundary = right
    value = 0 # (V) Voltage of grounded surface.
  [../]
[]
```

Create a Test (cont.)

```
[Problem]
  type = FEProblem  # The "normal" type of Finite Element
                    # Problem in MOOSE
  coord_type = XYZ  # Cartesian coordinate system
[]

[Executioner]
  type = Steady # Steady state problem
  solve_type = 'PJFNK' # Preconditioned Jacobian Free
                       # Newton Krylov
  # PetSc command line options that match with the values below
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]

[Outputs]
  exodus = true # Output data in Exodus file format
[]
```

Finite Element Principles

Finite Element Principles Contents

Polynomial Fitting

- To introduce the idea of finding coefficients to functions, lets 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:

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

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

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

where the c_i are coefficients to be determined.

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

Example

- Define a set of points:

$$(x_1, y_1) = (1, 4)$$
$$(x_2, y_2) = (3, 1)$$
$$(x_3, y_3) = (4, 2)$$

- Substitute (x_i, y_i) data into the model:

$$y_i = a + bx_i + cx_i^2, i = 1, 2, 3$$

- Leads to the following linear system in $a, b,$ and c:

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

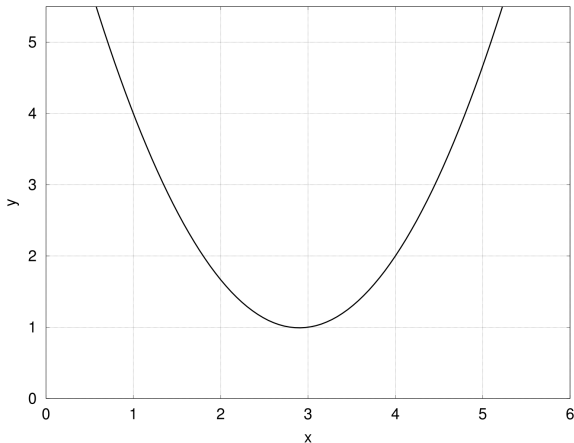# Example (cont.)

- Solving for the coefficients results in:

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 8 \\ -\frac{29}{6} \\ \frac{5}{6} \end{bmatrix}$$

- These define the solution *function*:

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

Example (cont.)

▶ Important! The solution is the function, *not* the coefficients.

Example (cont.)

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

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

  where the c_i correspond to the coefficients in the solution vector, and the g_i are the respective functions.

- ▶ Finally, note that the matrix consists of evaluating the functions at the points.

Finite Elements Simplified

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

Weak Form

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

Refresher: The Divergence Theorem

▶ Transforms a volume integral into a surface integral:

$$\int_\Omega \nabla \cdot \vec{g} \, dx = \int_{\partial\Omega} \vec{g} \cdot \hat{n} \, ds$$

▶ Slight variation: multiply by a smooth function, ψ:

$$\int_\Omega \psi(\nabla \cdot \vec{g}) \, dx = \int_\Omega \nabla \cdot (\psi\vec{g}) \, dx - \int_\Omega \nabla\psi \cdot \vec{g} \, dx$$
$$= \int_{\partial\Omega} \psi\vec{g} \cdot \hat{n} \, ds - \int_\Omega \nabla\psi \cdot \vec{g} \, dx$$

Refresher: The Divergence Theorem (cont.)

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

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

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

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

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

Example: Convection Diffusion

- ▶ Write the strong form of the equation:

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

- ▶ Rearrange to get zero on the right-hand side:

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

- ▶ Multiply by the test function ψ:

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

- ▶ Integrate over the domain Ω:

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

Example: Convection Diffusion (cont.)

▶ Apply the divergence theorem to the diffusion term:

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

▶ Write in inner product notation. Each term of the equation will inherit from an existing MOOSE type as shown below.

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

Finite Element Shape Functions

Finite Element Shape Functions Contents

Basis Function and Shape Functions

- ▶ While the weak form is essentially what you need for adding physics to MOOSE, in traditional finite element software more work is necessary.

- ▶ We need to discretize our weak form and select a set of simple basis functions amenable for manipulation by a computer.
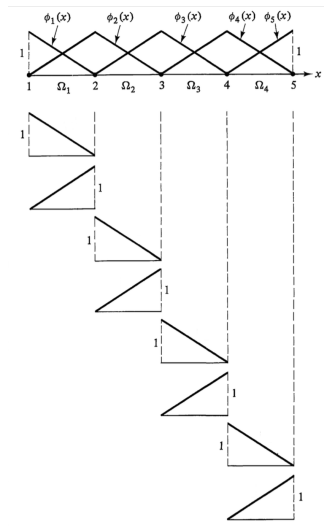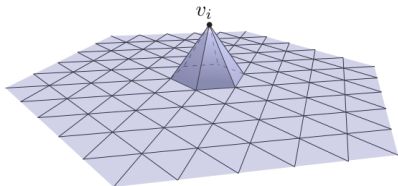
Copyright Oden, Becker, Cary 1981

Shape Functions

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

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

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

- The gradient of u can be expanded similarly:

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

Shape Functions (cont.)

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

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

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

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

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

Shape Functions (cont.)

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

Example 1D Shape Functions
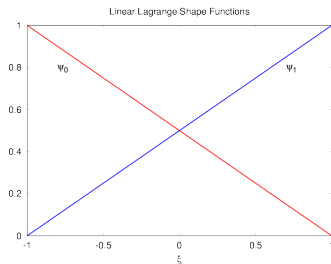
Linear Lagrange
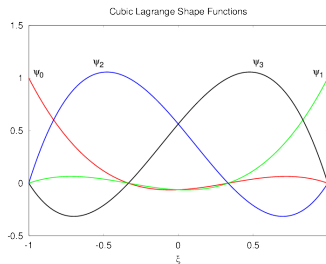
Cubic Lagrange
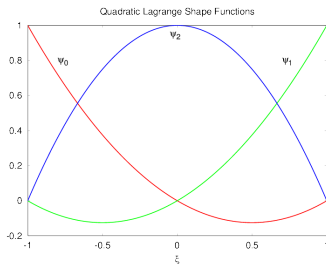
Quadratic Lagrange
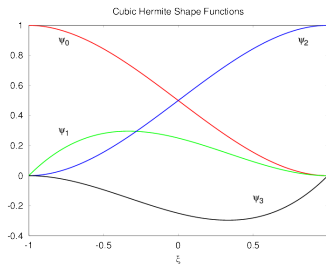
Cubic Hermite

2D Lagrange Shape Functions

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

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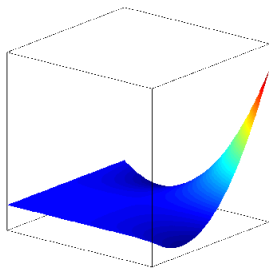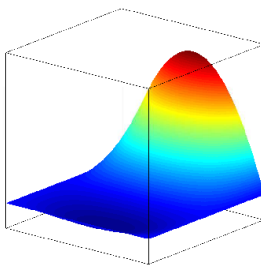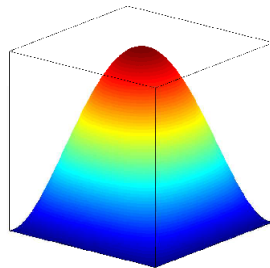
ψ_0 $\qquad\qquad\qquad$ ψ_4 $\qquad\qquad\qquad$ ψ_8

Numerical Implementation

Numerical Implementation Contents

Numerical Integration

- The only remaining non-discretized parts of the weak form are the integrals.

- We split the domain integral into a sum of integrals over elements:

$$\int_\Omega f(\vec{x})\, \mathrm{d}\vec{x} = \sum_e \int_{\Omega_e} f(\vec{x})\, \mathrm{d}\vec{x}$$

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

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

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

Numerical Integration (cont.)

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

$$\sum_e \int_{\hat{\Omega}_e} f(\vec{\xi}) |\mathbf{J}_e| d\vec{\xi} \approx \sum_e \sum_q w_q f(\vec{x}_q) |\mathbf{J}_e(\vec{x}_q)|$$

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

Numerical Integration (cont.)

- Note that sampling u_h at the quadrature points yields:

$$u(\vec{x}_q) \approx u_h(\vec{x}_q) = \sum u_j \phi_j(\vec{x}_q)$$

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

- And our weak form becomes:

$$R_i(u_h) = \sum_e \sum_q w_q |\mathbf{J}_e| \left[\nabla \psi_i \cdot k \nabla u_h + \psi_i \left(\vec{\beta} \cdot \nabla u_h \right) - \psi_i f \right] (\vec{x}_q)$$
$$- \sum_f \sum_{q_{face}} w_{q_{face}} |\mathbf{J}_f| [\psi_i k \nabla u_h \cdot \hat{n}] (\vec{x}_{q_{face}})$$

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

Newton's Method

- We now have a nonlinear system of equations,

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

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

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

- Newton's method is a "root finding" method: it finds zeros of nonlinear equations.

- Newton's Method in "Update Form" for finding roots of the scalar equation $f(x) = 0$, $f(x) : \mathbb{R} \to \mathbb{R}$ is given by

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

Newton's Method (cont.)

- We dont have just one scalar equation: we have a system of nonlinear equations.

- This leads to the following form of Newton's Method:

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

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

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

- Note that:

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

Newton for a Simple Equation

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

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

- The ith component of the residual vector is:

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

Newton for a Simple Equation

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

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

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

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

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

Chain Rule

- On the previous slide, the term $\frac{\partial f}{\partial u_j}$ was used, where f was a nonlinear forcing function.

- The chain rule allows us to write this term as

$$\frac{\partial f}{\partial u_j} = \frac{\partial f}{\partial u_h}\frac{\partial u_h}{\partial u_j}$$
$$= \frac{\partial f}{\partial u_h}\phi_j$$

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

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

Jacobian Free Newton Krylov

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

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

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

Jacobian Free Newton Krylov (cont.)

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

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

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

Jacobian Free Newton Krylov (cont.)

- This action can be approximated by:

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

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

Wrap Up

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

Step 2: Custom Diffusion Kernel

Step 2: Custom Diffusion Kernel Contents

Step 2: Custom Diffusion Kernel

- In order to implement diffusion terms in the ion and electron continuity equations, a `Kernel` object is needed to represent:

$$-\nabla \cdot D\nabla n = 0$$

Step 2: Custom Diffusion Kernel (cont.)

- The weak form of this equation is:

$$(D\nabla n, \nabla \psi_i) - \langle D\nabla n \cdot \mathbf{n}, \psi_i \rangle = 0$$

- For now, we'll ignore the boundary term (essentially setting $D\nabla n \cdot \mathbf{n} = 0$: a "no flux" boundary condition).
- The volume integral looks exactly like the `Diffusion` equation multiplied by some constants. So we can simply extend the `Diffusion` kernel by inheriting from it...

Mesh

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.

FileMesh

- ▶ `FileMesh` is the default type.
- ▶ MOOSE supports reading and writing a large number of formats and could be extended to read more.

Extension	Description
.dat	Tecplot ASCII file
.e, .exd	Sandias ExodusII format
.fro	ACDLs surface triangulation file
.gmv	LANLs GMV (General Mesh Viewer) format
.mat	Matlab triangular ASCII file (read only)
.msh	GMSH ASCII file
.n, .nem	Sandias Nemesis format
.plt	Tecplot binary file (write only)
.node, .ele, .poly	TetGen ASCII file (read; write)
.inp	Abaqus .inp format (read only)
.ucd	AVSs ASCII UCD format
.unv	I-deas Universal format
.xda, .xdr	libMesh formats
.vtk, .pvtu	Visualization Toolkit

GeneratedMesh

- `type = GeneratedMesh`
- Built-in mesh generation is implemented for lines, rectangles, and rectangular prisms ("boxes") but could be extended.
- The sides are named in a logical way and are numbered:
    - In 1D, left = 0, right = 1
    - In 2D, bottom = 0, right = 1, top = 2, left = 3
    - In 3D, back = 0, bottom = 1, right = 2, top = 3, left = 4, front = 5
- The length, width, and height of the domain, and the number of elements in each direction can be specified independently.

Named Entity Support

- 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.

Named Entity Support (cont.)

- An illustration for mesh in exodus file format:

```
[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'
[]
```

Displaced Mesh

▶ Calculations can take place in either the initial mesh configuration or, when requested, the displaced configuration.

▶ To enable displacements, provide a vector of displacement variable names for each spatial dimension in the section, e.g.:

```
displacements = disp_x disp_y disp_z
```

▶ Once enabled, the parameter 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 is a good idea to do it in code if you have a pure Langrangian formulation.

Outputs

Outputs

- The output system is designed to be just like any other system in MOOSE: modular and expandable.
- It is possible to create multiple output objects for outputting:
  - at a specific time or timestep intervals,
  - custom subsets of variables, and
  - to various file types.
- Supports common parameters and subblocks

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

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

Outputs and execute_on

Outputs can use the "execute_on" parameter like other systems in MOOSE, by default:

- ▶ All objects have execute_on = 'initial timestep_end', with Console objects being the exception.
- ▶ Console objects append 'initial timestep_begin linear nonlinear failed' to the "execute_on" settings at the common level.

When debugging output, use the --show-outputs flag when executing your application. This will add a section near the top of the simulation that shows the settings being used for each output object.

Basic Input File Syntax

- To enable output an input file must contain an [Outputs] block.
- The simplest method for enabling output is to utilize the short-cut syntax as shown below, which enables Exodus output for writing data to a file.

```
[Outputs]
  exodus = true    # output to ExodusII file with default settings
[]
```

Output Execution

- In similar fashion to many other systems in MOOSE it is possible to control when output occurs via the "execute_on" parameter.
- By default `execute_on = initial timestep_end`
- A convenience parameter, "additional_execute_on", allows appending flags to the existing "execute_on" flags.
- The toggles shown in Output Execution Toggles operate by appending to the "execute_on" flags.
- Currently, the following output execution times are available:

Text Flag	Description
"initial"	executes the output on the initial condition (on by default)
"linear"	executes the output on each linear iteration
"nonlinear"	executes the output on each non-linear iteration
"timestep_end"	calls the output method at the end of the timestep (on by default)
"timestep_begin"	executes the output method at the beginning of the timestep
"final"	calls the output method on the final timestep
"failed"	executes the output method when the solution fails

Output Execution (cont.)

- There are two types of outputs classes: `BasicOutput` and `AdvancedOutput`.
- Advanced outputs have additional control beyond `execute_on`, as detailed in the table below.

Input Parameter	Method Controlled
execute_postprocessors_on	`outputPostprocessors`
execute_vector_postprocessors_on	`outputVectorPostprocessors`
execute_elemental_on	`outputElementalVariables`
execute_nodal_on	`outputNodalVariables`
execute_scalar_on	`outputScalarVariables`
execute_system_information_on	`outputSystemInformation`
execute_input_on	`outputInput`

Output Execution (cont.)

- Each of the output controls accept the same output execution flags that execute_on utilizes
- In AdvancedOutput objects, the execute_on settings are used to set the defaults for each of the output type controls.
    - For example, setting
      execute_on = 'initial timestep_end' causes all of the output types (e.g., postprocessors, scalars, etc.) to execute on each timestep and the initial condition.

File Output Names

▶ The default naming scheme for output files utilizes the input file name (e.g., input.i) with a suffix that differs depending on how the output is defined:
  ▶ An "_out" suffix is used for Outputs created using the short-cut syntax.
  ▶ Sub-blocks use the actual sub-block name as the suffix.

▶ For example, if the input file (input.i) contained the following [Outputs] block, two files would be created: input_out.e and input_other.e.

```
[Outputs]
  console = true
  exodus = true      # creates input_out.e
  [./other]          # creates input_other.e
    type = Exodus
    interval = 2
  [../]
[]
```

▶ Note, the use of file_base anywhere in the [Outputs] block disables all default naming behavior.

Supported Types and Syntax

Format	Short-cut	Sub-block Type	C++ Object	Comments
Console (screen) output	`console`	`Console`	`Console`	Writes to the screen and optionally a file
Exodus II (recommended)	`exodus`	`Exodus`	`Exodus`	The most common, well-supported, and controllable output type
VTK	`vtk`	`VTK`	`VTKOutput`	Visual Analysis Toolkit format, requires --enable-vtk when building libMesh
GMV	`gmv`	`GMV`	`GMVOutput`	General Mesh Viewer format
Tecplot	`tecplot`	`Tecplot`	`Tecplot`	Support is limited, requires -enable-tecplot when building libMesh
XDA	`xda`	`XDA`	`XDA`	libMesh internal format (ASCII)
XDR	`xdr`	`XDR`	`XDR`	libMesh internal format(binary)
CSV	`csv`	`CSV`	`CSV`	Comma separated scalar values
GNUplot	`gnuplot`	`GNUPlot`	`GNUPlot`	Only supports scalar outputs
Checkpoint	`checkpoint`	`Checkpoint`	`Checkpoint`	MOOSE internal format used for restart and recovery
Solution History	`solution_history`	`SolutionHistory`	`SolutionHistory`	MOOSE internal format used for writing solution history

Coordinate Systems

Axisymmetric Coordinates

- MOOSE supports axisymmetric coordinates
    - i.e., cylindrical coordinates (r, θ, z), where the problem is symmetrical in the θ direction.
- The axis of rotation is controlled using:

```
[Problem]
  type = FEProblem
  coord_type = RZ
  rx_coord_axis = X
[]
```

- MOOSE assumes the problem is symmetrical about the specified axis and automatically applies the proper coordinate transformation.
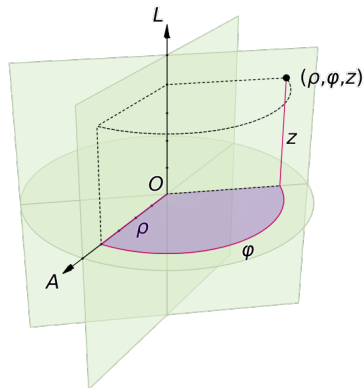
Spherical Coordinates

- ▶ MOOSE supports symmetrical spherical coordinates
  - ▶ i.e., spherical coordinates (r, θ, ϕ), where the problem is symmetrical in the θ and ϕ directions.
- ▶ Enable spherical coordinates using the following input file syntax:

```
[Problem]
  type = FEProblem
  coord_type = RSpherical
[]
```

Anatomy of a MOOSE Object

MooseObject Basics

- All objects in MOOSE inherit from `MooseObject`
- Therefore, every object that is created by a user to describe their problem has the same basic structure.
- This basic structure is given in the following example. (*Note, this is not valid MOOSE code.*)

MyObject.h

```
#ifndef MYOBJECT_H
#define MYOBJECT_H

#include "BaseClass.h"

class MyObject;

template <>
InputParameters validParams<MyObject>();

class MyObject : public BaseClass
{
public:
  MyObject(const InputParameters & parameters);

protected:
  virtual SomeType inheritedMethod();
};

#endif // MYOBJECT_H
```

MyObject.C

```
#include "MyObject.h

registerMooseObject("AnimalApp", MyObject);

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

MyObject::MyObject(const InputParameters & parameters):
  BaseClass(parameters)
{}

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

Input Parameters and MOOSE Types

Valid Parameters

- A set of custom parameters is used to construct every object.
- Every MOOSE-derived object must specify a `validParams` function.
- In this function you *must* start with the parameters from your parent class (e.g., 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: Mouse-over when using the MOOSE GUI "peacock"
- Option 2: 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 3: Generate a tree based on your input file
  - CLI option `--show-input`
- Option 4: View it online
  - http://mooseframework.org/old/wiki/InputSyntax

Valid Types

- 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(...)`

Valid Types (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`
    - ► `VariableName`
    - ► `FunctionName`
    - ► `UserObjectName`
    - ► `PostprocessorName`
    - ► `IndicatorName`
    - ► `MarkerName`
    - ► `MeshFileName`
    - ► `OutFileName`
    - ► `NonlinearVariableName`
    - ► `AuxVariableName`
- ► For a complete list, see the instantiations at the bottom of framework/include/utils/MooseTypes.h.

Default and Range 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 C++ objects:
  - 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 space 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 have saved our MooseEnum into an instance variable: _component
Real value = 0.0;
if (_component.isValid())
  value = _some_vector[_component];
```

▶ The Peacock GUI will create a drop box when using
  `MooseEnum`.

▶ If the user supplies an invalid option, the parser will catch it
  and throw an informative error message.

Multiple Value MooseEnums (MultiMooseEnum)

- Works the same way as `MooseEnum` but supports multiple ordered options.

```
template<>
InputParameters validParams<MyObject>()
{
  InputParameters params = validParams<ParentObject>();
  MultiMooseEnum transforms("scale rotate translate");
  params.addRequiredParam<MultiMooseEnum>("transforms", transforms,
                                          "The transforms to perform");
  return params;
}

...

if (_transforms.isValid())
for (unsigned int i=0; i<_transforms.size(); ++i)
  performTransform(transforms[i]);
```

- Can also ask if item is stored in the `MultiMooseEnum` by calling `_transforms.contains(item);`

Kernels

Overview

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

Kernel Member Variables (some)

- `_u, _grad_u`
  - Value and gradient of the 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 at 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.

Diffusion

The strong form of the diffusion equation is defined as:

$$-\nabla \cdot \nabla u - f = 0$$
$$u|_{\partial\Omega_1} = g_1$$
$$\nabla u \cdot \hat{n}|_{\partial\Omega_2} = g_2$$

Generate the weak form by:

- Multiplying by the test function and integrate over the domain:

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

- Integrating by parts results in the weak form:

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

The Jacobian is defined as:

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

Diffusion.h

```cpp
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef DIFFUSION_H
#define DIFFUSION_H

#include "Kernel.h

class Diffusion;

template <>
InputParameters validParams<Diffusion>();

/**
 * This kernel implements the Laplacian operator:
 * $\nabla u \cdot \nabla \phi_i$
 */
class Diffusion : public Kernel
{
public:
  Diffusion(const InputParameters & parameters);

protected:
  virtual Real computeQpResidual() override;

  virtual Real computeQpJacobian() override;
};

#endif /* DIFFUSION_H */
```

Diffusion.C

```cpp
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "Diffusion.h

registerMooseObject("MooseApp", Diffusion);

template <>
InputParameters
validParams<Diffusion>()
{
  InputParameters params = validParams<Kernel>();
  params.addClassDescription("The Laplacian operator ($-\\nabla \\cdot \\nabla u$), with
                             "the weak form of $(\\nabla \\phi_i, \\nabla u_h)$.");
  return params;
}

Diffusion::Diffusion(const InputParameters & parameters) : Kernel(parameters) {}

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

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

Kernel Registration

- Before a `Kernel` is available for use, it must be registered. This is done right in the source file (e.g. Diffusion.C).

    ```
    #include "Diffusion.h"

    registerMooseObject("MooseApp", Diffusion);
    ```

Step 2 Source Code

CoeffDiffusion.h

```
#ifndef COEFFDIFFUSION_H
#define COEFFDIFFUSION_H

// Including the "Diffusion" Kernel here so we can extend it
#include "Diffusion.h"

class CoeffDiffusion;

template<>
InputParameters validParams<CoeffDiffusion>();

/**
 * Computes the residual contribution: D * grad_u * grad_phi
 */
class CoeffDiffusion : public Diffusion
{
public:
  CoeffDiffusion(const InputParameters & parameters);
```

Step 2 Source Code (cont.)

CoeffDiffusion.h (cont.)

```
protected:
  // Kernels _must_ override computeQpResidual()
  virtual Real computeQpResidual() override;
  // This is optional, but recommended!
  virtual Real computeQpJacobian() override;

  // Will be set from the input file
  Real _diffusivity;
};

#endif // COEFFDIFFUSION_H
```

Step 2 Source Code (cont.)

CoeffDiffusion.C

```
#include "CoeffDiffusion.h"

registerMooseObject("TutorialApp", CoeffDiffusion);

template<>
InputParameters validParams<CoeffDiffusion>()
{
  // Start with the parameters from our parent kernel
  InputParameters params = validParams<Diffusion>();

  // Now add any extra parameters that this class needs:

  // Add a required parameter. If this isn't provided in the
  // input file MOOSE will throw an error.
  params.addRequiredParam<Real>("diffusivity",
                                "Particle diffusivity.");
  return params;
}
```

Step 2 Source Code (cont.)

CoeffDiffusion.C (cont.)

```
CoeffDiffusion::CoeffDiffusion(const InputParameters & parameters)
  : Diffusion(parameters),
  // Get the parameters from the input file
  _diffusivity(getParam<Real>("diffusivity"))
{
}

Real
CoeffDiffusion::computeQpResidual()
{
  return _diffusivity * Diffusion::computeQpResidual();
}

Real
CoeffDiffusion::computeQpJacobian()
{
  return _diffusivity * Diffusion::computeQpJacobian();
}
```

Step 2 Input File

```
[Mesh]
  type = GeneratedMesh
  dim = 1
  nx = 100
  xmax = 2.54
[]

[Variables]
  [./n]
  [../]
[]

[Kernels]
  [./laplace_coeff]
    type = CoeffDiffusion
    variable = n
    diffusivity = 64.29  # Ion diffusivity from paper
  [../]
[]
```

Step 2 Input File (cont.)

```
[BCs]
  [./left]
    type = DirichletBC
    variable = n
    boundary = left
    value = 1e4
  [../]
  [./right]
    type = DirichletBC
    variable = n
    boundary = right
    value = 0
  [../]
[]
```

Step 2 Input File (cont.)

```
[Problem]
  type = FEProblem
  coord_type = XYZ
[]

[Executioner]
  type = Steady
  solve_type = 'PJFNK'
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]

[Outputs]
  exodus = true
[]
```

Create a Test

- Create a simple test that mimics the behavior of the problem above, i.e., a test that verifies that the `CoeffDiffusion` kernel is operating as expected.

- In this case, we can take the input file on the previous slides and use that as the basis for the test.

```
[Tests]
  [./test]
    type = Exodiff
    input = 'diffusion.i'
    exodiff = 'diffusion_out.i'
  [../]
[]
```

Creating an Application

- To begin, navigate to your projects directory (outside of the MOOSE directory) and run the stork.sh script in the `scripts` directory inside of MOOSE.

- So you would type into your Terminal:

  ```
  cd ~/projects
  ./moose/scripts/stork.sh YourAppName
  ```

- Running the script will create a directory called "YourAppName" in the projects directory, and it will be automatically linked to MOOSE.

- Treaditionally MOOSE application names have been animals, mostly acronyms, but you can name your app whatever suits your needs!

- NOTE: Do **not** try to run this script while inside the MOOSE directories. This will result in an error!

Reaction-Diffusion Equation

Reaction-Diffusion Equation Contents

Problem Statement

Given a domain Ω, find u such that:

$$-\nabla \cdot (D(u)\nabla u) + bu = 0 \in \Omega,$$

and

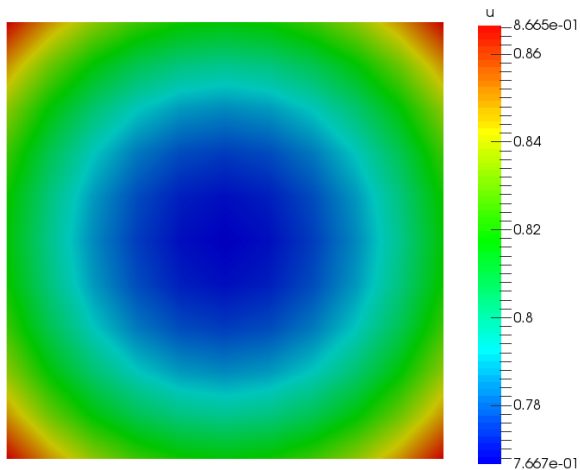$$D(u)\nabla u \cdot \hat{n} = \sigma \in \Omega,$$

where

$$D(u) \equiv \frac{1}{\sqrt{1 + |\nabla u|^2}}$$
$$b = 1$$
$$\sigma = 0.2$$

Results

Step 3: Poisson's Equation and Materials

Step 3: Poisson's Equation and Materials Contents

Step 3: Poisson's Equation and Materials

- Let's turn to Poisson's Equation.
- Instead of passing constant parameters to our CoeffDiffusion `Kernel`, we can use the `Material` system to supply the values.
  - This allows for properties that vary in space and can be coupled to variables in the simulation.

Materials

Materials

- The material system operates by creating a producer/consumer relationship among objects.
  - `Material` objects produce properties.
  - Other MOOSE objects (including Materials) consume these properties.

- `Material` objects produce properties:
  - Each property must be declared to be available for use by Kernels, boundary conditions, etc.
  - `declareProperty<TYPE>()` declares a material property, and returns a writable reference.
  - Override `computeQpProperties()` to compute all of the declared properties at one quadrature point. Within this method, the references obtained from `declareProperty` are updated.
  - Can use coupled variables in the same way as Kernels (e.g., `coupledValue()`, etc.).

Materials (cont.)

- Objects consume material properties:
  - To use a material property, call `getMaterialProperty<TYPE>()` in a `Kernel` or other object, and store the constant reference.
- Quantities are recomputed at quadrature points, as needed.
- The values are not stored between timesteps unless "stateful" properties are enabled.
  - Call `declarePropertyOld<TYPE>()` or `declarePropertyOlder<TYPE>()` to enable stateful material properties.
- Multiple `Material` objects may define the same "property" for different parts of the subdomain or boundaries.

Default Material Properties

▶ Default values for material properties may be assigned within the validParams function.

```
addParam<MaterialPropertyName>("combination_property_name", 12345,
"The name of the property providing the luggage combination");
```

▶ Only scalar (Real) values may have defaults.

▶ When
getMaterialProperty<Real>("combination_property_name")
is called, the default will be returned if the value has not been computed via a declareProperty call within a Material object.

Automatic Material Property Output

- Material properties can be output, provided they are one of the supported types.
- Output of Material properties is enabled by setting the "outputs" parameter.
- The following example creates two additional variables called "mat1" and "mat2" that will show up in the output file.

```
[Materials]
  [./generic]
    type = GenericConstantMaterial
    block = 0
    prop_names  = 'mat1 mat2'
    prop_values = '1 2'
    outputs = exodus
  [../]
[]
[Outputs]
  console = true
  exodus = true
  [./oversample]
    type = Exodus
    oversample = true
    refinements = 2
  [../]
[]
```

Automatic Material Property Output

- A list of possible options for the 'outputs' parameter:

'outputs ='	Behavior
none	disables outputting of all properties to all possible outputs (default)
all	enables outputting of all properties to all possible outputs
exodus	enables outputting of all properties to the output named "exodus" from the [Outputs] block

- Supported types include: `Real`, `RealVectorValue`, and `RealTensorValue`

Automatic Material Property Output

- It is possible to limit which properties are written to the output for each `Material` using the "output_properties" parameter.
- For example, setting `output_properties = mat1` from the previous example ensures that only "mat1" is written.
- Leaving this property empty will result in all properties being written (the default).
- For additional examples, see the associated tests in `$MOOSE_DIR$/test/tests/materials/output`.

Output via the Outputs Block

- ▶ The following input file section writes the `Material` property named 'mat1'.
- ▶ Notice that this method does not require changing anything within the `Material` blocks, which is useful if a material is defined on many subdomains and/or boundaries.
- ▶ The `show_material_properties` parameter is optional. Without it, all supported `Material` properties will be written.

```
[Materials]
  [./generic]
    type = GenericConstantMaterial
    block = 0
    prop_names  = 'mat1 mat2'
    prop_values = '1 2'
  [../]
[]
[Outputs]
  console = true
  [./out]
    type = Exodus
    output_material_properties = true
    show_material_properties = 'mat1'
  [../]
[]
```

Supported Output Types

- `Material` properties can be of arbitrary (C++) type, but not all types can be output.
- The following is a list of the `Material` property types that support automatic output. (Assume you named your property "prop" when calling `declareProperty`.)
- Note that each AuxKernel name can be clicked on to show the Doxygen documentation webpage for each one.

Type	AuxKernel	Variable Name(s)
Real	MaterialRealAux	prop
RealVectorValue	MaterialRealVectorValueAux	prop_1, prop_2, prop_3
RealTensorValue	MaterialRealTensorValueAux	prop_11, prop_12, prop_13, prop_21, etc.

Stateful Material Properties

- It can sometimes be useful to have "old" values of `Material` properties available in a simulation.
- For example, this situation arises in solid mechanics plasticity constituitive models.
- Traditionally, this type of value is called a "state variable".
- In MOOSE, they are called "Stateful Material Properties".
- To provide a material property with an old state, use `declarePropertyOld<TYPE>()` and `declarePropertyOlder<TYPE>()`.
- Stateful material properties require more memory.

Poisson's Equation Material and Properties

- For this Poisson example, we need one `Material` property: `permittivity`.
- This can be computed via one `Material` object that we'll call `ExampleMaterial`.
- Though for this scenario the relative permittivity will remain constant, this method could also apply to placing a spatially varying permittivity or some other important plasma parameter that can be thought of as a "material" (like reaction rate coefficients!).

Source Code (ExampleMaterial Header File)

```cpp
#ifndef EXAMPLEMATERIAL_H
#define EXAMPLEMATERIAL_H

#include "Material.h"

class ExampleMaterial;

template <>
InputParameters validParams<ExampleMaterial>();

/**
 * An example material class for providing a derived Diffusion
 * kernel class with a diffusiity parameter
 */
class ExampleMaterial : public Material
{
public:
  ExampleMaterial(const InputParameters & parameters);

protected:
  virtual void computeQpProperties() override;

  MaterialProperty<Real> & _diffusivity;
};

#endif // EXAMPLEMATERIAL_H
```

Source Code (ExampleMaterial C File)

```
#include "ExampleMaterial.h"

registerMooseObject("TutorialApp", ExampleMaterial);

template <>
InputParameters
validParams<ExampleMaterial>()
{
  InputParameters params = validParams<Material>();
  return params;
}

ExampleMaterial::ExampleMaterial(const InputParameters & parameters)
  : Material(parameters), _diffusivity(declareProperty<Real>("diffusivity"))
{
}

void
ExampleMaterial::computeQpProperties()
{
  // Relative permittivity
  _diffusivity[_qp] = 1.01;
}
```

Source Code (ExampleMatDiffusion Header File)

```
#ifndef EXAMPLEMATDIFFUSION_H
#define EXAMPLEMATDIFFUSION_H

// Including the "Diffusion" Kernel here so we can extend it
#include "Diffusion.h"

class ExampleMatDiffusion;

template <>
InputParameters validParams<ExampleMatDiffusion>();

/**
 * Computes the residual contribution: D * grad_u * grad_phi, where
 * D is defined via a material.
 */
```

Source Code (ExampleMatDiffusion Header File) (cont.)

```
class ExampleMatDiffusion : public Diffusion
{
public:
  ExampleMatDiffusion(const InputParameters & parameters);

protected:
  // Kernels _must_ override computeQpResidual()
  virtual Real computeQpResidual() override;
  // This is optional, but recommended!
  virtual Real computeQpJacobian() override;

  // Will be set from active material
  const MaterialProperty<Real> & _diffusivity;
};

#endif // EXAMPLEMATDIFFUSION_H
```

Source Code (ExampleMatDiffusion C File)

```
#include "ExampleMatDiffusion.h"

registerMooseObject("TutorialApp", ExampleMatDiffusion);

template <>
InputParameters
validParams<ExampleMatDiffusion>()
{
  // Start with the parameters from our parent kernel
  InputParameters params = validParams<Diffusion>();
  // This doesn't need any more parameters, since diffusivity
  // is being called from a material.
  return params;
}

ExampleMatDiffusion::ExampleMatDiffusion(const InputParameters & parameters)
  : Diffusion(parameters),

    // Get the parameters from the active material
    _diffusivity(getMaterialProperty<Real>("diffusivity"))
{
}
```

Source Code (ExampleMatDiffusion C File) (cont.)

```
Real
ExampleMatDiffusion::computeQpResidual()
{
  // D * grad_u * grad_phi[i]
  return _diffusivity[_qp] * Diffusion::computeQpResidual();
}

Real
ExampleMatDiffusion::computeQpJacobian()
{
  // D * grad_phi[j] * grad_phi[i]
  return _diffusivity[_qp] * Diffusion::computeQpJacobian();
}
```

Input File

```
[Mesh]
  type = GeneratedMesh
  dim = 1
  nx = 100
  xmax = 0.0254
[]

[Variables]
  [./V]
  [../]
[]

[Kernels]
  [./poisson]
    type = ExampleMatDiffusion
    variable = V
    # permittivity (eps_R = 1.01) will be taken from a material
  [../]
  [./RHS]
    type = BodyForce
    variable = V
    function = 9.05e6 # from (e / eps0)*(5e15 - 4.5e15)
  [../]
[]
```

Input File (cont.)

```
[BCs]
  [./left]
    type = DirichletBC
    variable = V
    boundary = left
    value = 100
  [../]
  [./right]
    type = DirichletBC
    variable = V
    boundary = right
    value = 0
  [../]
[]

[Materials]
  [./example_material]
    type = ExampleMaterial
  [../]
[]
```

Input File (cont.)

```
[Problem]
  type = FEProblem
  coord_type = XYZ
[]

[Executioner]
  type = Steady
  solve_type = 'PJFNK'
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]

[Outputs]
  exodus = true
[]
```

Create a Test

- A simple test can be made from the input file we've created.

```
[Tests]
  [./test]
    type = 'Exodiff'
    input = 'step03.i'
    exodiff = 'step03_out.e'
  [../]
[]
```

Step 4: Transient Analysis

Step 4: Transient Analysis Contents I

Step 4: Transient Analysis Contents II

Steady and Transient Analysis

► With the basis for Poisson's Equation and basic particle diffusion handled for a steady case, let's set up time dependence for our applied voltage, and discuss general Transient analysis.

Transient Poisson

- To add time dependence to Poisson's equation, we will first add a time-varying sinusoidal function to represent our applied voltage.

$$V(t) = V_{peak} \sin(2\pi f t)$$

  where $f = 13.56 MHz$.

- Then we'll modify the Executioner to perform a Transient solve.

Executioners

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 adaptively refining the mesh to improve the solution.

Common Executioner Options

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

Option	Definition
l_tol	Linear Tolerance (default: 1e-5)
l_max_its	Max Linear Iterations (default: 10000)
nl_rel_tol	Nonlinear Relative Tolerance (default: 1e-8)
nl_abs_tol	Nonlinear Absolute Tolerance (default: 1e-50)
nl_max_its	Max Nonlinear Iterations (default: 50)

Transient Executioners

- Transient executioners solve the nonlinear system at least once per time step.
- Some frequently-used Transient Executioner options are:

Option	Definition
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 Executioners (cont.)

Advanced Transient Executioner Options:

Option	Definition
`steady_state_detection`	Whether to try and detect achievement of steady-state (Default = false)
`steady_state_tolerance`	Used for determining steady-state; Compared against the difference in solution vectors between current and old time steps (Default = 1e-8)

- ▶ Note that if you have things like piecewise-constant transient forcing functions, you may achieve temporary steady-states and may not want to terminate the simulation at those points.
- ▶ In this case a better solution may be to set an `nl_abs_tol` which will allow your simulation to converge in 0 non-linear iterations during periodic steady-states, but will also allow your simulation to solve as usual when your forcing functions are perturbed.

Transient Analysis

- MOOSE provides the following implicit `TimeIntegrators`:
  - Backward Euler (default)
  - BDF2
  - Crank-Nicolson
  - Implicit-Euler
  - Implicit Midpoint (implemented as two-stage RK method)
  - Diagonally-Implicit Runge-Kutta (DIRK) methods of order 2 and 3.
- And these explicit `TimeIntegrators`:
  - Explicit Euler
  - Various two-stage explicit Runge-Kutta methods (Midpoint, Heun, Ralston, TVD)
- Each one of these supports adaptive time stepping.
- They are all accessed via the `TimeDerivative` Kernel object.

Time Kernels

Adding Time Dependence via Time Derivatives

- ▶ While this example focuses on time-varying BCs and functions, adding time dependence to a steady diffusion equation (like our reaction-diffusion example from earlier) can be as easy as adding a time derivative term.
- ▶ Let's discuss the `TimeDerivative|` Kernel as well as general Time Kernels.

TimeDerivative

The residual contribution for the time derivative term is

$$\left(\frac{\partial u_h}{\partial t}, \psi_i \right)$$

where u_h is the finite element solution, and

$$\frac{\partial u_h}{\partial t} \equiv \frac{\partial}{\partial t} \left(\sum_k u_k \phi_k \right) = \sum_k \frac{\partial u_k}{\partial t} \phi_k$$

because you can interchange the order of differentiation and summation. Call this equation (1).

TimeDerivative (cont.)

In the equation on the previous slide, $\frac{\partial u_k}{\partial t}$ is the time derivative of the kth finite element coefficient of u_h. While the exact form of this derivative depends on the time stepping scheme, without much loss of generality, we can assume the following form for the time derivative:

$$\frac{\partial u_k}{\partial t} = a u_k + b$$

for some constants a, b which depend on δt and the timestepping method.

TimeDerivative (cont.)

The derivative of (1) with respect to u_j is then:

$$\frac{\partial}{\partial u_j} \left(\sum_k \frac{\partial u_k}{\partial t} \phi_k \right) = \frac{\partial}{\partial u_j} \left(\sum_k (au_k + b)\phi_k \right) = a\phi_j$$

So that the Jacobian term for (1) is

$$(a\phi_j, \psi_i)$$

where a is what we call `du_dot_du` in MOOSE.

Therefore the `computeQpResidual()` function in the
`TimeDerivative` Kernel in MOOSE looks like:

```
return _test[_i][_qp] * _u_dot[_qp];
```
And the corresponding `computeQpJacobian()` is:
```
return _test[_i][_qp] * _phi[_j][_qp] * _du_dot_du[_qp];
```

Custom Time Kernel

- If you need to provide a coefficient for the transient term, inherit from the `TimeDerivative` Kernel in MOOSE.
- Then, in `computeQpResidual/Jacobian()` return the product of your coefficient and `TimeDerivative::computeQpResidual/Jacobian()`.

A Quick Note

The following code can be found in the MOOSE repository, under
`examples/ex06_transient/`

Custom Time Kernel (cont.)

```cpp
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "ExampleTimeDerivative.h"

#include "Material.h"

registerMooseObject("ExampleApp", ExampleTimeDerivative);

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

Custom Time Kernel (cont.)

```
ExampleTimeDerivative::ExampleTimeDerivative(const InputParameters & parameters
  : TimeDerivative(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();
}
```

Summary of input file syntax changes for transient analysis

- Add appropriate `TimeDerivative` Kernels
- Switch your executioner type to "Transient"
- Add a few typical parameters to the Executioner block
  - Real `time`: The start time for the simulation
  - Real `dt`: The initial time step size
  - string `scheme`: Time integrator scheme
    - `'crank-nicolson'`
    - `'backward-euler'` – default if you do not specify the scheme
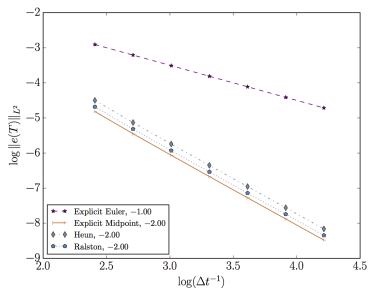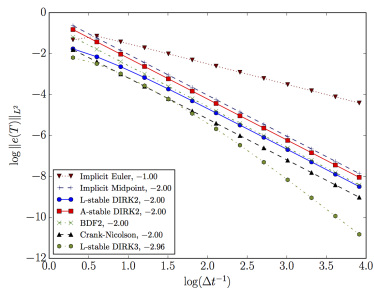   - `'bdf2'`

Convergence Rates

- Consider the test problem:

$$\frac{\partial u}{\partial t} - \nabla^2 u = f$$
$$u(t = 0) = u_0$$
$$u|_{\partial \Omega} = u_D$$

  for $t = (0, T]$, and $\Omega = (-1, 1)^2$

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

- u_0 and u_D are the initial and Dirichlet boundary conditions corresponding to this exact solution.

Convergence Rates (cont.)

TimeSteppers

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`
  - `SolutionAdaptiveDT`
  - `IterationAdaptiveDT`
  - `FunctionDT`
  - `PostprocessorDT`
  - `DT2`
  - `TimeSequenceStepper`
- MOOSE 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.

DT2 Adaptive TimeStepper

- Take one time step of size Δt to get $\hat{u}_{n+1}$ from u_n
- Take two time steps of size $\frac{\Delta t}{2}$ 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 $e_n \equiv \frac{\hat{e}_n}{\Delta t}$
- Adaptivity is based on target error tolerance e_{TOL} and a maximum acceptable error tolerance e_{MAX}.
  - If $e_n < e_{MAX}$, continue with a new time step size

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

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

  - If $e_n \geq e_{MAX}$, or if the solver fails, shrink Δt.
  - Parameters e_{TOL} and e_{MAX} can be specified in the input file as `e_tol` and `e_max` (in the `Executioner` block).

TimeSequenceStepper

- Provide a vector of time points using parameter `time_sequence`.
- `TimeSequenceStepper` simply moves through these time points.
- The time stepper automatically adds t_{start} and t_{end} to the sequence.
- Only time points satisfying $t_{start} < t < t_{end}$ are considered.
- If a solve fails at step n an additional time point $t_{new} = \frac{1}{2}(t_{n+1} + t_n)$ is inserted and the step is resolved.

Transient Code

- Back to our original example, we can add our time-varying applied voltage with a `ParsedFunction` and a `FunctionDirichletBC` on our biased boundary (we'll talk more about functions in Step 5).

Input File

```
[Mesh]
  type = GeneratedMesh
  dim = 1
  nx = 100
  xmax = 0.0254
[]

[Variables]
  [./V]
  [../]
[]

[Kernels]
  [./poisson]
    type = ExampleMatDiffusion
    variable = V
    # permittivity (eps_R = 1.01) will be taken from a material
  [../]
  [./RHS]
    type = BodyForce
    variable = V
    function = 9.05e6 # from (e / eps0)*(5e15 - 4.5e15)
  [../]
[]
```

Input File (cont.)

```
[BCs]
  [./left]
    type = FunctionDirichletBC
    variable = V
    boundary = left
    function = voltage_wave
  [../]
  [./right]
    type = DirichletBC
    variable = V
    boundary = right
    value = 0
  [../]
[]

[Materials]
  [./example_material]
    type = ExampleMaterial
  [../]
[]
```

Input File (cont.)

```
[Functions]
  [./voltage_wave]
    type = ParsedFunction
    value = '100*sin(2*pi*13.56e6*t)'
  [../]
[]

[Problem]
  type = FEProblem
  coord_type = XYZ
[]

[Executioner]
  type = Transient
  dt = 7.37464e-09 # 10 timesteps per period
  end_time = 7.37464e-07 # 10 periods
  nl_rel_tol = 1e-07
  solve_type = 'PJFNK'
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]

[Outputs]
  exodus = true
[]
```

Boundary Conditions

- Since we used a new boundary condition in this example, let's take some time to discuss the Boundary Condition system in MOOSE.

Boundary Conditions

BoundaryCondition

- A `BoundaryCondition` provides a residual (and optionally a Jacobian) on a boundary (or internal 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).
- BCs which are integrated over the boundary 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
- ▶ `_test`, `_grad_test`: Value (ψ) and gradient ($\nabla\psi$) of the test functions
- ▶ `_q_point`: XYZ coordinates
- ▶ `_i`, `_j`: test and trial shape function indices
- ▶ `_qp`: Current quadrature point index.
- ▶ `_normals`: Normal vector
- ▶ `_boundary_id`: The boundary ID
- ▶ `_current_elem`: A pointer to the element
- ▶ `_current_side`: The side number of `_current_elem`

Non-integrated BCs:

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

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.

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

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:

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

Integrated BCs

Integrated BCs (including Neumann BCs) are actually integrated over the external face of an element.

Their residuals look similar to kernels. Thus

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

becomes:

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

▶ Also note that if $\nabla u \cdot \hat{\boldsymbol{n}} = 0$, then the boundary integral is zero (sometimes known as the "natural boundary condition").

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 (cont.)

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

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

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

Step 5: Fluid Equations

Step 5: Fluid Equations Contents I

Step 5: Fluid Equations

▶ Now that we have a handle on Custom Kernels and Steady and Transient Analysis, we now have enough knowledge to tackle the electron fluid equation.

Fluid Equations: What do we need?

- In order to simulate fluid equations for the ions and electrons, we need to use or build a few things:

    - a time derivative term (`TimeDerivative`);

    - a particle diffusion term (`CoeffDiffusion` or `ExampleMatDiffusion` if we wanted to use a material, but lets stick with `CoeffDiffusion` for now);

    - a field advection term (let's create this one from scratch);

    - reaction terms (we could use `Reaction` in MOOSE, but we would need to modify it in order to use a reaction rate coefficient constant or function.)

- All of this we've seen examples of so far!

- We will then be "manufacturing" a solution to properly test this functionality.

Fluid Equation for Electrons

The fluid equation for ions and electrons are identical in our model. So, let's focus on the electron continuity and momentum balance for the example in this step. We have:

$$\frac{\partial n_e}{\partial t} + \nabla \cdot \mathbf{\Gamma}_e = k_i N n_e$$

and

$$\mathbf{\Gamma}_e = -D_e \nabla n_e + \mu_e n_e \nabla V$$

where:

- k_i is the reaction rate coefficient for the ground state ionization reactions in the plasma

- N is the background gas density

- V is voltage

- D_e is the electron diffusivity

- μ_e is the electron mobility

Custom Kernel: Field Advection

- To add the field advection term, $\nabla \cdot (\mu_e n_e \nabla V)$, we need to add a coupled kernel.

- The residual contribution for this term would be $\nabla \psi_i \mu_e n_e \nabla V$

- Since the sign for this term changes based on the charge of the species, we'll add a "sign" parameter for the user to easily change the sign of the term and allow the same code to be re-used.

Field Advection Header File

```
#ifndef EFIELDADVECTION_H
#define EFIELDADVECTION_H

#include "Kernel.h"

class EFieldAdvection;

template <>
InputParameters validParams<EFieldAdvection>();

/**
 *
 */
class EFieldAdvection : public Kernel
{
public:
  EFieldAdvection(const InputParameters & parameters);
```

Field Advection Header File (cont.)

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

private:
  const VariableGradient & _grad_potential;

  Real _mobility;

  Real _sign;

  unsigned int _potential_id;
};

#endif // EFIELDADVECTION_H
```

Field Advection C File

```
#include "EFieldAdvection.h"

registerMooseObject("TutorialApp", EFieldAdvection);

template <>
InputParameters
validParams<EFieldAdvection>()
{
  InputParameters params = validParams<Kernel>();
  params.addRequiredCoupledVar("potential", "The electric potential.");
  params.addRequiredParam<Real>("mobility", "Electron mobility.");
  params.addRequiredParam<Real>("sign", "Species charge sign (1.0 or -1.0).");
  return params;
}

EFieldAdvection::EFieldAdvection(const InputParameters & parameters)
  : Kernel(parameters),

    _grad_potential(coupledGradient("potential")),
    _mobility(getParam<Real>("mobility")),
    _sign(getParam<Real>("sign")),
    _potential_id(coupled("potential"))
{
}
```

Field Advection C File (cont.)

```
Real
EFieldAdvection::computeQpResidual()
{
  return _sign * _mobility * _u[_qp] * _grad_potential[_qp] *
        _grad_test[_i][_qp];
}

Real
EFieldAdvection::computeQpJacobian()
{
  return _sign * _mobility * _phi[_j][_qp] * _grad_potential[_qp] *
        _grad_test[_i][_qp];
}

Real
EFieldAdvection::computeQpOffDiagJacobian(unsigned int jvar)
{
  if (jvar == _potential_id)
    return _sign * _mobility * _u[_qp] * _grad_phi[_j][_qp] *
          _grad_test[_i][_qp];

  else
    return 0;
}
```

Ground State Ionization Header File

```
#ifndef GNDSTATEIONIZATIONELECTRONS_H
#define GNDSTATEIONIZATIONELECTRONS_H

#include "Kernel.h"

class GndStateIonizationElectrons;

template <>
InputParameters validParams<GndStateIonizationElectrons>();

class GndStateIonizationElectrons : public Kernel
{
public:
  GndStateIonizationElectrons(const InputParameters & parameters);

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

private:
  const VariableValue & _second_species_density;

  const MaterialProperty<Real> & _k;

  const VariableValue & _mean_en;
};
```

Ground State Ionization C File

```
#include "GndStateIonizationElectrons.h"

registerMooseObject("TutorialApp", GndStateIonizationElectrons);

template <>
InputParameters
validParams<GndStateIonizationElectrons>()
{
  InputParameters params = validParams<Kernel>();
  params.addRequiredCoupledVar("second_species",
          "Second species involved in reaction.");
  params.addRequiredCoupledVar("mean_energy",
          "Electron mean energy variable name.");
  return params;
}

GndStateIonizationElectrons::GndStateIonizationElectrons
                            (const InputParameters & parameters)
  : Kernel(parameters),

    _second_species_density(coupledValue("second_species")),
    _k(getMaterialProperty<Real>("ki")),
    _mean_en(coupledValue("mean_energy"))
{
}
```

Ground State Ionization C File (cont.)

```
Real
GndStateIonizationElectrons::computeQpResidual()
{
  return -_test[_i][_qp] * _k[_qp] * _second_species_density[_qp] * _u[_qp];
}

Real
GndStateIonizationElectrons::computeQpJacobian()
{
  return 0;
}
```

Manufactured Solutions

- As will be discussed more in depth later on in this workshop, it is generally important to manufacture an artificial test for a smaller part of a larger code to test proper functionality.

- Manufacturing a solution creates boundary conditions and custom "right hand side" term that "forces" the problem to converge to a pre-determined solution.

- In this case, we have designed the solution to this problem to be:

$$n_e = 2e7 * (-x^2 + 2.54x) + 1000$$

Input File

```
[Mesh]
  type = GeneratedMesh
  dim = 1
  nx = 1000
  xmax = 2.54
[]

[Variables]
  [./ne] # Scaling is sometimes needed for convergence
    scaling = 1e-13
  [../]
[]

[AuxVariables]
  [./potential_MMS]
    order = FIRST
    family = LAGRANGE
  [../]
  [./ne_MMS]
    order = FIRST
    family = LAGRANGE
  [../]
[]
```

Input File (cont.)

```
[AuxKernels]
  [./potential_aux]
    type = FunctionAux
    variable = potential_MMS
    function = potential_func
  [../]
  [./ne_MMS_aux]
    type = FunctionAux
    variable = ne_MMS
    function = ne_profile_func
  [../]
[]
```

Input File (cont.)

```
[Kernels]
  [./electron_diffusion]
    type = CoeffDiffusion
    diffusivity = 1.1988e6
    variable = ne
  [../]
  [./electron_field_advection]
    type = EFieldAdvection
    variable = ne
    potential = potential_MMS
    mobility = 3e5
    sign = -1.0
  [../]
  [./ne_gnd_state_ioniz]
    type = GndStateIonizationElectrons
    variable = ne
    second_species = 3.22e16 # background gas density (cm^-3)
    mean_energy = 4 # electron mean energy (3/2 * T_e)
  [../]
  [./rhs]
    type = BodyForce
    function = rhs_func
    variable = ne
  [../]
[]
```

Input File (cont.)

```
[BCs]
  [./left]
    type = DirichletBC
    variable = ne
    value = 1000
    boundary = left
  [../]
  [./right]
    type = DirichletBC
    variable = ne
    value = 1000
    boundary = right
  [../]
[]
```

Input File (cont.)

```
[Functions]
  [./potential_func]
    type = ParsedFunction
    value = '(1 / 5) * (-x * x + 2.54 * x)'
  [../]
  [./rhs_func]
    type = ParsedFunction
    value = '2400038827339.6 * x * x - 6096098621442.58 * x +
      (-1.2e13 * x + 1.524e13) * (-0.4 * x + 0.508) + 47951879998058.6'
  [../]
  [./ne_profile_func]
    type = ParsedFunction
    value = '2e7 * (-x * x + 2.54 * x) + 1000'
  [../]
[]

[Materials]
  [./argon_ionization] # assume mean_en = 4
    type = GenericConstantMaterial
    prop_names = ki
    prop_values = 6.02909e-17
  [../]
[]
```

Input File (cont.)

```
[Problem]
  type = FEProblem
  coord_type = XYZ
[]

[Executioner]
  type = Steady
  solve_type = 'PJFNK'
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]

[Outputs]
  exodus = true
[]
```

Create a Test

- Let's take what we've made, since there are a few brand new kernels, and make a test for it.

```
[Tests]
  [./test]
    type = 'Exodiff'
    input = 'step05.i'
    exodiff = 'step05_out.e'
  [../]
[]
```

Coupling

Coupling Contents

Coupling

- In Step 5: we also showcased variable coupling (in that case, AuxVariables into Kernels).

- We'll elaborate more about that system in this section.

Multiphysics Coupling

Coupling Parameters

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

  ```
  params.addCoupledVar("temperature", "The gas temperature.");
  ```

- 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 `const` references that you hold onto in the class.

- The `const` references are then used in the `computeQpResidual()` and `computeQpJacobian()` methods as needed.

Default Coupling Parameters

- To enable rapid development and make debugging more flexible, MOOSE allows you to supply default scalar values where a coupled value would otherwise be required.

  ```
  params.addCoupledVar("temperature", 300, "The gas temperature.");
  ```

- 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.

A bit more about Functions

- ▶ After we couple these equations together, an oscillating voltage will be applied to the biased surface, as in the previous step.

- ▶ MOOSE supports the use of function strings, e.g. cos(x), in the input file via `ParsedFunction` objects.

- ▶ Is is also possible to create C++ `Function` objects that may be used by other objects.

Functions

Functions Overview

- ▶ Function objects allow you to evaluate analytic expressions based on the spatial location (x, y, z) and time, t.

- ▶ You can create a custom Function object by inheriting from Function and overriding the virtual value() (and optionally gradient()) functions.

- ▶ Functions can be accessed in most MOOSE objects (BCs, ICs, Materials, etc.) by calling getFunction("name"), where "name" matches a name from the input file.

- ▶ MOOSE has several built-in Functions, such as:

    - ▶ FunctionDirichletBC
    - ▶ FunctionNeumannBC
    - ▶ FunctionIC
    - ▶ UserForcingFunction

- ▶ Each of these types has a "function" parameter which is set in the input file, and controls which Function object is used.

Default Functions

- Whenever a `Function` object is added via `addParam()`, a default can be provided.

- Both constant values and parsed function strings can be used as the default.

  ```
  ...
  // 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 are declared in the Functions 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 supported.

- After you have declared a Function, you can use them in objects like `FunctionDirichletBC`.

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

Energy Equation

Energy Equation

$$\frac{\partial}{\partial t}\left(\frac{3}{2}n_e T_e\right) + \nabla \cdot \mathbf{q}_e - e\mathbf{\Gamma}_e \cdot \nabla V + \sum_j H_j R_j = 0$$

$$\mathbf{q}_e = -\frac{5}{2}D_e n_e \nabla T_e + \frac{5}{2}T_e \mathbf{\Gamma}_e$$

and

$$\sum_j H_j R_j = 11.56(\text{eV})k_{ex}Nn_e + 15.7k_iNn_e$$

where:

- T_e is the electron temperature

- $k_e x$ is the reaction rate coefficient for the ground state excitation reactions in the plasma

  Assuming that $T_+ = T_{gas}$, we do not need a balance equation for ions.

Things to Note

- To simplify some aspects of the equation, we will be solving for "electron mean energy" rather than electron temperaure. The relationship is $E_N = (3/2) T_e$

- This contribution to the plasma model will be discussed on Day 2, during the Zapdos demonstration.

Auxiliary Variables and Kernels

Auxiliary Variables and Kernels Contents

Auxiliary Variables and Kernels

- In this example, we haven't been calculating the electric field directly.

- However, since we do calculate potential, we can use an `AuxKernel` to calculate an electric field `AuxVariable` that is added to the output file.

- Let's go back the example from Step 3 and expand it to calculate electric field based on the potential result.

Auxiliary 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 input file...useful for viewing things you don't solve for (e.g., velocity).

- Auxiliary variables come in two flavors:

    - Element (constant or higher order monomials)

    - Nodal (linear Lagrange)

- When using element auxiliary variables:

    - You are computing average values per element (constant) or Cholesky solve.

    - You can couple to nonlinear variables and both element and nodal auxiliary variables.

Auxiliary Variables (cont.)

- 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 like nonlinear variables.

Auxiliary Kernels Overview

- `AuxKernel` objects 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, there is no difference between a nodal auxiliary kernel and an elemental.

- The difference is only in the input file.

- An `AuxKernel` operates on an Auxiliary Variable.

(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)

(Some) Values Available to AuxKernels (cont.)

- `_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!

Source Code Header File

```
#ifndef FIELDAUX_H
#define FIELDAUX_H

#include "AuxKernel.h"

class FieldAux;

template <>
InputParameters validParams<FieldAux>();

class FieldAux : public AuxKernel
{
public:
  FieldAux(const InputParameters & parameters);

protected:
  virtual Real computeValue() override;

  const VariableGradient & _grad_potential;
};

#endif // FIELDAUX_H
```

Source Code C File

```c
#include "FieldAux.h"

registerMooseObject("TutorialApp", FieldAux);

template <>
InputParameters
validParams<FieldAux>()
{
  InputParameters params = validParams<AuxKernel>();
  params.addRequiredCoupledVar("potential", "The potential variable.");
  return params;
}

FieldAux::FieldAux(const InputParameters & parameters)
  : AuxKernel(parameters),

    _grad_potential(coupledGradient("potential"))
{
}

Real
FieldAux::computeValue()
{
  return -_grad_potential[_qp](0);
}
```

Input File

```
[Mesh]
 type = GeneratedMesh
 dim = 1
 nx = 1000
 xmax = 0.0254
[]

[Variables]
  [./V]
  [../]
[]

[AuxVariables]
  [./E]
    family = MONOMIAL
    order = FIRST
  [../]
[]
```

Input File (cont.)

```
[Kernels]
  [./poisson]
    type = ExampleMatDiffusion
    variable = V
    # permittivity (eps_R = 1.01) will be taken from a material
  [../]
  [./RHS]
    type = BodyForce
    variable = V
    function = 9.05e6 # from (e / eps0)*(5e15 - 4.5e15)
  [../]
[]

[AuxKernels]
  [./field_aux]
    type = FieldAux
    variable = E
    potential = V
  [../]
[]
```

Input File (cont.)

```
[BCs]
  [./left]
    type = DirichletBC
    variable = V
    boundary = left
    value = 100
  [../]
  [./right]
    type = DirichletBC
    variable = V
    boundary = right
    value = 0
  [../]
[]

[Materials]
  [./example_material]
    type = ExampleMaterial
  [../]
[]
```

Input File (cont.)

```
[Problem]
  type = FEProblem
  coord_type = XYZ
[]

[Executioner]
  type = Steady
  solve_type = 'PJFNK'
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]

[Outputs]
  exodus = true
[]
```

Adaptivity

Adaptivity Contents

Adaptivity

- In the following example, a "traveling wave" profile moves through the porous medium.

    - Instead of using a uniform mesh to resolve the wave profile, we can dynamically adapt the mesh to the solution.

    - No additional code is required to turn on mesh adaptivity.

h-Adaptivity

- *h*-adaptivity is a method of automatically refining/coarsening the mesh in regions of high/low estimated solution error.

- The idea is to concentrate degrees of freedom (DOFs) where the error is highest, while reducing DOFs where the solution is already well-captured.

- This is achieved through splitting and joining elements from the original mesh based on an error `Indicator`.

- Once an error Indicator has been computed, a `Marker` is used to decide which cells to refine and coarsen.

- Mesh adaptivity can be employed in both `Steady` and `Transient` Executioners.

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 "child" elements are created when the element is refined.
- ▶ Coarsening happens in reverse, children are deleted and the "parent" element is reactivated.
- ▶ The original mesh starts at refinement level 0.
- ▶ Each time an element is split, the children are assigned a refinement level one higher than their parents.
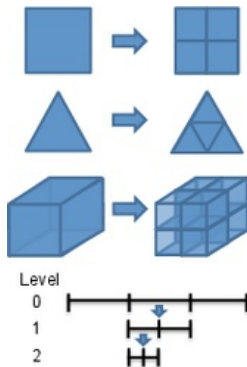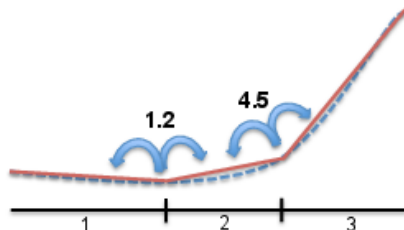
Indicators

- Indicators attempt to report a relative amount of "error" for each element.

- Built-in `Indicators` include:

Indicator|	**Description**
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|	Similar to `GradientJump|`, except that a scalar coefficient (e.g. thermal conductivity) can be provided to produce a physical ``flux" quantity.
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 user-supplied Function representing the analytic solution to the problem.

Indicators (cont.)

- 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:

Marker|	Description
ErrorFractionMarker|	Selects elements based on their contribution to the total error (see Figure).
ErrorToleranceMaker|	Refine if error is greater than a specified value. Coarsen if less than.
ValueThresholdMarker|	Refine if variable value is greater than a specific value. Coarsen if less than.
ValueThresholdMarker|	Refine or coarsen all elements.
BoxMarker|	Refine or coarsen inside or outside a given box.
ComboMarker|	Combine several of the above `Markers`|.

Marker (cont.)

- `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

- To enable adaptivity, add an Adaptivity block to the input file.

- The Adaptivity block 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 by Transient executioners.

- Adaptivity has two sub-blocks: Indicators and Markers. Within these blocks, you can specify multiple Indicator and Marker objects that will be active in the simulation.

```
[Adaptivity]
  marker = errorfrac
  steps = 2
  [./Indicators]
    [./error]
      type = GradientJumpIndicator
      variable = convected
    [../]
  [../]
  [./Markers]
    [./errorfrac]
      type = ErrorFractionMarker
      refine = 0.5
      coarsen = 0
      indicator = error
    [../]
  [../]
[]
```

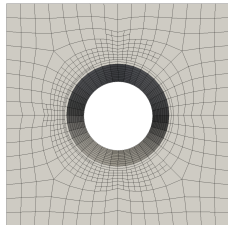# Mesh Adaptivity Input Syntax Example

- Let's look at `moose/examples/ex05_amr` for an example of flow around an obstruction requiring a fine mesh to resolve.
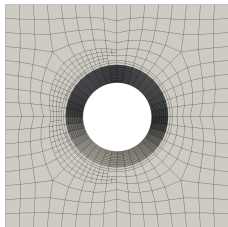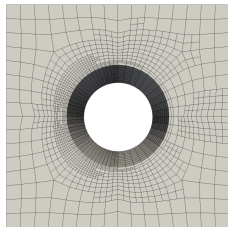
MOOSE Example 05 Results

Original Mesh
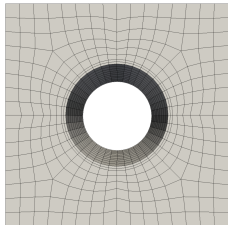
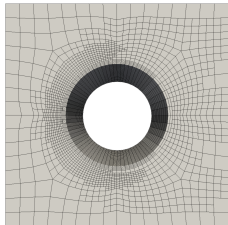Adaptivity Step 2

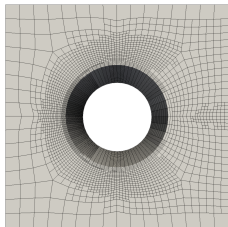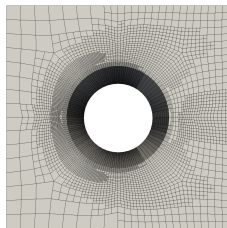Adaptivity Step 1

Adaptivity Step 3

MOOSE Example 05 Results (cont.)

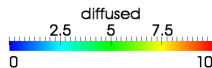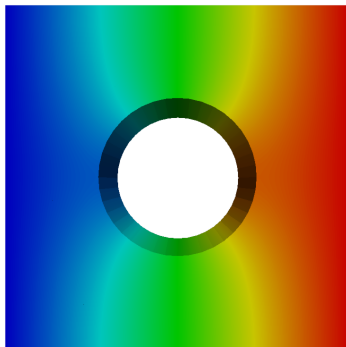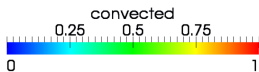Adaptivity Step 4

Adaptivity Step 5

Adaptivity Step 6

MOOSE Example 05 Results (cont.)

Final Results
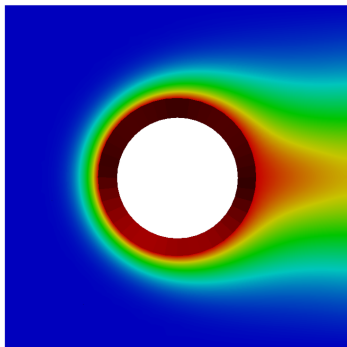
Interface Kernels

Interface Kernels Contents

Interface Kernels

- Interface kernels are meant to assist in coupling different physics across sub-domains

- The most straightforward example is the case in which one wants to set the flux of a specie A in subdomain 0 equal to the flux of a specie B in subdomain 1 at the boundary between subdomains 0 and 1

- Interface kernels can be used to provide any general flux condition at an interface, and even more generally can be used to impose any interfacial condition that requires access to values of different variables and gradients of different variables on either side of an interface

Interface Kernels Example

- In mathematical terms, one might be interested in setting the condition:

$$-D_0 \frac{\partial c_0}{\partial x} = -D_1 \frac{\partial c_1}{\partial x}$$

- This condition requires domain 0 to access a variable that exists at the neighboring domain 1 (and vice-versa).

- In an input file, the user will specify the following parameters:

  - `type`: The type of interface kernel to be used

  - `variable`: The "master" variable. This variable must exist on the same subdomain as the sideset specified in the `boundary` parameter. In this case, our variable might be c_0.

  - `neighbor_var`: The "slave" variable. (c_1)

  - `boundary`: The interfacial boundary between subdomains. This must be a sideset that exists on the same subdomain as the master variable. (Being a sideset, it has access to variable gradients.)

Interface Kernels Example (Input File)

```
[Mesh]
  type = GeneratedMesh
  dim = 1
  nx = 10
  xmax = 2
[]

[MeshModifiers]
  [./subdomain1]
    type = SubdomainBoundingBox
    bottom_left = '1.0 0 0'
    block_id = 1
    top_right = '2.0 1.0 0'
  [../]
  [./interface]
    type = SideSetsBetweenSubdomains
    depends_on = subdomain1
    master_block = '0'
    paired_block = '1'
    new_boundary = 'master0_interface'
  [../]
[]

[Variables]
  [./u]
    order = FIRST
    family = LAGRANGE
    block = '0'
  [../]
  [./v]
    order = FIRST
    family = LAGRANGE
    block = '1'
  [../]
[]
```

```
[Kernels]
  [./diff_u]
    type = CoeffParamDiffusion
    variable = u
    D = 4
    block = 0
  [../]
  [./diff_v]
    type = CoeffParamDiffusion
    variable = v
    D = 2
    block = 1
  [../]
[]

[InterfaceKernels]
  active = 'interface'
  [./interface]
    type = InterfaceDiffusion
    variable = u
    neighbor_var = v
    boundary = master0_interface
    D = 'D'
    D_neighbor = 'D'
  [../]
  [./penalty_interface]
    type = PenaltyInterfaceDiffusion
    variable = u
    neighbor_var = v
    boundary = master0_interface
    penalty = 1e6
  [../]
[]
```

Interface Kernels Example (Input File (cont.))

```
[BCs]
  active = 'left right middle'
  [./left]
    type = DirichletBC
    variable = u
    boundary = 'left'
    value = 1
  [../]
  [./right]
    type = DirichletBC
    variable = v
    boundary = 'right'
    value = 0
  [../]
  [./middle]
    type = MatchedValueBC
    variable = v
    boundary = 'master0_interface'
    v = u
  [../]
[]

[Preconditioning]
  [./smp]
    type = SMP
    full = true
  [../]
[]
```

```
[Materials]
  [./stateful]
    type = StatefulMaterial
    initial_diffusivity = 1
    boundary = master0_interface
  [../]
  [./block0]
    type = GenericConstantMaterial
    block = '0'
    prop_names = 'D'
    prop_values = '4'
  [../]
  [./block1]
    type = GenericConstantMaterial
    block = '1'
    prop_names = 'D'
    prop_values = '2'
  [../]
[]

[Executioner]
  type = Steady
  solve_type = NEWTON
[]

[Outputs]
  exodus = true
  print_linear_residuals = true
[]

[Debug]
  show_var_residual_norms = true
[]
```

Interface Kernels Example (Header File)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef INTERFACEDIFFUSION_H
#define INTERFACEDIFFUSION_H

#include "InterfaceKernel.h"

// Forward Declarations
class InterfaceDiffusion;

template <>
InputParameters validParams<InterfaceDiffusion>();

/**
 * DG kernel for interfacing diffusion between two variables on adjacent blocks
 */
class InterfaceDiffusion : public InterfaceKernel
{
public:
  InterfaceDiffusion(const InputParameters & parameters);

protected:
  virtual Real computeQpResidual(Moose::DGResidualType type) override;
  virtual Real computeQpJacobian(Moose::DGJacobianType type) override;

  const MaterialProperty<Real> & _D;
  const MaterialProperty<Real> & _D_neighbor;
};
```

Interface Kernels Example (C File)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "InterfaceDiffusion.h"

registerMooseObject("MooseTestApp", InterfaceDiffusion);

template <>
InputParameters
validParams<InterfaceDiffusion>()
{
  InputParameters params = validParams<InterfaceKernel>();
  params.addParam<MaterialPropertyName>("D", "D", "The diffusion coefficient.");
  params.addParam<MaterialPropertyName>(
      "D_neighbor", "D_neighbor", "The neighboring diffusion coefficient.");
  return params;
}

InterfaceDiffusion::InterfaceDiffusion(const InputParameters & parameters)
  : InterfaceKernel(parameters),
    _D(getMaterialProperty<Real>("D")),
    _D_neighbor(getNeighborMaterialProperty<Real>("D_neighbor"))
{
}
```

Interface Kernels Example (C File cont.)

```
Real
InterfaceDiffusion::computeQpResidual(Moose::DGResidualType type)
{
  Real r = 0;

  switch (type)
  {
    case Moose::Element:
      r = _test[_i][_qp] * -_D_neighbor[_qp] * _grad_neighbor_value[_qp] *
          _normals[_qp];
      break;

    case Moose::Neighbor:
      r = _test_neighbor[_i][_qp] * _D[_qp] * _grad_u[_qp] *
          _normals[_qp];
      break;
  }

  return r;
}
```

Interface Kernels Example (C File cont.)

```
Real
InterfaceDiffusion::computeQpJacobian(Moose::DGJacobianType type)
{
  Real jac = 0;

  switch (type)
  {
    case Moose::ElementElement:
    case Moose::NeighborNeighbor:
      break;

    case Moose::NeighborElement:
      jac = _test_neighbor[_i][_qp] * _D[_qp] * _grad_phi[_j][_qp] *
            _normals[_qp];
      break;

    case Moose::ElementNeighbor:
      jac = _test[_i][_qp] * -_D_neighbor[_qp] *
            _grad_phi_neighbor[_j][_qp] * _normals[_qp];
      break;
  }

  return jac;
}
```

Postprocessors

Postprocessors Contents

Postprocessors

- ▶ Postprocessors are used to compute aggregate values from solution fields. For example: average velocity, surface power deposition, maximum temperature, etc.

    - ▶ MOOSE comes with several standard Postprocessors, but you can also inherit from the Postprocessor class and customize it for a particular problem.

    - ▶ As an example, an ElementAverageValue is used to compute the average temperature of a domain:

    $$T_{avg} = \frac{1}{\mid \Omega \mid} \int_\Omega T \, dx$$

    where $\mid \Omega \mid$ is the "volume" of the domain.

Overview

- A Postprocessor is a "reduction" or "aggregation" calculation based on the solution variables which results in a **single** scalar value.

- Postprocessors are computed at the times specified by the execute_on option in the input file:

  - execute_on = timestep_end

  - execute_on = linear

  - execute_on = nonlinear

  - execute_on = timestep_begin

  - execute_on = custom

- They can be restricted to specific blocks, sidesets, and nodesets in your domain.

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`.

Types of Postprocessors (cont.)

- Side
  - Operate on boundaries.
  - Must specify one or more boundary ids to compute on.
  - 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.

Postprocessor Anatomy (cont.)

- ▶ `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

If the `Postprocessor` you are creating has custom data (i.e. you are accumulating or computing a member variable inside your `Postprocessor`) you will need to ensure that the value is communicated properly in (both MPI and thread-based) parallel simulations.

For MPI 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.

Helpful Aggregation Routines (cont.)

- 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, . . .)

```
void
PPSum::finalize()
{
  gatherSum(_total_value);
}
```

ThreadJoin (Advanced)

- ▶ You do not need to implement this function 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.

- ▶ Use to perform custom aggregation operations for your class.

```
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;
}
```

Postprocessor Types

- ▶ A few types of built in Postprocessors:

    - ▶ ElementIntegral, ElementAverageValue
    - ▶ SideIntegral, SideAverageValue
    - ▶ ElementL2Error, ElementH1Error

- ▶ Each of these classes can be extended via inheritance.

- ▶ 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.

- ▶ Postprocessors can also be written to Exodus files as "global" data.

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, "Doc String");
  ```

- 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 in the same way that Kernels, BCs, etc. can.

```
[Postprocessors]
  [./dofs]
    type = NumDOFs
  [../]
  [./h1_error]
    type = ElementH1Error
    variable = forced
    function = bc_func
  [../]
  [./l2_error]
    type = ElementL2Error
    variable = forced
    function = bc_func
  [../]
[]
```

Input File Syntax and Output (cont.)

▶ Postprocessor results are printed to the screen as well as *.csv and *.e files.

```
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 |
+----------------+----------------+----------------+----------------+
```

MooseApp and main()

MooseApp and main() Contents

MooseApp Base Class

- ► Every MOOSE application contains a `MooseApp`-derived class.

- ► The MooseApp contains:

    - ► Factories where application objects are built.
    - ► Warehouses where application objects are stored.

- ► MooseApp objects can be combined (coupled) together to form larger simulations, see the `MultiApp` discussion.

- ► MooseApp objects specialize the `validParams()` function.

- ► Parameters for MooseApps are read from the command line via:

    - ► `InputParameters::addRequiredCommandLineParam()`
    - ► `InputParameters::addCommandLineParam()`

Your AnimalApp.h/.C

- ▶ Your application's `MooseApp`-derived class must implement several important static functions:

    - ▶ `registerApps()`

    - ▶ `registerObjects()`

    - ▶ `associateSyntax()` (optional)

- ▶ The `MooseApp` is a convenient place to see the other applications and objects a given application depends on.

Object Registration

- The `MooseApp` object calls its own `registerObjects()` function and those of applications to which it is coupled.

    - This is where you register all of the objects created in your application.

    - Typically done in the constructor.

- Strong coupling to other applications is implemented by calling the `registerObjects()` functions of other applications (also done in the constructor).

    - This makes the other application's objects available in your application.

- Newly-created applications (via `Stork`) register all MOOSE and MOOSE module objects by default.

main.C

- ▶ Every MOOSE application has a `main()` program, typically located in `src/main.C`.

- ▶ By default, a MOOSE application's `main()` program only does a few very specific things:

    - ▶ Calls `MooseInit()`

    - ▶ Calls `YourApp::registerApps()`

    - ▶ Builds your application object

    - ▶ Runs your application object

    - ▶ Deletes your application object

- ▶ See main.C in Example 1

- ▶ While it is possible to add additional code in `main()`, it's not recommended.

Preconditioning

Preconditioning Contents

Preconditioning Overview

- Krylov methods need preconditioning to be efficient (or even effective!).

  - Reduces the total number of linear iterations

  - Each linear iteration in MOOSE includes a residual evaluation (computeQpResidual)

  - Krylov methods, in theory, converge in the number of linear iterations equal to the number of unknowns in the system

- 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.

Preconiditoned JFNK

- Using right preconditioning, solve

$$\mathbf{J}(\vec{u_i})\mathbf{M}^{-1}(\mathbf{M}\delta\vec{u}_{i+1}) = -\vec{R}(\vec{u_i})$$

- **M** symbolically represents the preconditioning matrix or process

- Inside GMRES, we only apply the action of $\mathbf{M}^{-1}$ on a vector

- Recall the *unpreconditioned* JFNK approximation ($\mathbf{M}^{-1} = \mathbf{I}$):

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

- Compare to the right-preconditioned, matrix-free version:

$$\mathbf{J}(\vec{u_i})\mathbf{M}^{-1}\vec{v} \approx \frac{\vec{R}(\vec{u_i} + \epsilon\mathbf{M}^{-1}\vec{v}) - \vec{R}(\vec{u_i})}{\epsilon}$$

Preconditioning Matrix vs Process

- On the previous slide **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 several different ways to build preconditioning matrices:

  - Default: Block Diagonal Preconditioning

  - Single Matrix Preconditioner (SMP)

  - Finite Difference Preconditioner (FDP)

  - Physics Based Preconditioner (PBP)

  - Field Split Preconditioner

- 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 Executioner Options

petsc_options	Description
-snes_ksp_ew	Variable linear solve tolerance, useful for transient solves
-help	Show PETSc options during the solve

PETSc Specific Executioner Options (cont.)

petsc_options_iname	petsc_options_value	Description
-pc_type	ilu	Default for serial
	bjacobi	Default for parallel with -sub_pc_type ilu
	asm	Additive Schwartz with -sub_pc_type ilu
	lu	Full LU, serial only
	gamg	Generalized Geometric-Algebraic Multigrid
	hypre	Hypre, usually used with boomeramg
-sub_pc_type	ilu, lu, hypre	Can be used with bjacobi or asm
-pc_hypre_type	boomeramg	Algebraic Multigrid
-ksp_gmres_restart	# (default = 30)	Number of Krylov vectors to store

Default Preconditioning Matrix

- Consider the fully coupled system of equations:

$$-\nabla \cdot k(T,s)\nabla T = 0$$
$$-\nabla \cdot D(T,s)\nabla s = 0$$

- Fully coupled Jacobian approximation

$$\mathbf{J}(T,s) = \begin{bmatrix} \frac{\partial (R_T)_i}{\partial T_j} & \frac{\partial (R_T)_i}{\partial s_j} \\ \frac{\partial (R_s)_i}{\partial T_j} & \frac{\partial (R_s)_i}{\partial s_j} \end{bmatrix} \approx \begin{bmatrix} \frac{\partial (R_T)_i}{\partial T_j} & 0 \\ 0 & \frac{\partial (R_s)_i}{\partial s_j} \end{bmatrix}$$

- For our example:

$$\mathbf{M} \equiv \begin{bmatrix} (k(T,s)\nabla \phi_j, \nabla \psi_i) & 0 \\ 0 & (D(T,s)\nabla \phi_j, \nabla \psi_i) \end{bmatrix} \approx \mathbf{J}(T,s)$$

- 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)

▶ Single Matrix Preconditioner (SMP) builds one preconditioning matrix.

▶ 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 **M** like this:

$$\mathbf{M} = \begin{bmatrix} (k(T,s)\nabla\phi_j, \nabla\psi_i) & 0 \\ \left(\frac{\partial D(T,s)}{\partial T_j}\nabla s, \nabla\psi_i\right) & (D(T,s)\nabla\phi_j, \nabla\psi_i) \end{bmatrix} \approx \mathbf{J}$$

▶ In order for this to work, you must procide a `computeQpOffDiagJacobian()` function in your Kernels that computes the required partial derivatives.

▶ To use *all* off diagonal blocks set: `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

▶ Default Preconditioning Matrix, Preconditioned JFNK, monitor linear solver, variable linear solver tolerance.

```
[Executioner]
 petsc_options = '-snes_ksp_ew -ksp_monitor'
[]
```

▶ Use Hypre with algebraic multigrid and store 101 Krylov vectors.

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

Examples (cont.)

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

```
[./SMP_jfnk]
  type = SMP

  off_diag_row    = 'forced'
  off_diag_column = 'diffused'

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

Example 11: Preconditioning

Example 11: Preconditioning (SMP Input File)

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

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

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

Example 11: Preconditioning (SMP Input File, cont.)

```
# The Preconditioning block
[Preconditioning]
  active = 'SMP_jfnk'

  [./SMP_jfnk]
    type = SMP

    off_diag_row    = 'forced'
    off_diag_column = 'diffused'

    #Preconditioned JFNK (default)
    solve_type = 'PJFNK'

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

```
[./SMP_jfnk_full]
  type = SMP

  full = true

  #Preconditioned JFNK (default)
  solve_type = 'PJFNK'

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

[./SMP_n]
  type = SMP

  off_diag_row    = 'forced'
  off_diag_column = 'diffused'

  solve_type = 'NEWTON'

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

Example 11: Preconditioning (SMP Input File, cont.)

```
[Kernels]
  [./diff_diffused]
    type = Diffusion
    variable = diffused
  [../]

  [./conv_forced]
    type = CoupledForce
    variable = forced
    v = diffused
  [../]

  [./diff_forced]
    type = Diffusion
    variable = forced
  [../]
[]
```

Example 11: Preconditioning (SMP Input File, cont.)

```
[BCs]
  # Note we have active on and
  # neglect the right_forced BC
  active = 'left_diffused right_diffused left_forced'
  [./left_diffused]
    type = DirichletBC
    variable = diffused
    boundary = 1
    value = 0
  [../]

  [./right_diffused]
    type = DirichletBC
    variable = diffused
    boundary = 2
    value = 100
  [../]
```

```
  [./left_forced]
    type = DirichletBC
    variable = forced
    boundary = 1
    value = 0
  [../]

  [./right_forced]
    type = DirichletBC
    variable = forced
    boundary = 2
    value = 0
  [../]
[]

[Executioner]
  type = Steady
[]

[Outputs]
  exodus = true
[]
```

Example 11: Preconditioning (FDP Input File)

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

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

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

Example 11: Preconditioning (FDP Input File, cont.)

```
# The Preconditioning block
[Preconditioning]
  active = 'FDP_jfnk'

  [./FDP_jfnk]
    type = FDP

    off_diag_row    = 'forced'
    off_diag_column = 'diffused'

    #Preconditioned JFNK (default)
    solve_type = 'PJFNK'

    petsc_options_iname = '-pc_type -mat_fd_coloring_err -mat_fd_type'
    petsc_options_value = 'lu        1e-6                  ds'
  [../]
```

Example 11: Preconditioning (FDP Input File, cont.)

```
  [./FDP_n]
    type = FDP

    off_diag_row    = 'forced'
    off_diag_column = 'diffused'

    solve_type = 'NEWTON'

    petsc_options_iname = '-pc_type -mat_fd_coloring_err -mat_fd_type'
    petsc_options_value = 'lu        1e-6                  ds'
  [../]

  [./FDP_n_full]
    type = FDP

    full = true

    solve_type = 'NEWTON'

    petsc_options_iname = '-pc_type -mat_fd_coloring_err -mat_fd_type'
    petsc_options_value = 'lu        1e-6                  ds'
  [../]
[]
```

Example 11: Preconditioning (FDP Input File, cont.)

```
[Kernels]
  [./diff_diffused]
    type = Diffusion
    variable = diffused
  [../]

  [./conv_forced]
    type = CoupledForce
    variable = forced
    v = diffused
  [../]

  [./diff_forced]
    type = Diffusion
    variable = forced
  [../]
[]
```

Example 11: Preconditioning (FDP Input File, cont.)

```
[BCs]
  # Note we have active on and
  # neglect the right_forced BC
  active = 'left_diffused right_diffused left_forced'
  [./left_diffused]
    type = DirichletBC
    variable = diffused
    boundary = 'left
    value = 0
  [../]

  [./right_diffused]
    type = DirichletBC
    variable = diffused
    boundary = 'right'
    value = 100
  [../]
```

```
  [./left_forced]
    type = DirichletBC
    variable = forced
    boundary = 'left'
    value = 0
  [../]

  [./right_forced]
    type = DirichletBC
    variable = forced
    boundary = 'right'
    value = 0
  [../]
[]

[Executioner]
  type = Steady
[]

[Outputs]
  exodus = true
[]
```

CoupledForce.h

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef COUPLEDFORCE_H
#define COUPLEDFORCE_H

#include "Kernel.h"

class CoupledForce;

template <>
InputParameters validParams<CoupledForce>();

class CoupledForce : public Kernel
{
public:
  CoupledForce(const InputParameters & parameters);

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

private:
  unsigned int _v_var;
  const VariableValue & _v;
  Real _coef;
};

#endif // COUPLEDFORCE_H
```

CoupledForce.C

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "CoupledForce.h"

#include "MooseVariable.h"

registerMooseObject("MooseApp", CoupledForce);

template <>
InputParameters
validParams<CoupledForce>()
{
  InputParameters params = validParams<Kernel>();

  params.addClassDescription("Implements a source term proportional to the value of a coupled "
                             "variable. Weak form: $(\\psi_i, -\\sigma v)$.");
  params.addRequiredCoupledVar("v", "The coupled variable which provides the force");
  params.addParam<Real>(
      "coef", 1.0, "Coefficent ($\\sigma$) multiplier for the coupled force term.");

  return params;
}
```

CoupledForce.C (cont.)

```
CoupledForce::CoupledForce(const InputParameters & parameters)
  : Kernel(parameters), _v_var(coupled("v")), _v(coupledValue("v")), _coef(getParam<Real>("coef"))
{
  if (_var.number() == _v_var)
    mooseError("Coupled variable 'v' needs to be different from 'variable' with CoupledForce, "
               "consider using Reaction or somethig similar");
}

Real
CoupledForce::computeQpResidual()
{
  return -_coef * _v[_qp] * _test[_i][_qp];
}

Real
CoupledForce::computeQpJacobian()
{
  return 0;
}

Real
CoupledForce::computeQpOffDiagJacobian(unsigned int jvar)
{
  if (jvar == _v_var)
    return -_coef * _phi[_j][_qp] * _test[_i][_qp];
  return 0.0;
}
```

Physics Based Preconditioning (PBP)

- ▶ 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.

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

$$\mathbf{M} \equiv \begin{bmatrix} \frac{\partial (R_u)_i}{\partial u_j} & 0 \\ \frac{\partial (R_v)_i}{\partial u_j} & \frac{\partial (R_v)_i}{\partial v_j} \end{bmatrix} \approx \mathbf{J}$$

$$\mathbf{M}\vec{q} = \vec{p} \quad \Rightarrow \quad \begin{cases} \frac{\partial (R_u)_i}{\partial u_j} \vec{q}_u = \vec{p}_u \\ \frac{\partial (R_v)_i}{\partial v_j} \vec{q}_v = \vec{p}_v - \frac{\partial (R_v)_i}{\partial u_j} \vec{q}_u \end{cases}$$

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'
  [../]
[]
```
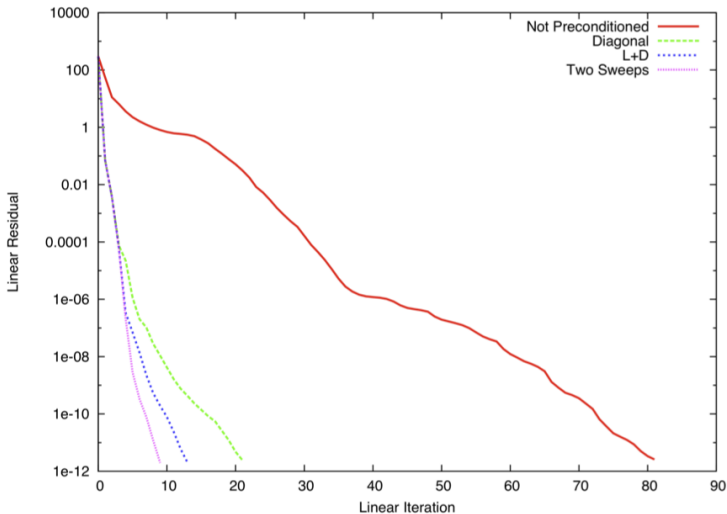
- ▶ Set up a PBP object for a two variable system (consisting of variables "u" and "v").
- ▶ Use ILU for the "u" block and AMG for "v".
- ▶ Use the lower diagonal (v,u) block.
- ▶ When using type = PBP, will set solve_type = JFNK automatically.

Applying PBP

▶ Applying these ideas to a coupled thermo-mechanics problem.

MOOSE Example 12: Physics Based
Preconditioning

MOOSE Example 12: Physics Based Preconditioning (PBP Input File)

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

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

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

MOOSE Example 12: Physics Based Preconditioning (PBP Input File, cont.)

```
# The Preconditioning block
[Preconditioning]
  [./PBP]
    type = PBP
    solve_order = 'diffused forced'
    preconditioner  = 'LU LU'
    off_diag_row    = 'forced'
    off_diag_column = 'diffused'
  [../]
[]
```

MOOSE Example 12: Physics Based Preconditioning (PBP Input File, cont.)

```
[Kernels]
  [./diff_diffused]
    type = Diffusion
    variable = diffused
  [../]

  [./conv_forced]
    type = CoupledForce
    variable = forced
    v = diffused
  [../]

  [./diff_forced]
    type = Diffusion
    variable = forced
  [../]
[]
```

MOOSE Example 12: Physics Based Preconditioning (PBP Input File, cont.)

```
[BCs]
  # Note we have active on and
  # neglect the right_forced BC
  active = 'left_diffused right_diffused left_forced'
  [./left_diffused]
    type = DirichletBC
    variable = diffused
    boundary = 'left
    value = 0
  [../]

  [./right_diffused]
    type = DirichletBC
    variable = diffused
    boundary = 'right'
    value = 100
  [../]
```

```
  [./left_forced]
    type = DirichletBC
    variable = forced
    boundary = 'left'
    value = 0
  [../]

  [./right_forced]
    type = DirichletBC
    variable = forced
    boundary = 'right'
    value = 0
  [../]
[]

[Executioner]
  type = Steady
  solve_type = JFNK
[]

[Outputs]
  exodus = true
[]
```

CoupledForce.h

```cpp
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef COUPLEDFORCE_H
#define COUPLEDFORCE_H

#include "Kernel.h"

class CoupledForce;

template <>
InputParameters validParams<CoupledForce>();

class CoupledForce : public Kernel
{
public:
  CoupledForce(const InputParameters & parameters);

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

private:
  unsigned int _v_var;
  const VariableValue & _v;
  Real _coef;
};

#endif // COUPLEDFORCE_H
```

CoupledForce.C

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "CoupledForce.h"

#include "MooseVariable.h"

registerMooseObject("MooseApp", CoupledForce);

template <>
InputParameters
validParams<CoupledForce>()
{
  InputParameters params = validParams<Kernel>();

  params.addClassDescription("Implements a source term proportional to the value of a coupled "
                             "variable. Weak form: $(\\psi_i, -\\sigma v)$.");
  params.addRequiredCoupledVar("v", "The coupled variable which provides the force");
  params.addParam<Real>(
      "coef", 1.0, "Coefficent ($\\sigma$) multiplier for the coupled force term.");

  return params;
}
```

CoupledForce.C (cont.)

```
CoupledForce::CoupledForce(const InputParameters & parameters)
  : Kernel(parameters), _v_var(coupled("v")), _v(coupledValue("v")), _coef(getParam<Real>("coef"))
{
  if (_var.number() == _v_var)
    mooseError("Coupled variable 'v' needs to be different from 'variable' with CoupledForce, "
               "consider using Reaction or somethig similar");
}

Real
CoupledForce::computeQpResidual()
{
  return -_coef * _v[_qp] * _test[_i][_qp];
}

Real
CoupledForce::computeQpJacobian()
{
  return 0;
}

Real
CoupledForce::computeQpOffDiagJacobian(unsigned int jvar)
{
  if (jvar == _v_var)
    return -_coef * _phi[_j][_qp] * _test[_i][_qp];
  return 0.0;
}
```

Code Verification using Method of Manufactured Solutions

Code Verification using Method of Manufactured Solutions
Contents

Overview

- 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).

MMS Example:

PDE: $-\nabla \cdot \nabla u = 0$

Assumed solution: $u = \sin(\alpha \pi x)$

Forcing function: $f = \alpha^2 \pi^2 \sin(\alpha \pi x)$

Need to solve: $-\nabla \cdot \nabla u - f = 0$

Error Analysis

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

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

$$||f_1 - f_2||^2_{H_{1,\text{semi}}(\Omega)} = \int_\Omega |\nabla(f_1 - f_2)|^2 d\Omega$$

- 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`.

MOOSE Example 14: Postprocessors and Code Verification

Complete Source File

```
[Mesh]
  type = GeneratedMesh
  dim = 2
  nx = 10
  ny = 10
  xmin = 0.0
  xmax = 1.0
  ymin = 0.0
  ymax = 1.0
[]

[Variables]
  [./forced]
    order = FIRST
    family = LAGRANGE
  [../]
[]
```

Complete Source File (cont.)

```
[Functions]
  # A ParsedFunction allows us to supply analytic expression
  # directly in the input file
  [./bc_func]
    type = ParsedFunction
    value = sin(alpha*pi*x)
    vars = alpha
    vals = 16
  [../]

  # This function is an actual compiled function
  # We could have used ParsedFunction for this as well
  [./forcing_func]
    type = ExampleFunction
    alpha = 16
  [../]
[]
```

Complete Source File (cont.)

```
[Kernels]
  [./diff]
    type = Diffusion
    variable = forced
  [../]

  # This Kernel can take a function name to use
  [./forcing]
    type = BodyForce
    variable = forced
    function = forcing_func
  [../]
[]

[BCs]
  # The BC can take a function name to use
  [./all]
    type = FunctionDirichletBC
    variable = forced
    boundary = 'bottom right top left'
    function = bc_func
  [../]
[]
```

Complete Source File (cont.)

```
[Executioner]
  type = Steady
  solve_type = PJFNK
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]

[Adaptivity]
  marker = uniform
  steps = 5

  # Uniformly refine the mesh
  # for the convergence study
  [./Markers]
    [./uniform]
      type = UniformMarker
      mark = REFINE
      outputs = none
    [../]
  [../]
[]
```
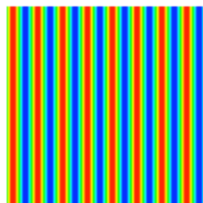
Complete Source File (cont.)

```
[Postprocessors]
  [./dofs]
    type = NumDOFs
  [../]
  [./integral]
    type = ElementL2Error
    variable = forced
    function = bc_func
  [../]
[]

[Outputs]
  execute_on = 'timestep_end'
  exodus = true
  csv = true
[]
```
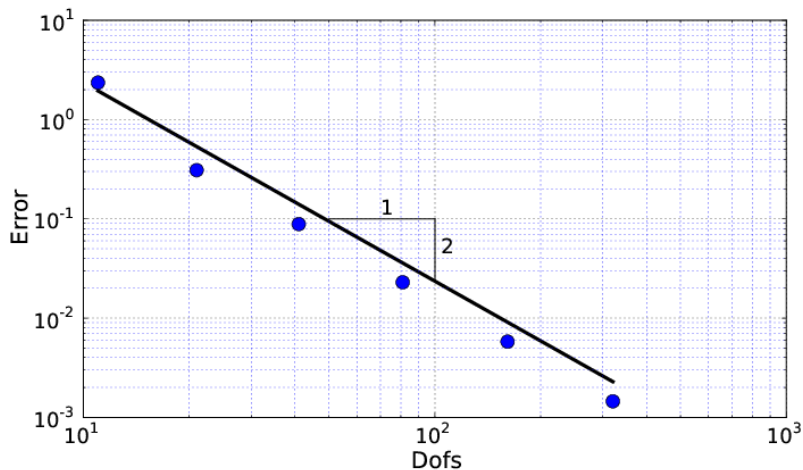
MOOSE Example 14 Output


```
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 |
+----------------+----------------+----------------+
```

MOOSE Example 14 Output (cont.)

Comparison to a Fine Grid Solution

Also present in Example 14 are two input files
(ex14_solution_comparison_1.i and
ex14_solution_comparison_2.i) that demonstrate how to use
a SolutionUserObject to read in a fine grid solution and then
compare a coarse grid solution to that using ElementL2Error.

► The first input file computes the fine grid solution and outputs an
XDA file.

► An XDA file contains the full set of data necessary to perfectly
read a previous solution...even on adapted meshes.

► The second input file uses SolutionUserObject to read in the
fine-grid solution.

► Next, a SolutionFunction utilizes SolutionUserObject to
present the solution field as a MOOSE Function.

► Finally, an ElementL2Error Postprocessor is utilized to compute
the difference between the solutions.

Comparison to a Fine Grid Solution (1)

```
[Mesh]
  type = GeneratedMesh
  dim = 2
  nx = 100
  ny = 100
  xmin = 0.0
  xmax = 1.0
  ymin = 0.0
  ymax = 1.0

  parallel_type = replicated # This uses SolutionUserObject
                             # which doesn't work with
                             # DistributedMesh.
[]

[Variables]
  [./forced]
    order = THIRD
    family = HERMITE
  [../]
[]
```

Comparison to a Fine Grid Solution (1, cont.)

```
[Kernels]
  [./diff]
    type = Diffusion
    variable = forced
  [../]

  [./forcing]
    type = BodyForce
    variable = forced
    function = 'x*x+y*y' # Any object expecting a function
                         # name can also receive a
                         # ParsedFunction string
  [../]
[]

[BCs]
  [./all]
    type = DirichletBC
    variable = forced
    boundary = 'bottom right top left'
    value = 0
  [../]
[]
```

Comparison to a Fine Grid Solution (1, cont.)

```
[Executioner]
  type = Steady

  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]

[Outputs]
  execute_on = 'timestep_end'
  exodus = true
  xda = true # XDA writes out the perfect internal state of
             # all variables, allowing SolutionUserObject
             # to read back in higher order solutions and
             # adapted meshes
[]
```

Comparison to a Fine Grid Solution (2)

```
[Mesh]
  type = GeneratedMesh
  dim = 2
  nx = 11
  ny = 11
  xmin = 0.0
  xmax = 1.0
  ymin = 0.0
  ymax = 1.0
[]

[Variables]
  [./forced]
    order = FIRST
    family = LAGRANGE
  [../]
[]
```

Comparison to a Fine Grid Solution (2, cont.)

```
[Kernels]
  [./diff]
    type = Diffusion
    variable = forced
  [../]

  [./forcing]
    type = BodyForce
    variable = forced
    function = 'x*x+y*y'
  [../]
[]

[BCs]
  [./all]
    type = DirichletBC
    variable = forced
    boundary = 'bottom right top left'
    value = 0
  [../]
[]
```

Comparison to a Fine Grid Solution (2, cont.)

```
[UserObjects]
  [./fine_solution]
    # Read in the fine grid solution
    type = SolutionUserObject
    system_variables = forced
    mesh = ex14_compare_solutions_1_out_0000_mesh.xda
    es = ex14_compare_solutions_1_out_0000.xda
  [../]
[]

[Functions]
  [./fine_function]
    # Create a Function out of the fine grid solution
    # Note: This references the SolutionUserObject above
    type = SolutionFunction
    solution = fine_solution
  [../]
[]
```

Comparison to a Fine Grid Solution (2, cont.)

```
[Executioner]
  type = Steady

  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'

  [./Quadrature]
    # The integration of the error happens on the coarse mesh
    # To reduce integration error of the finer solution we can
    # raise the integration order.
    # Note: This will slow down the calculation a bit
    order = SIXTH
  [../]
[]
```

Comparison to a Fine Grid Solution (2, cont.)

```
[Postprocessors]
  [./error]
    # Compute the error between the computed solution
    # and the fine-grid solution
    type = ElementL2Error
    variable = forced
    function = fine_function
  [../]
[]

[Outputs]
  execute_on = 'timestep_end'
  exodus = true
[]
```

Debugging

Debugging Contents

Debugging

- It's inevitable: at some point in your MOOSE application development career, you will create a bug.

- Sometimes, print statements are sufficient to help you determine the cause of the error.

- For more complex bugs, a debugger can be more effective than print statements in helping to pinpoint the problem.

- Many debuggers exist: LLDB, GDB, Totalview, ddd, Intel Debugger, etc.

- It's typically best to use a debugger that is associated with your compiler, if one is available.

- Here, we focus on LLDB/GDB since it is relatively simple to use and is included in the MOOSE binary redistributable package.

- A "Segmentation Fault", "Segfault", or "Signal 11" error denotes a memory bug (often array access out of bounds).

- 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.

Debugging Example Problem (MOOSE Example 21)

- ▶ This example is similar to Example 3, except that a common error has been introduced.

- ▶ In `ExampleDiffusion.h`, a `VariableValue` that should be declared as a reference is not:

      const VariableValue _coupled_coef;

- ▶ Not storing this as a reference will cause a **copy** of the `VariableValue` to be made.

- ▶ That copy will never be resized, nor will it ever have values written to it.

- ▶ Attempting to access that `VariableValue` results in a segfault when running in optimized mode:

      Time Step 0, time = 0
                      dt = 0

      Time Step 1, time = 0.1
                      dt = 0.1

Debug Executable

- To use a debugger with a MOOSE-based application, you must compile your application in something other than optimized mode (opt). We highly recommend debug (dbg) mode so that you'll get full line number information in your stack traces:

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

- You will now have a "debug version" of your application called ex21-dbg.

- Next, you need to run your application using either GDB (gcc) or LLDB (clang):

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

- When using either of these tools, the command line arguments to the application appear after the -- separator.

- This will start debugger, load your executable, and leave you at the debugger command prompt.

Using GDB or LLDB

- At any prompt in GDB or LLDB, you can type `h` and hit enter to get help.

- We set a "breakpoint" in `MPI_Abort` so that the code pauses (maintaining the stack trace) before exiting.

      b MPI_Abort

- To run your application, type `r` and hit enter.

- If your application hits the breakpoint in `MPI_Abort` you know it has crashed.

- Type `where` (or `bt`) to see a backtrace.

```
frame #0: 0x0000000106b14a20 libmpi.12.dylib`MPI_Abort
frame #1: 0x00000001000d8e78 libex21-dbg.0.dylib`MooseArray<double>
    ::operator[](this=0x0000000108852680, i=0) const + 2200 at
    MooseArray.h:267
frame #2: 0x00000001000d85ab libex21-dbg.0.dylib`ExampleDiffusion
    ::computeQpResidual(this=0x0000000108852000) + 43 at
    ExampleDiffusion.C:40
frame #3: 0x0000000100c2affb libmoose-dbg.0.dylib`Kernel
    ::computeResidual(this=0x0000000108852000) + 443 at Kernel.C:57
```

Using GDB or LLDB (cont.)

- This backtrace shows that, in
  `ExampleDiffusion::computeQpResidual()` we tried to access
  entry 0 of a `MooseArray` with 0 entries.

- If we look at the relevant line of code, we'll see:

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

- There is only one thing we're indexing into on that line:
  `_coupled_coef`.

- Therefore, we can look at how `_coupled_coef` was declared,
  realize that we forgot an ampersand (&), and fix it!

MOOSE Example 21 Source Code (Input File)

```
[Mesh]
  file = reactor.e
  #Let's assign human friendly names to the blocks on the fly
  block_id = '1 2'
  block_name = 'fuel deflector'

  boundary_id = '4 5'
  boundary_name = 'bottom top'
[]

[Variables]
  #Use active lists to help debug problems. Switching on and off
  #different Kernels or Variables is extremely useful!
  active = 'diffused convected'
  [./diffused]
    order = FIRST
    family = LAGRANGE
    initial_condition = 0.5
  [../]

  [./convected]
    order = FIRST
    family = LAGRANGE
    initial_condition = 0.0
  [../]
[]
```

MOOSE Example 21 Source Code (Input File, cont.)

```
[Kernels]
  # This Kernel consumes a real-gradient material property from the
  # active material
  active = 'convection diff_convected example_diff
           time_deriv_diffused time_deriv_convected'
  [./convection]
    type = ExampleConvection
    variable = convected
  [../]
  [./diff_convected]
    type = Diffusion
    variable = convected
  [../]
  [./example_diff]
    type = ExampleDiffusion
    variable = diffused
    coupled_coef = convected
  [../]
  [./time_deriv_diffused]
    type = TimeDerivative
    variable = diffused
  [../]
  [./time_deriv_convected]
    type = TimeDerivative
    variable = convected
  [../]
[]
```

MOOSE Example 21 Source Code (Input File, cont.)

```
[BCs]
  [./bottom_diffused]
    type = DirichletBC
    variable = diffused
    boundary = 'bottom'
    value = 0
  [../]
  [./top_diffused]
    type = DirichletBC
    variable = diffused
    boundary = 'top'
    value = 5
  [../]
  [./bottom_convected]
    type = DirichletBC
    variable = convected
    boundary = 'bottom'
    value = 0
  [../]
  [./top_convected]
    type = NeumannBC
    variable = convected
    boundary = 'top'
    value = 1
  [../]
[]
```

MOOSE Example 21 Source Code (Input File, cont.)

```
[Materials]
  [./example]
    type = ExampleMaterial
    block = 'fuel'
    diffusion_gradient = 'diffused'

    #Approximate Parabolic Diffusivity
    independent_vals = '0 0.25 0.5 0.75 1.0'
    dependent_vals = '1e-2 5e-3 1e-3 5e-3 1e-2'
  [../]

  [./example1]
    type = ExampleMaterial
    block = 'deflector'
    diffusion_gradient = 'diffused'

    # Constant Diffusivity
    independent_vals = '0 1.0'
    dependent_vals = '1e-1 1e-1'
  [../]
[]
```

MOOSE Example 21 Source Code (Input File, cont.)

```
[Executioner]
  type = Transient
  solve_type = 'PJFNK'
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
  dt = 0.1
  num_steps = 10
[]

[Outputs]
  execute_on = 'timestep_end'
  exodus = true
[]
```

MOOSE Example 21 Source Code (ExampleDiffusion.h)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef EXAMPLEDIFFUSION_H
#define EXAMPLEDIFFUSION_H

#include "Diffusion.h"

// Forward Declarations
class ExampleDiffusion;

/**
 * validParams returns the parameters that this Kernel accepts / needs
 * The actual body of the function MUST be in the .C file.
 */
template <>
InputParameters validParams<ExampleDiffusion>();
```

MOOSE Example 21 Source Code (ExampleDiffusion.h, cont.)

```
class ExampleDiffusion : public Diffusion
{
public:
  ExampleDiffusion(const InputParameters & parameters);

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

  /**
   * THIS IS AN ERROR ON PURPOSE!
   *
   * The "&" is missing here!
   *
   * Do NOT copy this line of code!
   */

  const VariableValue _coupled_coef;
};
#endif // EXAMPLEDIFFUSION_H
```

MOOSE Example 21 Source Code (ExampleDiffusion.C)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "ExampleDiffusion.h"
/**
 * This function defines the valid parameters for
 * this Kernel and their default values
 */
registerMooseObject("ExampleApp", ExampleDiffusion);

template <>
InputParameters
validParams<ExampleDiffusion>()
{
  InputParameters params = validParams<Diffusion>();
  params.addRequiredCoupledVar(
      "coupled_coef", "The value of this variable will be used as the
                       diffusion coefficient.");
  return params;
}
```

MOOSE Example 21 Source Code (ExampleDiffusion.C, cont.)

```
ExampleDiffusion::ExampleDiffusion(const InputParameters & parameters)
  : Diffusion(parameters), _coupled_coef(coupledValue("coupled_coef"))
{
}

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

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

Testing

Testing Contents

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]
```

Test Specification File (Quick Look)

▶ 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
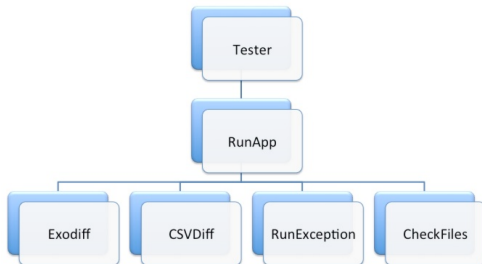
- ▶ RunApp: Runs a MOOSE-based application with specified options.

- ▶ 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!

  - Place the `Tester` object in "/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 Tests

./run_tests [options]

Options	Description
-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' processors

Other Notes on Tests

- Individual tests should run relatively quickly ($\sim$2 second rule)

- Outputs or other generated files should not be checked into the git 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

Mesh Modifiers (Advanced)

Mesh Modifiers (Advanced) Contents

Overview

- `[MeshModifiers]` further modify the mesh after it has been created.

- Possible modifications include: adding node sets, translating, rotating, and scaling the mesh points.

- Users can create their own MeshModifiers by inheriting from `MeshModifier` and defining a custom `modify()` method.

- A few built-ins:

  - `AddExtraNodeset`

  - `SideSetFromNormals` and `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 preserved.
- Newly-created top and bottom sidesets can be named by the user.
- The extrusion vector's direction and length must be specified.
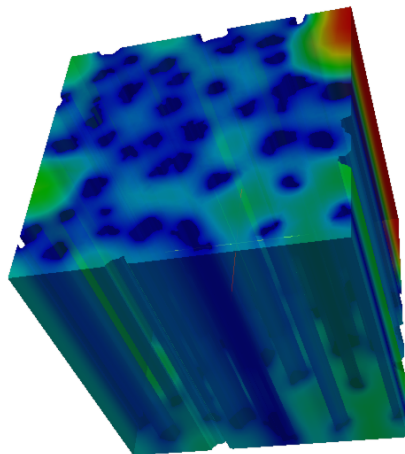
Extruded Mesh result from MAMBA courtest of Michael Short.

Output Oversampling (Advanced)

Output Oversampling (Advanced) Contents

Motivation

- 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.

Input File Syntax

- ▶ Oversampling is handled via the 'refinements' input parameter, which by default is set to zero.

- ▶ To enable oversampling, set this to a positive integer.

- ▶ The following input file snippet created an Exodus II file with two oversample refinements. It also includes a positional offset for the oversampled mesh.

```
[Outputs]
  console = true
  exodus = true
  [./exodus_oversample]
    type = Exodus
    refinements = 2
    file_base = oversample
    position = '1 2 0'
  [../]
[]
```
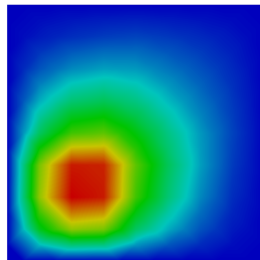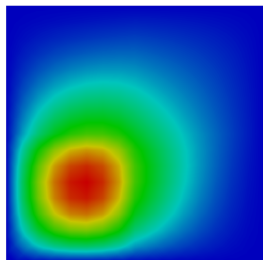
Oversample Example
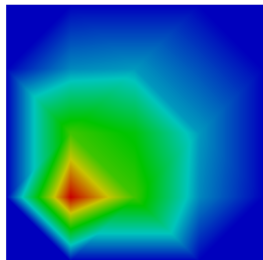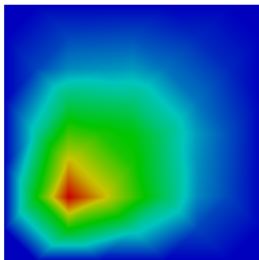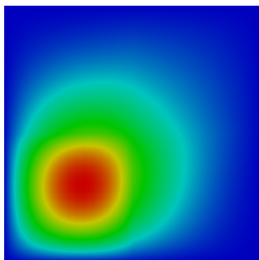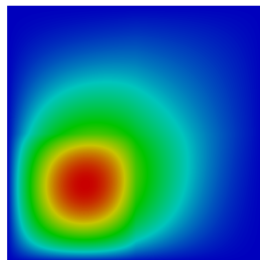

refinements = 0   refinements = 1   refinements = 2

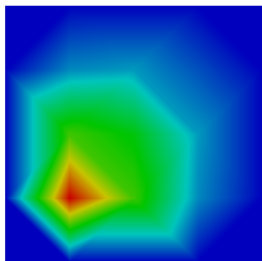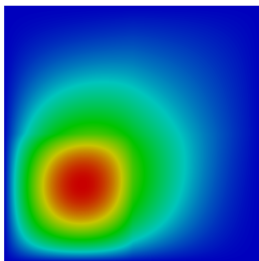refinements = 3   refinements = 4   refinements = 5

Oversample Example (cont.)

- It is 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).

refinements = 0   refinements = 5   "Fine" grid
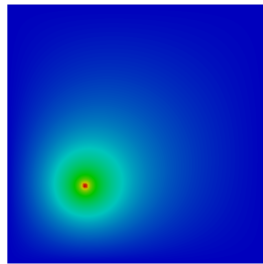
Parallel-Agnostic Random Number Generation
(Advanced)

Parallel-Agnostic Random Number Generation (Advanced)
Contents

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, and
// 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.

The Actions System (Advanced)

The Actions System (Advanced) Contents

The Actions System

- Actions are used to construct other MOOSE objects (Variables, Kernels, etc.)

- MOOSE has a large set of common tasks that are performed for every problem

- Customizable and extendable

Actions System Model - tasks

- ▶ Setting up a problem in MOOSE is very order sensitive.

- ▶ The setup order is defined in `Moose.C` for all MOOSE-based applications

- ▶ Each step of the setup phase is called a `task`

- ▶ MOOSE looks for Actions that will satisfy the current `task`

- ▶ Some tasks are optional, but they must be executed in the proper order

- ▶ The list of tasks can be augmented by registering a new `task` and declaring its dependencies:

  - ▶ `registerTask(task, isrequired);`
  - ▶ `addTaskDependency(task, dependson);`

Actions System Model - Action Object

- To build a new `Action`, inherit from `Action` and override the `act()` function

- Typically you will set up other MOOSE objects with your `Action`. For example:

  - 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` can be registered with more than one `task`

- A single `Action` can only satisfy one `task`, however the system may create several instances of any one `Action` to satisfy multiple `tasks`

Actions System Model - The Parser

▶ 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 the `InputParameters` object with options from the input file
5. Builds and stores the appropriate `Action` in MOOSE's ActionWarehouse

Actions System Model
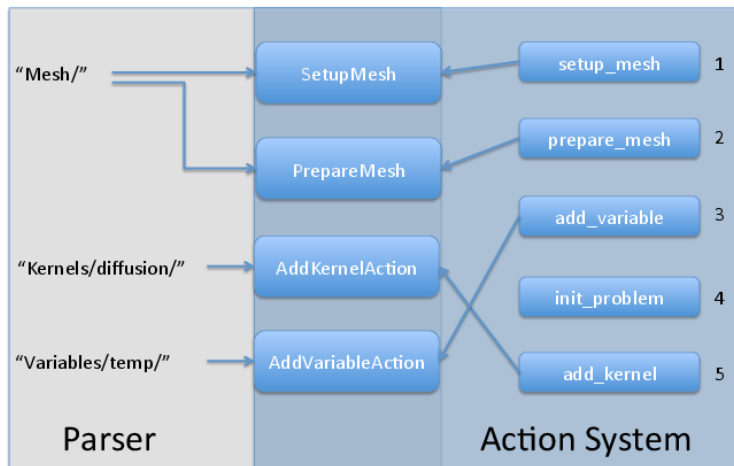
MOOSE Example 15: Custom Action (Input File)

```
[Mesh]
  file = square.e
  uniform_refine = 4
[]

[Variables]
  [./convected]
    order = FIRST
    family = LAGRANGE
  [../]

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

[ConvectionDiffusion]
    variables = 'convected diffused'
[]

[BCs]
  [./left_convected]
    type = DirichletBC
    variable = convected
    boundary = 'left'
    value = 0
  [../]
```

```
  [./right_convected]
    type = DirichletBC
    variable = convected
    boundary = 'right'
    value = 1
    some_var = diffused
  [../]

  [./left_diffused]
    type = DirichletBC
    variable = diffused
    boundary = 'left'
    value = 0
  [../]

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

[Executioner]
  type = Steady
  solve_type = 'PJFNK'
[]

[Outputs]
  execute_on = 'timestep_end'
  exodus = true
[]
```

MOOSE Example 15: Custom Action (Input File)

```
[ConvectionDiffusion]
    variables = 'convected diffused'
[]
```

- ▶ This is our new custom Convection-Diffusion "Meta" block that adds multiple kernels into our simulation

- ▶ Convection and diffusion kernels on the first variable

- ▶ Diffusion kernel on the second variable

- ▶ The convection kernel is coupled to the diffusion kernel on the second variable:

$$\nabla v \cdot \nabla u + \nabla \cdot \nabla u = 0$$
$$\nabla \cdot \nabla v = 0$$

- ▶ where in this case, convected $= u$ and diffused $= v$

MOOSE Example 15: Custom Action (Header File)

```cpp
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef CONVECTIONDIFFUSIONACTION_H
#define CONVECTIONDIFFUSIONACTION_H

#include "Action.h"

class ConvectionDiffusionAction : public Action
{
public:
  ConvectionDiffusionAction(InputParameters params);

  virtual void act() override;
};

template <>
InputParameters validParams<ConvectionDiffusionAction>();

#endif // CONVECTIONDIFFUSIONACTION_H
```

MOOSE Example 15: Custom Action (C File)

```c
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "ConvectionDiffusionAction.h"
#include "Factory.h"
#include "Parser.h"
#include "FEProblem.h"

registerMooseAction("ExampleApp", ConvectionDiffusionAction, "add_kernel");

template <>
InputParameters
validParams<ConvectionDiffusionAction>()
{
  InputParameters params = validParams<Action>();
  params.addRequiredParam<std::vector<NonlinearVariableName>>(
      "variables",
      "The names of the convection and diffusion variables in the simulation");

  return params;
}
```

MOOSE Example 15: Custom Action (C File cont.)

```
void
ConvectionDiffusionAction::act()
{
  std::vector<NonlinearVariableName> variables =
      getParam<std::vector<NonlinearVariableName>>("variables");
  std::vector<VariableName> vel_vec_variable;

  // Do some error checking
  mooseAssert(variables.size() == 2, "Expected 2 variables, received " << variables.size());

  // Setup our Diffusion Kernel on the "u" variable
  {
    InputParameters params = _factory.getValidParams("Diffusion");
    params.set<NonlinearVariableName>("variable") = variables[0];
    _problem->addKernel("Diffusion", "diff_u", params);
  }

  // Setup our Convection Kernel on the "u" variable coupled to the diffusion variable "v"
  {
    InputParameters params = _factory.getValidParams("ExampleConvection");
    params.set<NonlinearVariableName>("variable") = variables[0];
    //    params.addCoupledVar("some_variable", "The gradient of this var");
    vel_vec_variable.push_back(variables[1]);
    params.set<std::vector<VariableName>>("some_variable") = vel_vec_variable;
    _problem->addKernel("ExampleConvection", "conv", params);
  }

  // Setup out Diffusion Kernel on the "v" variable
  {
    InputParameters params = _factory.getValidParams("Diffusion");
    params.set<NonlinearVariableName>("variable") = variables[1];
    _problem->addKernel("Diffusion", "diff_v", params);
  }
}
```

Dirac Kernels (Advanced)

Dirac Kernels (Advanced) Contents

Point Source

- Point sources (sometimes known as point loads) are typically modeled with Dirac distributions in finite element analysis.

- We will employ the following generalized form for the Dirac delta distribution acting at x_0:

$$(\delta_{x_0} f, \psi) \equiv \int_\Omega \delta(x - x_0) f(x) \psi(x) dx = f(x_0) \psi(x_0)$$

  where f and ψ are continuous functions.

- The Dirac delta distribution is thus an *integral operator* which "samples" the continuous functions f and ψ at the point x_0.

- The special case $\psi = 1$ is frequently used to induce the short-hand notation

$$\delta_{x_0} f \equiv \int_\Omega \delta(x - x_0) f(x) dx = f(x_0)$$

Point Source (cont.)

- A diffusion equation with a point source/sink of strength f (which can depend on u) located at x_0 can therefore be written as:

$$-\nabla \cdot \nabla u - \delta_{x_0} f = 0$$

- Using the definition above, the weak form for this equation (assuming Dirichlet BCs) is

$$(\nabla u, \nabla \psi_i) - (\delta_{x_0} f, \psi_i) = 0$$

- Assume the point x_0 falls in element Ω_e. Then:

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

- That is, we get a contribution for each DOF whose associated basis function ψ_i is nonzero at x_0.

- The DiracKernel class computes these contributions given x_0 and f.

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 the virtual `addPoints()` function to tell MOOSE the points at which the `DiracKernel` is active.

Dirac Kernels (cont.)

- Inside of `addPoints()` there are two different ways to tell MOOSE about 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 Dirac Kernels

- `_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 current Dirac point.

- `_test`, `_grad_test`
  - Value (ψ) and gradient ($\nabla\psi$) of the test functions at the current Dirac point.

- `_q_point`
  - XYZ coordinates of the current Dirac point.

- `_i`, `_j`
  - Current shape functions for test and trial functions, respectively.

- `_current_elem`

MOOSE Example 17: Adding a DiracKernel

MOOSE Example 17: Adding a DiracKernel (Input File)

```
[Mesh]
  file = 3-4-torus.e
[]

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

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

[DiracKernels]
  [./example_point_source]
    type = ExampleDirac
    variable = diffused
    value = 1.0
    point = '-2.1 -5.08 0.7'
  [../]
[]
```

MOOSE Example 17: Adding a DiracKernel (Input File) (cont.)

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

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

[Executioner]
  type = Steady
  solve_type = PJFNK
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]

[Outputs]
  execute_on = 'timestep_end'
  exodus = true
[]
```

MOOSE Example 17: Adding a DiracKernel (Header File)

```cpp
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef EXAMPLEDIRAC_H
#define EXAMPLEDIRAC_H

// Moose Includes
#include "DiracKernel.h"

// Forward Declarations
class ExampleDirac;

template <>
InputParameters validParams<ExampleDirac>();
```

```cpp
class ExampleDirac : public DiracKernel
{
public:
  ExampleDirac(const InputParameters & parameters);

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

protected:
  Real _value;
  Point _point;
};

#endif // EXAMPLEDIRAC_H
```

MOOSE Example 17: Adding a DiracKernel (C File)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "ExampleDirac.h"

registerMooseObject("ExampleApp", ExampleDirac);

template <>
InputParameters
validParams<ExampleDirac>()
{
  InputParameters params = validParams<DiracKernel>();
  params.addRequiredParam<Real>("value", "The value of the point source");
  params.addRequiredParam<Point>("point",
                                 "The x,y,z coordinates of the point");
  return params;
}
```

# MOOSE Example 17: Adding a DiracKernel (C File) (cont.)

```
ExampleDirac::ExampleDirac(const InputParameters & parameters)
  : DiracKernel(parameters), _value(getParam<Real>("value")),
    _point(getParam<Point>("point"))
{
}

void
ExampleDirac::addPoints()
{
  // Add a point from the input file
  addPoint(_point);

  // Add another point not read from the input file
  addPoint(Point(4.9, 0.9, 0.9));
}

Real
ExampleDirac::computeQpResidual()
{
  // This is negative because it's a forcing function that has been brought
  // over to the left side
  return -_test[_i][_qp] * _value;
}
```

Example 17 Results

diffused
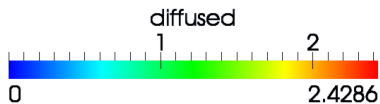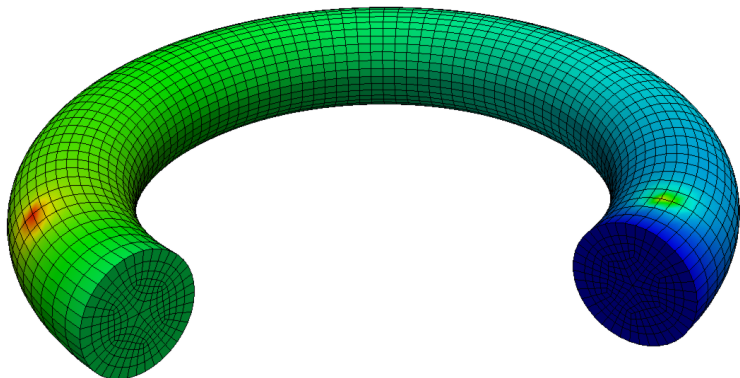
Scalar Kernels (Advanced)

Scalar Kernels (Advanced) Contents

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 (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 (cont.)

▶ Problem being solved:

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

▶ Where the boundary conditions are being governed by the ODEs:

$$\frac{dX}{dt} = 3X + 2Y$$
$$\frac{dY}{dt} = 4X + Y$$

plus suitable initial conditions.

MOOSE Example 18: ODE Coupling

MOOSE Example 18: ODE Coupling (Input File)

```
[Mesh]
  type = GeneratedMesh
  dim = 2
  xmin = 0
  xmax = 1
  ymin = 0
  ymax = 1
  nx = 10
  ny = 10
  elem_type = QUAD4
[]

[Functions]
  # ODEs
  [./exact_x_fn]
    type = ParsedFunction
    value = (-1/3)*exp(-t)+(4/3)*exp(5*t)
  [../]
  [./exact_y_fn]
    type = ParsedFunction
    value = (2/3)*exp(-t)+(4/3)*exp(5*t)
  [../]
[]
```

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

  # ODE variables
  [./x]
    family = SCALAR
    order = FIRST
    initial_condition = 1
  [../]
  [./y]
    family = SCALAR
    order = FIRST
    initial_condition = 2
  [../]

[]
```

MOOSE Example 18: ODE Coupling (Input File) (cont.)

```
[Kernels]
  [./td]
    type = TimeDerivative
    variable = diffused
  [../]
  [./diff]
    type = Diffusion
    variable = diffused
  [../]
[]
```

```
[ScalarKernels]
  [./td1]
    type = ODETimeDerivative
    variable = x
  [../]
  [./ode1]
    type = ImplicitODEx
    variable = x
    y = y
  [../]

  [./td2]
    type = ODETimeDerivative
    variable = y
  [../]
  [./ode2]
    type = ImplicitODEy
    variable = y
    x = x
  [../]
[]
```

```
[BCs]
  [./right]
    type = ScalarDirichletBC
    variable = diffused
    boundary = 1
    scalar_var = x
  [../]

  [./left]
    type = ScalarDirichletBC
    variable = diffused
    boundary = 3
    scalar_var = y
  [../]
[]
```

MOOSE Example 18: ODE Coupling (Input File) (cont.)

```
[Postprocessors]
  # to print the values of x, y
  # into a file so we can plot it
  [./x]
    type = ScalarVariable
    variable = x
    execute_on = timestep_end
  [../]

  [./y]
    type = ScalarVariable
    variable = y
    execute_on = timestep_end
  [../]

  [./exact_x]
    type = FunctionValuePostprocessor
    function = exact_x_fn
    execute_on = timestep_end
    point = '0 0 0'
  [../]

  [./exact_y]
    type = FunctionValuePostprocessor
    function = exact_y_fn
    execute_on = timestep_end
    point = '0 0 0'
  [../]
```

```
  # Measure the error in ODE solution for 'x'.
  [./l2err_x]
    type = ScalarL2Error
    variable = x
    function = exact_x_fn
  [../]

  # Measure the error in ODE solution for 'y'.
  [./l2err_y]
    type = ScalarL2Error
    variable = y
    function = exact_y_fn
  [../]
[]


[Executioner]
  type = Transient
  start_time = 0
  dt = 0.01
  num_steps = 10

  #Preconditioned JFNK (default)
  solve_type = 'PJFNK'
[]

[Outputs]
  exodus = true
[]
```

MOOSE Example 18: ODE Coupling (ImplicitODEx Header File)

```cpp
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef IMPLICITODEX_H
#define IMPLICITODEX_H

#include "ODEKernel.h"

// Forward Declarations
class ImplicitODEx;

template <>
InputParameters validParams<ImplicitODEx>();
```

MOOSE Example 18: ODE Coupling (ImplicitODEx Header File) (cont.)

```cpp
class ImplicitODEx : public ODEKernel
{
public:
  ImplicitODEx(const InputParameters & parameters);

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

  unsigned int _y_var;

  const VariableValue & _y;
};

#endif /* IMPLICITODEX_H */
```

MOOSE Example 18: ODE Coupling (ImplicitODEx C File)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "ImplicitODEx.h"

registerMooseObject("ExampleApp", ImplicitODEx);

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

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

Real
ImplicitODEx::computeQpResidual()
{
  // the term of the ODE without the time derivative term
  return -3. * _u[_i] - 2. * _y[_i];
}
```

MOOSE Example 18: ODE Coupling (ImplicitODEx C File) (cont.)

```
Real
ImplicitODEx::computeQpJacobian()
{
  // dF/dx
  return -3.;
}

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

MOOSE Example 18: ODE Coupling (ImplicitODEy Header File)

```cpp
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef IMPLICITODEY_H
#define IMPLICITODEY_H

#include "ODEKernel.h"

// Forward Declarations
class ImplicitODEy;

template <>
InputParameters validParams<ImplicitODEy>();
```

```cpp
class ImplicitODEy : public ODEKernel
{
public:
  ImplicitODEy(const InputParameters & parameters);

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

  unsigned int _x_var;

  const VariableValue & _x;
};

#endif /* IMPLICITODEY_H */
```

MOOSE Example 18: ODE Coupling (ImplicitODEy C File)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#include "ImplicitODEy.h"

registerMooseObject("ExampleApp", ImplicitODEy);

template <>
InputParameters
validParams<ImplicitODEy>()
{
  InputParameters params = validParams<ODEKernel>();
  params.addCoupledVar("x", "variable X coupled into this kernel");
  return params;
}
```

```
ImplicitODEy::ImplicitODEy(const InputParameters & parameters)
  : ODEKernel(parameters),
    _x_var(coupledScalar("x")),
    _x(coupledScalarValue("x"))
{
}

Real
ImplicitODEy::computeQpResidual()
{
  // the term of the ODE without the time derivative term
  return -4 * _x[_i] - _u[_i];
}
```

MOOSE Example 18: ODE Coupling (ImplicitODEy C File) (cont.)

```
Real
ImplicitODEy::computeQpJacobian()
{
  // dF/dy
  return -1.;
}

Real
ImplicitODEy::computeQpOffDiagJacobian(unsigned int jvar)
{
  if (jvar == _x_var)
    return -4.; // dF/dx
  else
    return 0.; // everything else
}
```

MOOSE Example 18: ODE Coupling (ScalarDirichletBC Header File)

```cpp
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html

#ifndef SCALARDIRICHLETBC_H
#define SCALARDIRICHLETBC_H

#include "NodalBC.h"

// Forward Declarations
class ScalarDirichletBC;
```

```
template <>
InputParameters validParams<ScalarDirichletBC>();

class ScalarDirichletBC : public NodalBC
{
public:
  ScalarDirichletBC(const InputParameters & parameters);

protected:
  virtual Real computeQpResidual() override;

  const VariableValue & _scalar_val;
};

#endif // SCALARDIRICHLETBC_H
```

MOOSE Example 18: ODE Coupling (ScalarDirichletBC C File)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html
#include "ScalarDirichletBC.h"

registerMooseObject("ExampleApp", ScalarDirichletBC);

template <>
InputParameters
validParams<ScalarDirichletBC>()
{
  InputParameters params = validParams<NodalBC>();
  params.addRequiredCoupledVar("scalar_var",
                               "Value of the scalar variable");
  return params;
}
```

MOOSE Example 18: ODE Coupling (ScalarDirichletBC C File) (cont.)

```
ScalarDirichletBC::ScalarDirichletBC(const InputParameters & parameters
  : NodalBC(parameters),
    _scalar_val(coupledScalarValue("scalar_var"))
{
}

Real
ScalarDirichletBC::computeQpResidual()
{
  // We coupled in a first order scalar variable,
  // thus there is only one value in _scalar_val (and
  // it is - big surprise - on index 0)
  return _u[_qp] - _scalar_val[0];
}
```

Example 18 Results
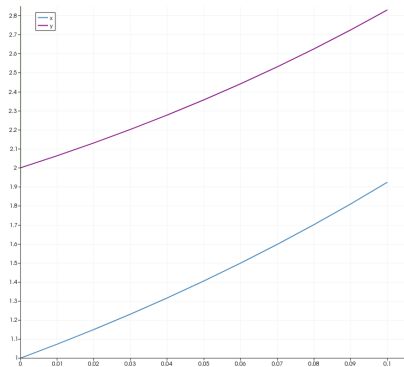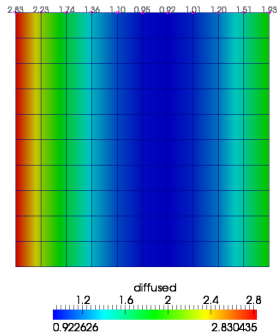
Geometric Search (Advanced)

Geometric Search (Advanced) Contents

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.
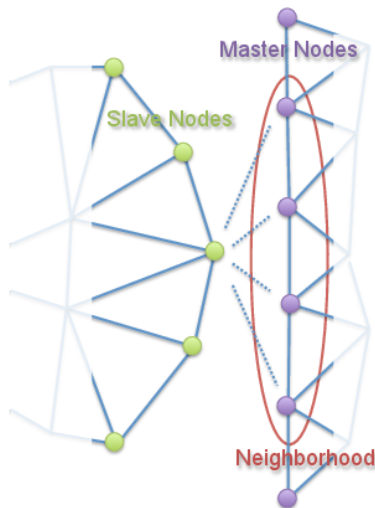
NearestNodeLocator (cont.)

- ▶ To use a NearestNodeLocator

    - ▶ #include "NearestNodeLocator.h"
    - ▶ 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.
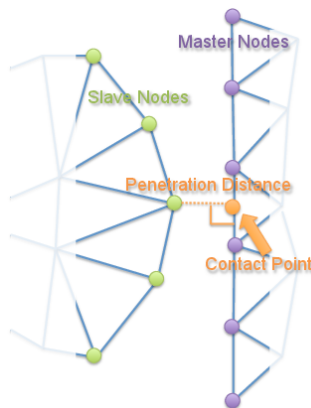
Dampers (Advanced)

Dampers (Advanced) Contents

Dampers

- Dampers compute the Newton damping parameter $0 < \alpha \leq 1$:

  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

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

- A `Damper` is created by inheriting from `Damper` and overriding `computeQpDamping()`.

- `computeQpDamping()` computes a damping parameter at each quadrature point throughout the mesh.

  - The minimum 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 step vector.

Dampers (cont.)
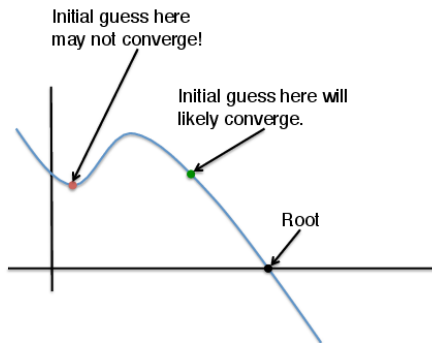
MOOSE Example 19: Newton Damping

MOOSE Example 19: Newton Damping (Input File)

```
[Mesh]
  type = GeneratedMesh
  dim = 2

  xmin = 0.0
  xmax = 1.0
  nx = 10

  ymin = 0.0
  ymax = 1.0
  ny = 10
[]

[Variables]
  [./diffusion]
    order = FIRST
    family = LAGRANGE
  [../]
[]

[Kernels]
  [./diff]
    type = Diffusion
    variable = diffusion
  [../]
[]
```

MOOSE Example 19: Newton Damping (Input File) (cont.)

```
[BCs]
  [./left]
    type = DirichletBC
    variable = diffusion
    boundary = 3
    value = 3
  [../]

  [./right]
    type = DirichletBC
    variable = diffusion
    boundary = 1
    value = 1
  [../]
[]
```

MOOSE Example 19: Newton Damping (Input File) (cont.)

```
[Dampers]
  # Use a constant damping parameter
  [./diffusion_damp]
    type = ConstantDamper
    variable = diffusion
    damping = 0.9
  [../]
[]

[Executioner]
  type = Steady

  #Preconditioned JFNK (default)
  solve_type = 'PJFNK'
[]

[Outputs]
  exodus = true
[]
```

MOOSE Example 19 Results

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)

Discontinuous Galerkin (DG)

- As the name implies, DG methods employ discontinuous finite element basis functions.

  - The finite element solution is piecewise-discontinuous across element boundaries, and continuous within each element.

  - DG formulations for elliptic problems are unstable unless special "penalty" terms are employed (see, e.g. "interior penalty DG" method).

  - You can use DG in MOOSE by specifying a discontinuous family of basis functions (e.g. MONOMIALS) and adding one or more DGKernels.

  - DGKernels can execute alongside regular Kernels.

  - DGKernels are responsible for computing residual and Jacobian contributions due to the "jump" terms along inter-element edges/faces.

  - DG is beyond the scope of this training class, if you want to learn more, please ask!

UserObjects (Advanced)

UserObjects (Advanced) Contents

User Objects

- ▶ The `test` system provides data and calculation results to other MOOSE objects

- ▶ All Postprocessors are UserObjects that compute a single scalar value

- ▶ Therefore, a `UserObject` can 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` - perform evaluations on each element

    - ▶ `NodalUserObject` - perform evaluations on each node

    - ▶ `SideUserObject` - perform evaluations on each side

    - ▶ `GeneralUserObject` - a generic object that can do "anything" while providing a common interface for use by other MOOSE objects

- ▶ 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:

- `virtual void initialize();`

  - Called just one time before beginning the `UserObject` calculation

  - Useful for resetting data structures

- `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.

UserObject Anatomy (cont.)

- `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.

- `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 means the function cannot modify any member variables of the object, and is required for `UserObject` accessors functions.

Using a UserObject

- 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;
}
```

Using a UserObject (cont.)

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

  `const BlockAverageValue & _block_average_value;`

- The reference is set in the initialization list of your object by calling the templated `getUserObject()` method:

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

Using a UserObject (cont.)

- ▶ 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());
```

MOOSE Example 20 - UserObjects

MOOSE Example 20: UserObjects

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. Thus, as the concentration diffuses, the diffusion coefficient increases, but the coefficient is different for each block (based on the average of the variable on that block).

MOOSE Example 20: UserObjects (cont.)

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 have seen before that uses a "diffusivity" material property. The main purpose of this class is to provide the `averageValue` method that accepts a `SubdomainID`, which is simply an integer value specifying which block of the mesh to perform the average value calculation.

BlockAverageValue UserObject

The first step is to create a UserObject for computing the average value of a variable on block (subdomain). The complete header and source file for this custom UserObject are linked below, within each file the comments detail the functionality of the class.

BlockAverageValue UserObject: (Header File)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html
#ifndef BLOCKAVERAGEVALUE_H
#define BLOCKAVERAGEVALUE_H
#include "ElementIntegralVariablePostprocessor.h"
#include "libmesh/mesh_tools.h"
// Forward Declarations
class BlockAverageValue;
template <>
InputParameters validParams<BlockAverageValue>();
/**
 * Computes the average value of a variable on each block
 */
```

BlockAverageValue UserObject: (Header File cont.)

```
class BlockAverageValue : public ElementIntegralVariablePostprocessor
{
public:
  BlockAverageValue(const InputParameters & parameters);
  /**
   * Given a block ID return the average value for a variable on that block
   *
   * Note that accessor functions on UserObjects like this _must_ be const.
   * That is because the UserObject system returns const references to objects
   * trying to use UserObjects.  This is done for parallel correctness.
   *
   * @return The average value of a variable on that block.
   */
  Real averageValue(SubdomainID block) const;
  /**
   * This is called before execute so you can reset any internal data.
   */
```

BlockAverageValue UserObject: (Header File cont.)

```cpp
  virtual void initialize() override;
  /**
   * Called on every "object" (like every element or node).
   * In this case, it is called at every quadrature point on every element.
   */
  virtual void execute() override;
  /**
   * Called when using threading.  You need to combine the data from "y"
   * into _this_ object.
   */
  virtual void threadJoin(const UserObject & y) override;
  /**
   * Called _once_ after execute has been called all all "objects".
   */
  virtual void finalize() override;
protected:
  // This map will hold the partial sums for each block
  std::map<SubdomainID, Real> _integral_values;
  // This map will hold the partial volume sums for each block
  std::map<SubdomainID, Real> _volume_values;
  // This map will hold our averages for each block
  std::map<SubdomainID, Real> _average_values;
};
#endif
```

BlockAverageValue UserObject: (C File)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html
#include "BlockAverageValue.h"
#include "MooseMesh.h"
#include "libmesh/mesh_tools.h"
registerMooseObject("ExampleApp", BlockAverageValue);
template <>
InputParameters
validParams<BlockAverageValue>()
{
  InputParameters params = validParams<ElementIntegralVariablePostprocessor>();
  // Since we are inheriting from a Postprocessor we override this to make sure
  // That MOOSE (and Peacock) know that this object is _actually_ a UserObject
  params.set<std::string>("built_by_action") = "add_user_object";
  return params;
}
```

BlockAverageValue UserObject: (C File cont.)

```
BlockAverageValue::BlockAverageValue(const InputParameters & parameters)
  : ElementIntegralVariablePostprocessor(parameters)
{
}
Real
BlockAverageValue::averageValue(SubdomainID block) const
{
  // Note that we can't use operator[] for a std::map in a const function!
  if (_average_values.find(block) != _average_values.end())
    return _average_values.find(block)->second;
  mooseError("Unknown block requested for average value!");
  return 0; // To satisfy compilers
}
void
BlockAverageValue::initialize()
{
  // Explicitly call the initialization routines for our base class
  ElementIntegralVariablePostprocessor::initialize();
  // Set averages to 0 for each block
  const std::set<SubdomainID> & blocks = _subproblem.mesh().meshSubdomains();
  for (std::set<SubdomainID>::const_iterator it = blocks.begin(); it != blocks.end(); ++it)
  {
    _integral_values[*it] = 0;
    _volume_values[*it] = 0;
    _average_values[*it] = 0;
  }
}
```

BlockAverageValue UserObject: (C File cont.)

```
void
BlockAverageValue::execute()
{
  // Compute the integral on this element
  Real integral_value = computeIntegral();
  // Add that value to the others we've computed on this subdomain
  _integral_values[_current_elem->subdomain_id()] += integral_value;
  // Keep track of the volume of this block
  _volume_values[_current_elem->subdomain_id()] += _current_elem_volume;
}
```

BlockAverageValue UserObject: (C File cont.)

```
void
BlockAverageValue::threadJoin(const UserObject & y)
{
  ElementIntegralVariablePostprocessor::threadJoin(y);
  // We are joining with another class like this one so do a cast so we can get to it's data
  const BlockAverageValue & bav = dynamic_cast<const BlockAverageValue &>(y);
  for (std::map<SubdomainID, Real>::const_iterator it = bav._integral_values.begin();
       it != bav._integral_values.end();
       ++it)
    _integral_values[it->first] += it->second;
  for (std::map<SubdomainID, Real>::const_iterator it = bav._volume_values.begin();
       it != bav._volume_values.end();
       ++it)
    _volume_values[it->first] += it->second;
  for (std::map<SubdomainID, Real>::const_iterator it = bav._average_values.begin();
       it != bav._average_values.end();
       ++it)
    _average_values[it->first] += it->second;
}
```

BlockAverageValue UserObject: (C File cont.)

```
void
BlockAverageValue::finalize()
{
  // Loop over the integral values and sum them up over the processors
  for (std::map<SubdomainID, Real>::iterator it = _integral_values.begin();
       it != _integral_values.end();
       ++it)
    gatherSum(it->second);
  // Loop over the volumes and sum them up over the processors
  for (std::map<SubdomainID, Real>::iterator it = _volume_values.begin();
       it != _volume_values.end();
       ++it)
    gatherSum(it->second);
  // Now everyone has the correct data so everyone can compute the averages properly:
  for (std::map<SubdomainID, Real>::iterator it = _average_values.begin();
       it != _average_values.end();
       ++it)
  {
    SubdomainID id = it->first;
    _average_values[id] = _integral_values[id] / _volume_values[id];
  }
}
```

BlockAverageDiffusionMaterial Material

The second step is to create the Material object that will utilize the block average value for computing a diffusion coefficient. The complete header and source file for this custom Material object are linked below, which includes comments detailing the functionality of the Material. This class simply creates a Material object that creates a material property, "diffusivity", that is computed by the BlockAverageValue class and accessed vial the averageValue method.

BlockAverageDiffusionMaterial Material: (Header File)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html
#ifndef BLOCKAVERAGEDIFFUSIONMATERIAL_H
#define BLOCKAVERAGEDIFFUSIONMATERIAL_H
#include "Material.h"
#include "BlockAverageValue.h"
// Forward Declarations
class BlockAverageDiffusionMaterial;
template <>
InputParameters validParams<BlockAverageDiffusionMaterial>();
```

BlockAverageDiffusionMaterial Material: (Header File cont.)

```cpp
class BlockAverageDiffusionMaterial : public Material
{
public:
  BlockAverageDiffusionMaterial(const InputParameters & parameters);
protected:
  virtual void computeQpProperties() override;
private:
  /**
   * This is the member reference that will hold the computed values
   * for the Real value property in this class.
   */
  MaterialProperty<Real> & _diffusivity;
  /**
   * A member reference that will hold onto a UserObject
   * of type BlockAverageValue for us to be able to query
   * the average value of a variable on each block.
   *
   * NOTE: UserObject references are _const_!
   */
  const BlockAverageValue & _block_average_value;
};
#endif // BLOCKAVERAGEDIFFUSIONMATERIAL_H
```

BlockAverageDiffusionMaterial Material: (C File)

```c
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html
#include "BlockAverageDiffusionMaterial.h"
registerMooseObject("ExampleApp", BlockAverageDiffusionMaterial);
template <>
InputParameters
validParams<BlockAverageDiffusionMaterial>()
{
  InputParameters params = validParams<Material>();
  // UserObjectName is the MOOSE type used for getting the name of a
  // UserObject from the input file
  params.addRequiredParam<UserObjectName>(
      "block_average_userobject",
      "The name of the UserObject that is going to be computing the "
      "average value of a variable on each block");
  return params;
}
```

BlockAverageDiffusionMaterial Material: (C File cont.)

```
BlockAverageDiffusionMaterial::BlockAverageDiffusionMaterial(const InputParameters & parameters)
  : Material(parameters),
    // Declare that this material is going to provide a Real
    // valued property named "diffusivity" that Kernels can use.
    _diffusivity(declareProperty<Real>("diffusivity")),
    // When getting a UserObject from the input file pass the name
    // of the UserObjectName _parameter_
    // Note that getUserObject returns a _const reference_ of the type in < >
    _block_average_value(getUserObject<BlockAverageValue>("block_average_userobject"))
{
}
```

BlockAverageDiffusionMaterial Material: (C File cont.)

```
void
BlockAverageDiffusionMaterial::computeQpProperties()
{
  // We will compute the diffusivity based on the average value of the variable on each block.
  // We'll get that value from a UserObject that is computing it for us.
  // To get the current block number we're going to query the "subdomain_id()" of the current
  // element
  _diffusivity[_qp] = 0.5 * _block_average_value.averageValue(_current_elem->subdomain_id());
}
```

ExampleDiffusion Kernel

In order to utilize the "diffusivity" material property a Kernel that uses a material property as a coefficient is required. This is accomplished by creating a new Kernel, in this case a Kernel that inherits from the MOOSE Diffusion Kernel. This newly created Kernel simply multiplies the Diffusion Kernel computeQpResidual() and computeQpJacobian() methods with a material property. The complete code for this custom Kernel is supplied in the links below, again the comments in the source detail the behavior of the class.

ExampleDiffusion Kernel: (Header File)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html
#ifndef EXAMPLEDIFFUSION_H
#define EXAMPLEDIFFUSION_H
#include "Diffusion.h"
// Forward Declarations
class ExampleDiffusion;
/**
 * validParams returns the parameters that this Kernel accepts / needs
 * The actual body of the function MUST be in the .C file.
 */
template <>
InputParameters validParams<ExampleDiffusion>();
```

ExampleDiffusion Kernel: (Header File cont.)

```
class ExampleDiffusion : public Diffusion
{
public:
  ExampleDiffusion(const InputParameters & parameters);
protected:
  virtual Real computeQpResidual() override;
  virtual Real computeQpJacobian() override;
  /**
   * This MooseArray will hold the reference we need to our
   * material property from the Material class
   */
  const MaterialProperty<Real> & _diffusivity;
};
#endif // EXAMPLEDIFFUSION_H
```

ExampleDiffusion Kernel: (C File)

```
//* This file is part of the MOOSE framework
//* https://www.mooseframework.org
//*
//* All rights reserved, see COPYRIGHT for full restrictions
//* https://github.com/idaholab/moose/blob/master/COPYRIGHT
//*
//* Licensed under LGPL 2.1, please see LICENSE for details
//* https://www.gnu.org/licenses/lgpl-2.1.html
#include "ExampleDiffusion.h"
/**
 * This function defines the valid parameters for
 * this Kernel and their default values
 */
registerMooseObject("ExampleApp", ExampleDiffusion);
template <>
InputParameters
validParams<ExampleDiffusion>()
{
  InputParameters params = validParams<Diffusion>();
  return params;
}
ExampleDiffusion::ExampleDiffusion(const InputParameters & parameters)
  : Diffusion(parameters), _diffusivity(getMaterialProperty<Real>("diffusivity"
{
}
```

ExampleDiffusion Kernel: (C File cont.)

```
Real
ExampleDiffusion::computeQpResidual()
{
  // We're dereferencing the _diffusivity pointer to get to the
  // material properties vector... which gives us one property
  // value per quadrature point.
  // Also... we're reusing the Diffusion Kernel's residual
  // so that we don't have to recode that.
  return _diffusivity[_qp] * Diffusion::computeQpResidual();
}
Real
ExampleDiffusion::computeQpJacobian()
{
  // We're dereferencing the _diffusivity pointer to get to the
  // material properties vector... which gives us one property
  // value per quadrature point.
  // Also... we're reusing the Diffusion Kernel's residual
  // so that we don't have to recode that.
  return _diffusivity[_qp] * Diffusion::computeQpJacobian();
}
```

Input File

```
[Mesh]
  file = two_squares.e
  dim = 2
[]
[Variables]
  [./u]
    initial_condition = 0.01
  [../]
[]
[Kernels]
  [./diff]
    type = ExampleDiffusion
    variable = u
  [../]
  [./td]
    type = TimeDerivative
    variable = u
  [../]
[]
[BCs]
  [./left]
    type = DirichletBC
    variable = u
    boundary = leftleft
    value = 0
  [../]
  [./right]
    type = DirichletBC
    variable = u
    boundary = rightright
    value = 1
  [../]
[]
```

```
[Materials]
  [./badm]
    type = BlockAverageDiffusionMaterial
    block = 'left right'
    block_average_userobject = bav
  [../]
[]
[UserObjects]
  [./bav]
    type = BlockAverageValue
    variable = u
    execute_on = timestep_begin
    outputs = none
  [../]
[]
[Executioner]
  type = Transient
  num_steps = 10
  dt = 1
  #Preconditioned JFNK (default)
  solve_type = 'PJFNK'
  petsc_options_iname = '-pc_type -pc_hypre_type'
  petsc_options_value = 'hypre boomeramg'
[]
[Outputs]
  exodus = true
[]
```

Running the Problem
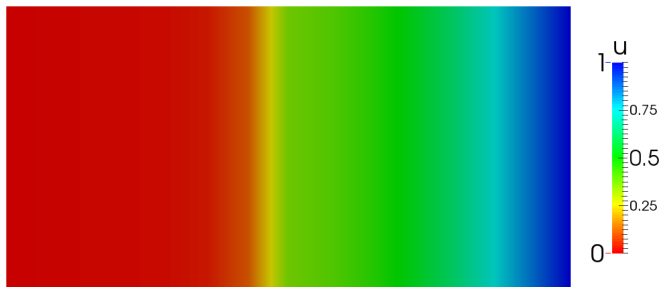
Example 20 Results after 4 timesteps.
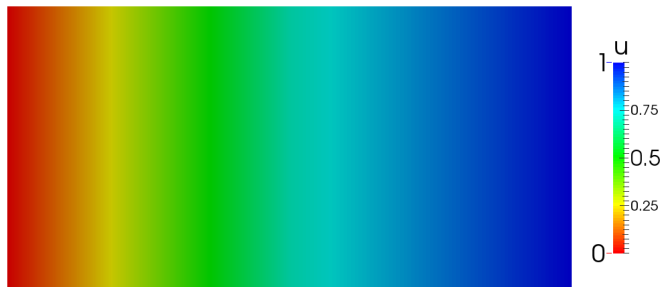
Example 20 Results after 10 timesteps.

Restart and Recovery (Advanced)

Restart and Recovery (Advanced) Contents

Definitions

- **Restart**: Running a simulation that uses data from a previous simulation. Data in this context is very broad, it can mean spatial field data, non-spatial variables or postprocessors, or stateful object data. Usually the previous and new simulations use different input files.

- **Recover**: Resuming an existing simulation either due to a fault or other premature termination.

- **Solution File**: A mesh format containing field data in addition to the mesh (i.e., a normal output file).

- **Checkpoint**: A snapshot of the simulation data including all meshes, solutions, and stateful object data. Typically one checkpoint is stored in several different files.

- **N to N**: In a restart context, this means the number of processors for the previous and current simulations must match.

- **N to M**: In a restart context, different numbers of processors may

Variable Initialization

- This method is best suited for restarting a simulation when the mesh in the previous simulation exactly matches the mesh in the current simulation and only initial conditions need to be set for one or more variables.

- This method requires only a valid Solution File.

- MOOSE supports N to M restart when using this method.

Variable Initialization (cont.)

```
[Mesh]
  #MOOSE supports reading field data from ExodusII, XDA/XDR, and
  #mesh checkpoint files (.e, .xda, .xdr, .cp)
  file = previous.e
  #This method of restart is only supported on serial meshes
  distribution = serial
[]

[Variables]
  [./nodal]
    family = LAGRANGE
    order = FIRST
    initial_from_file_var = nodal
    initial_from_file_timestep = 10
  [../]
[]

[AuxVariables]
  [./elemental]
    family = MONOMIAL
    order = CONSTANT
    initial_from_file_var = elemental
    initial_from_file_timestep = 10
  [../]
[]
```

Enabling Checkpoints

- Advanced restart in MOOSE requires checkpoint files.

- To enable automatic checkpoints using the default options (every time step, and keep last two) in your simulation simply add the following flag to your input file:

```
[Outputs]
  checkpoint = true
[]
```

Enabling Checkpoints (cont.)

- ▶ If you need more control over the checkpoint system, you can create a subblock in the input file that will allow you to change the file format, suffix, frequency of output, the number of checkpoint files to keep, etc.

- ▶ For a complete list see the Doxygen page for Checkpoint.

- ▶ You should always set num_files to at least 2 to minimize the chance of ending up with a corrupt restart file.

```
[Outputs]
  [./my_checkpoint]
    type = Checkpoint
    num_files = 4
    interval = 5
  [../]
[]
```

Advanced Restart

- ▶ This method is best suited for situations when the mesh from the previous simulation and the current simulation match but all variables should be reloaded and all stateful data should be restored.

- ▶ Support for modifying some variables is supported such as dt and time_step. By default, MOOSE will automatically use the last values found in the checkpoint files.

- ▶ Only N to N restarts are supported using this method.

```
[Mesh]
  #Serial number should match corresponding Executioner parameter
  file = out_cp/0010_mesh.cpr
  #This method of restart is only supported on serial meshes
  distribution = serial
[]

[Problem]
  #Note that the suffix is left off in the parameter below.
  restart_file_base = out_cp/LATEST  # You may also use a specific number here.
[]
```

Reloading Data

- It is possible to load and project data onto a different mesh from a solution file usually as an initial condition in the new simulation.

- MOOSE fully supports this through the use of SolutionUserObject.

Recover

- A simulation that has terminated due to a fault can be recovered simply by using the `--recover` CLI flag.

- Requires checkpoint files.

MultiApp Restart

When running a MultiApp simulation you do not need to enable checkpoint output in each sub app input file. The master app stores the restart data for all sub apps in its file.

Controls (Advanced)

Controls (Advanced) Contents

Creating a Controllable Parameter

The control system in MOOSE has one primary purpose: **to modify input parameters during runtime of a MOOSE-based simulation**. The input parameters of objects you wish to be controlled must:

▶ Store parameter as a `const` reference; in your *.h files, declare storage for the parameters as follows:

```
const Real & _value;
```

▶ Initialize the reference in the *.C file as follows:

```
: NodalBC(parameters), _value(getParam<Real>("value"))
```

Creating a Controllable Parameter (cont.)

In order to "control" a parameter it must be communicated that
the parameter is allowed to be controlled. This is done in the
validParams function as in the code below. The input can be a
single value or a space separated list of parameters.

```
template <>
InputParameters
validParams<DirichletBC>()
{
  InputParameters p = validParams<NodalBC>();
  p.addRequiredParam<Real>("value", "Value of the BC");
  p.declareControllable("value");
  p.addClassDescription("Imposes the essential boundary condition $u=g$, where $g$ "
                        "is a constant, controllable value.");

  return p;
}
```

NOTE: The declareControllable method also accepts a space
separated list of parameters.

Create a Control Object

`Control` objects are similar to other systems in MOOSE. You create a control in your application by inheriting from the `Control` C++ class in MOOSE. It is required to override the `execute` method in your custom object. Within this method the following methods are generally used to get or set controllable parameters:

▶ `getControllableValue`
This method returns the current controllable parameter. In the case that multiple parameters are being controlled, only the first value will be returned and a warning will be produced if the values are different.

▶ `setControllableValue`
This method allows for a controllable parameter to be changed. In the case that multiple parameters are being controlled, all of the values will be set.

Create a Control Object (cont.)

- These methods operate in a similar fashion as the other systems in MOOSE (e.g., `getPostprocessorValue` in the Postprocessors system). Each expects an input parameter name (`std::string`) that is prescribed in the `validParams` method.

- There are additional overloaded methods that allow for the setting and getting of controllable values with various inputs for prescribing the parameter name, but the two listed above are generally what is needed. Please refer to the source code for a complete list.

Controls Block

Control objects are defined in the input file in the Controls block, similar to other systems in MOOSE. For example, the following input file snippet shows the use of the RealFunctionControl object.

```
[Controls]
  [./func_control]
    type = RealFunctionControl
    parameter = '*/*/coef'
    function = 'func_coef'
    execute_on = 'initial timestep_begin'
  [../]
[]
```

Object and Parameter Names

Notice that, in the previous slide, the syntax for specifying a parameter is shown. In general, the syntax for a parameter name is specified as:
`block/object/name`

- `block`: specifies the input file block name (e.g., "Kernels", "BCs").

- `object`: specifies the input file sub-block name (e.g., "diff" in the snippet below).

- `name`: specifies the parameter name (e.g., "coef" in the snippet below).

```
[Kernels]
  [./diff]
    type = CoefDiffusion
    variable = u
    coef = 0.1
  [../]
  [./time]
    type = TimeDerivative
    variable = u
  [../]
[]
```

Object and Parameter Names (cont.)

- As shown in the `Controls` block snippet, an asterisk (*) can be substituted for any one of these three "names". Doing so allows multiple parameters to match and be controlled simultaneously.

- In similar fashion, object names can be defined (e.g., as in the <u>TimePeriod</u> object). In this case, the general name scheme is the same as above, but the parameter name is not included.

- In both cases there is an alternative form for defining an object and parameter names: `base::object/name`. In this case "base" is the MOOSE base system that the object is derived from. For example, `Kernel::diff/coef`.

MultiApps (Advanced)

MultiApps (Advanced) Contents

MultiApps

- MOOSE is capable of running multiple applications together and transfer data between the various applications.

- 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.
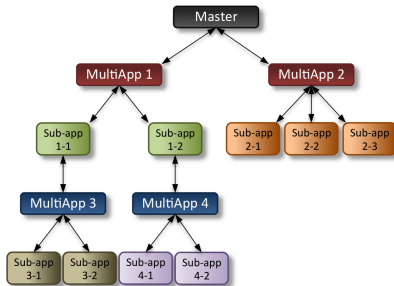
Input File Syntax

- `MultiApps` are declared in the `MultiApps` block.

- They require a `type` just like any other block.

- `app_type` is required and it 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_end`.

- `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.

Input File Syntax (cont.)

```
[MultiApps]
  [./some_multi]
    type = TransientMultiApp
    app_type = SomeApp
    execute_on = timestep_end
    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'
  [../]
[]
```

Dynamically Loading MultiApps

- ▶ If you are building with dynamic libraries (the default) you may load other applications without explicitly adding them to your MakeFile and registering them.

- ▶ Simply set the proper `app_type` in your input file (e.g. `AnimalApp`) and MOOSE attempts to find the other library dynamically.

- ▶ You may specify a path (relative preferred) in your input file using the parameter `library_path`. This path needs to point to the `lib` folder underneath your application.

- ▶ You may also set an environment variable for paths to search: `MOOSE_LIBRARY_PATH`

- ▶ Note: You will need to compile each application separately since the Makefile does not have any knowledge of the dependent application.

TransientMultiApp

- Some of the currently-available MultiApp types are `TransientMultiApp` and `FullSolveMultiApp`.

- 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.

- By default, the minimum `dt` over the master and all sub-apps is used.

- However, we have the ability to do "sub-cycling", which allows a sub-app to take multiple time steps during a single master app time step.

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.
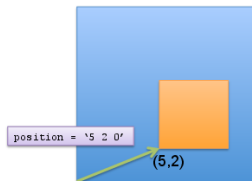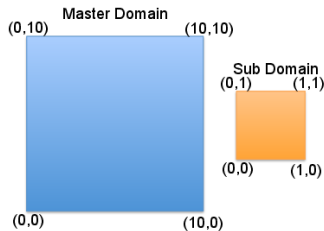
Parallel
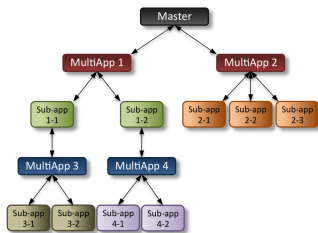
- ▶ The `MultiApp` system is designed for efficient parallel execution of hierarchical problems.

- ▶ 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

Transfers

- While a `MultiApp` allows you to execute many solves in parallel, it doesn't allow for data to flow in these 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.

Example Problem Information

▶ For an example of using the multiapp system for combining microscale and engineering scale calculations for a Darcy thermomechanical flow scenario, please see the `MultiApps` example in the MOOSE repository in tutorials/darcy_thermo_mech/step10_multiapps