C3 Summer Boot Camp

Sponsored by Center For Advanced Energy Studies

Computational Modeling and Data Science Workshops

Introduction to the MOOSE Framework

Min Long

Department of Computer Science, Boise State

2020/06/10

Online Helpers: Dr. Larry Aagesen (INL), Andy Lau (BSU)

https://nanohub.org/groups/caesinlc3compute/workshop_page_or_topic_page

CAES Center for Advanced Energy Studies

A COLLABORATION BETWEEN

INL Idaho National Laboratory

BOISE STATE UNIVERSITY

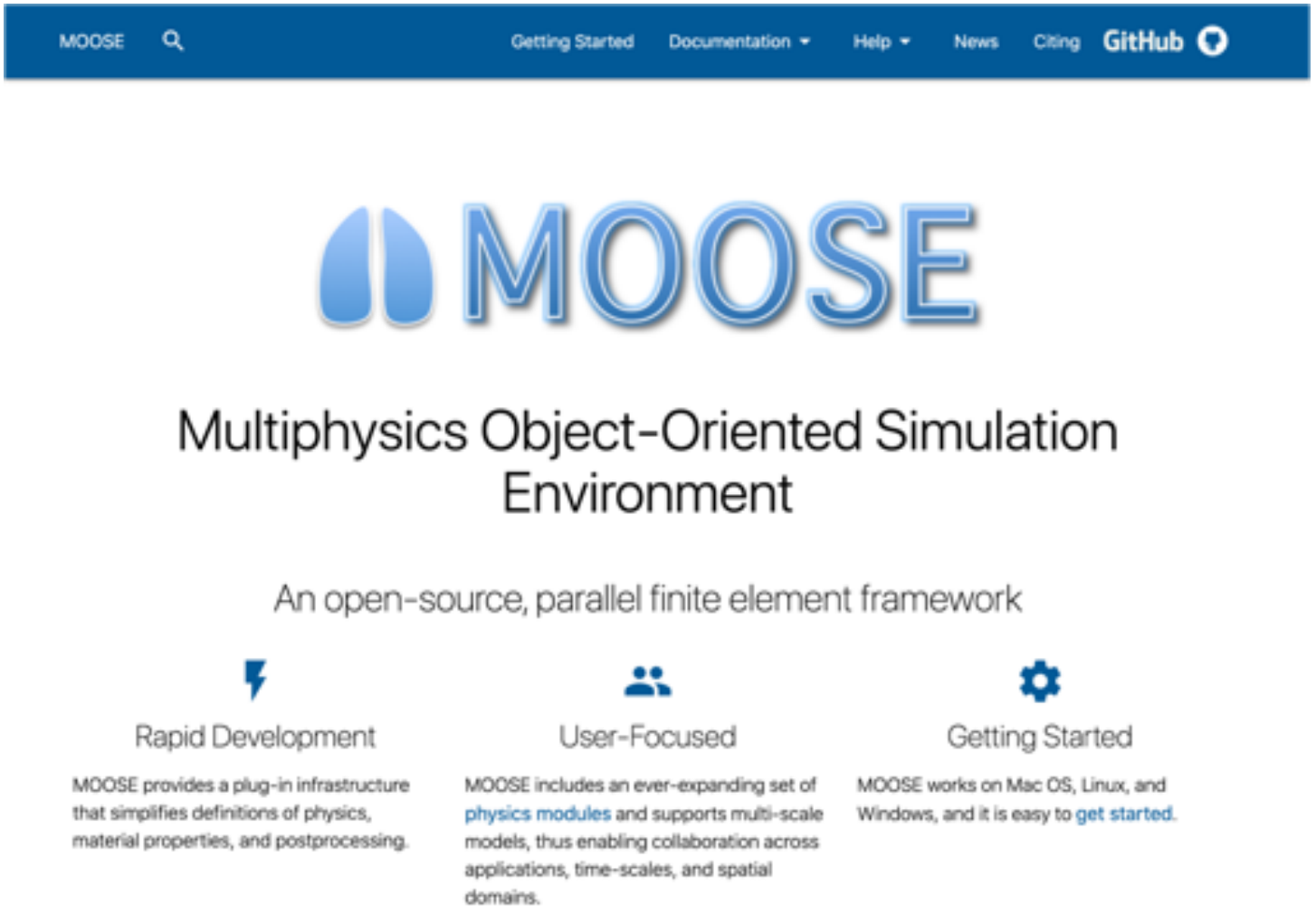
Idaho State UNIVERSITY

University of Idaho

UNIVERSITY OF WYOMING

Reference

- http://mooseframework.org/



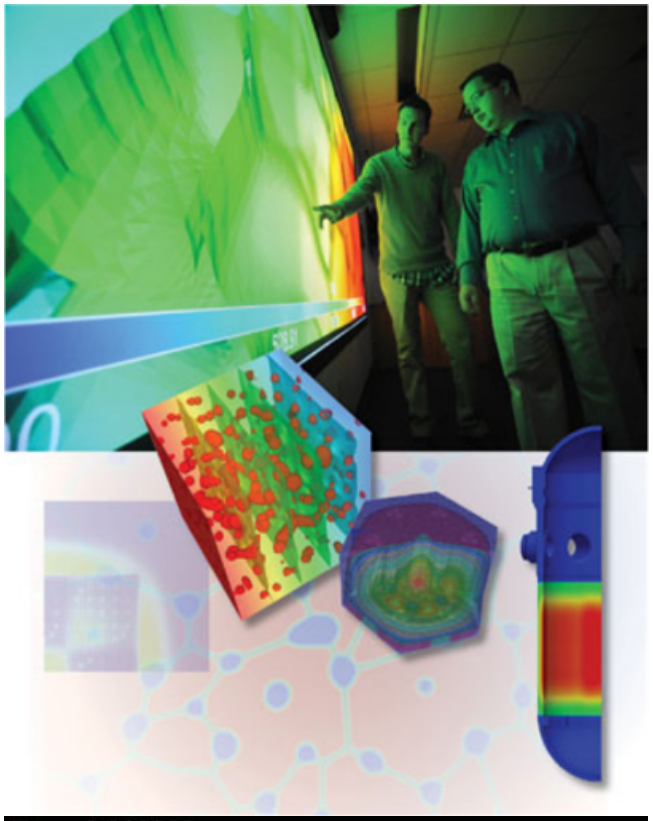
Outline

- MOOSE Overview
- Installation
- A First Example of Diffusion Problem
- Finite Element Principles
- Input Files
- Examples with a more complex geometry
- An Advanced Example: DarcyFlow
- MOOSE on the NanoHub

MOOSE Overview

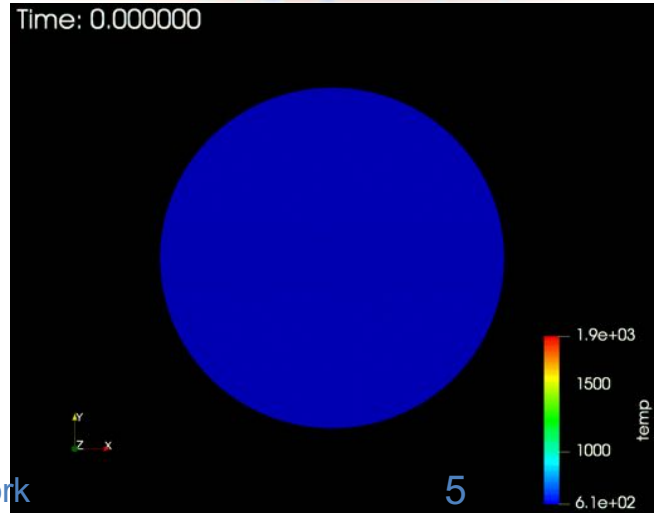
MOOSE: a tool for solving PDEs with FEM

- It provides a high-level interface to **PETSc** and **massively parallel computational capability** since 2008
- It is **open source** and freely available at **mooseframework.org** since 2014
- It is maintained and developed by a team of full time staff at Idaho National Laboratory



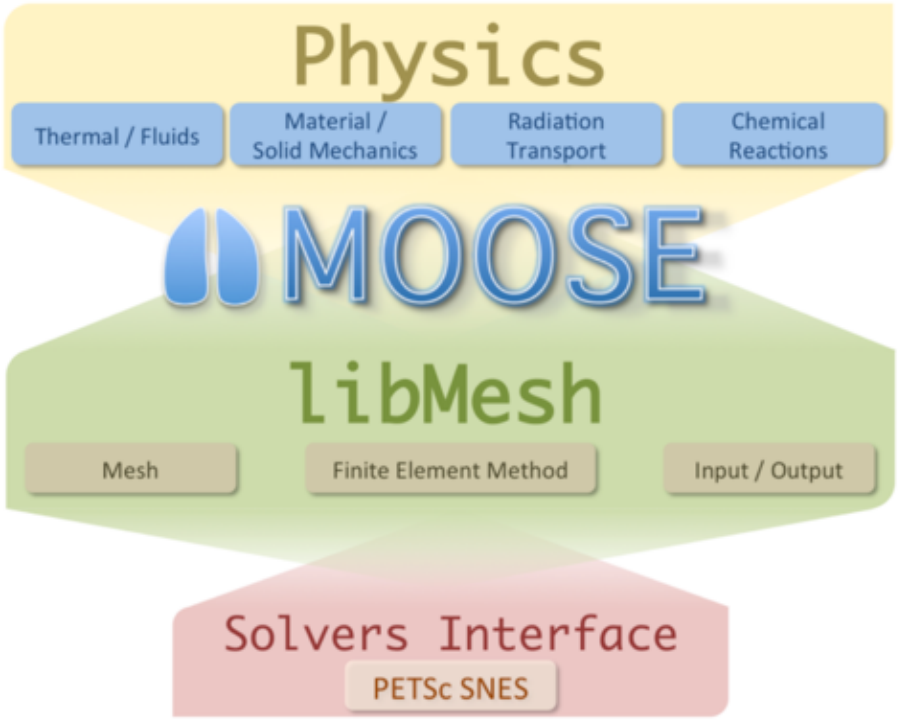
MOOSE is in use across the world to solve:

- Phase field
- Solid mechanics
- Heat conduction
- Neutronics
- Comp. fluid dynamics
- Geomechanics
- Reactive transport
- Corrosion
- Crystal plasticity
- Fracture



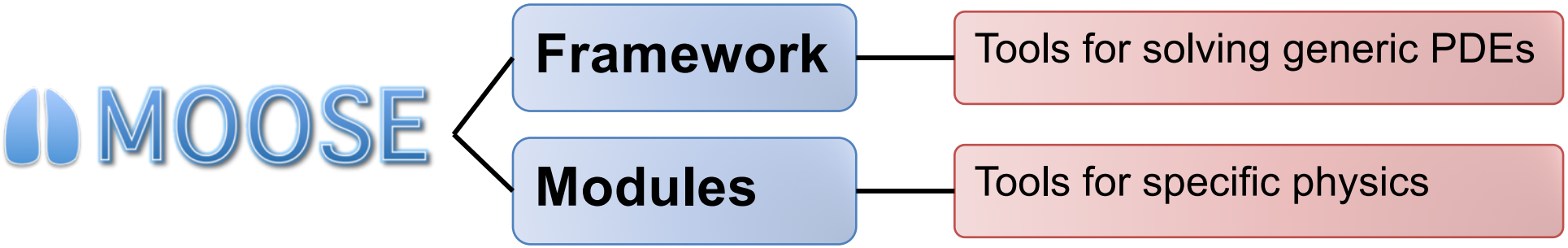
Powerful Capabilities

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



- Uses well-established libraries
  - MPI, PETSc, libmesh
- Implements robust and state-of-the-art solution methods

Physics Modules in MOOSE



- Chemical reactions
- Contact
- Heat conduction
- Level set
- Navier-Stokes
- Peridynamics
- Phase field
- Porous flow
- Tensor mechanics
- Water/steam EOS
- XFEM

Child Applications in MOOSE

- BISON – Macroscale fuel performance code
- MARMOT – Mesoscale code for computational nuclear materials
- RATTLESNAKE – Radiation transport code
- RELAP-7 – Nuclear reactor thermal hydraulics code
- FALCON – Geomechanics code under development in the US and Australia
- And many more …

MOOSE is used all across the World

11

Installation

System Requirements

- GNU/Linux/MacOS Requirements:
  - GCC/Clang C++11 compiler (GCC > 4.8.4, or Clang > 3.5.1)
    - Intel Compilers are not supported
  - Memory (> 8 GB)
  - Processor (64bit x86)
  - Disk (> 30GB)

- Using MOOSE on Windows 10 is experimental and not fully supported.

1. Install GNU Compiler Collection

- Mac: Xcode command line tools

```
[long@Helix:moose]$ xcode-select --install
```

- Ubuntu: build-essential

```
mlong@ENG401524:~$ sudo apt-get install build-essential
[sudo] password for mlong:
Reading package lists... Done
Building dependency tree
Reading state information... Done
build-essential is already the newest version (12.4ubuntu1).
The following packages were automatically installed and are no longer required:
  app-install-data apt-xapian-index cdrdao gcc-6-base gconf-service
  gconf-service-backend gconf2-common libappindicator1 libaprutil1-dbd-mysql
  libaprutil1-ldap libdbusmenu-gtk4 libgconf-2-4 libgtkmm-3.0-1v5
  libindicator7 libmime-charset-perl libmysqlclient20 libnih-dbus1
  libperl4-corelibs-perl libqt4-designer libqt4-help libqt4-scripttools
  libqt4-svg libqt4-test libqtassistantclient4 librsync1 libsombok3
  libunicode-linebreak-perl mysql-common python-apt python-aptdaemon
  python-aptdaemon.gtk3widgets python-asn1crypto python-attr python-automat
  python-blinker python-cairo python-cffi-backend python-click python-colorama
  python-constantly python-cryptography python-cups python-dbus python-debian
  python-debtagshw python-defer python-dirspec python-enum34 python-gi
  python-gi-cairo python-httplib2 python-hyperlink python-idna python-imaging
  python-incremental python-ipaddress python-jwt python-ldb python-lockfile
  python-oauthlib python-olefile python-openssl python-pam python-pil
  python-piston-mini-client python-pyasn1 python-qt4 python-qt4-dbus
  python-serial python-sip python-tdb python-twisted-bin python-xapian
  python-xdg python-zope.interface python3-piston-mini-client python3-xapian
  ruby-minitest ruby-power-assert software-center-aptdaemon-plugins
  texlive-font-utils texlive-pictures texlive-pictures-doc
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 235 not upgraded.
```

2. Install Conda Environment

- Conda is an open source package management system and environment management system of packages and their dependencies.
- MOOSE preferred to obtain support from Conda's libraries.
- Install Miniconda (or Anaconda)
  - Linux

```
curl -L -O https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh -b -p ~/miniconda3
```

  - Mac

```
curl -L -O https://repo.anaconda.com/miniconda/Miniconda3-latest-MacOSX-x86_64.sh
bash Miniconda3-latest-MacOSX-x86_64.sh -b -p ~/miniconda3
```

  - Or download the installer from
    - https://docs.conda.io/projects/conda/en/latest/user-guide/install/macos.html

3. Create a Conda MOOSE Environment

- Export PATH

```
$ export PATH=$HOME/miniconda3/bin:$PATH
```

  - Or edit your .bashrc or .profile and add the above line and run command

```
$ . ~/.profile
```

- Configure Conda to work with conda-forge, and our mooseframework.org channel:

```
$ conda config --add channels conda-forge
$ conda config --add channels https://mooseframework.org/conda/moose
```

- Install the moose-libmesh and moose-tools package from mooseframework.org

```
$ conda create --name moose moose-libmesh moose-tools
```

- Activate MOOSE environment

```
$ conda activate moose
```

3b. Manage Conda MOOSE Environment

- Update Conda MOOSE environment

```
$ conda update --all
```

- Deactivate Conda MOOSE environment

```
$ conda deactivate
```

- Uninsall Conda MOOSE environment

```
$ conda remove --name moose --all
```

4. Install MOOSE

- Prepare a working folder

```
$ mkdir ~/projects
$ cd ~/projects
```

- MOOSE is hosted on GitHub and should be cloned directly from there using git

```
(moose) [long@Helix:moose]$ git clone https://github.com/idaholab/moose.git
Cloning into 'moose'...
remote: Enumerating objects: 110, done.
remote: Counting objects: 100% (110/110), done.
remote: Compressing objects: 100% (76/76), done.
Receiving objects: 100% (431533/431533), 316.67 MiB | 1.39 MiB/s, done.
remote: Total 431533 (delta 34), reused 62 (delta 15), pack-reused 431423
Resolving deltas: 100% (328360/328360), done.
Updating files: 100% (25007/25007), done.
```

- Switch to a master branch

```
$ cd moose
$ git checkout master
```

5. Compile and Run a Test

- Compile

```
$ cd ~/projects/moose/test
$ make -j 4
```

- Run

```
$ ./run_tests -j 4
```

- A successful installation will present

```
time_integrators/convergence.implicit_astabledirk4_bootstrap/level0 ..................................... OK
time_integrators/convergence.implicit_astabledirk4_bootstrap/level1 ..................................... OK
time_integrators/convergence.implicit_astabledirk4_bootstrap/level2 ..................................... OK
mesh/nemesis.nemesis_repartitioning_test ................................................... [min_cpus=4] OK
outputs/nemesis.nemesis_elemental_replicated ............................................... [min_cpus=4] OK
outputs/nemesis.nemesis_scalar_replicated .................................................. [min_cpus=4] OK
partitioners/petsc_partitioner.ptscotch_weight_elment ...................................... [min_cpus=4] OK
partitioners/petsc_partitioner.ptscotch_weight_side ........................................ [min_cpus=4] OK
partitioners/petsc_partitioner.ptscotch_weight_both ........................................ [min_cpus=4] OK
partitioners/petsc_partitioner.parmetis .................................................... [min_cpus=4] OK
partitioners/petsc_partitioner.parmetis_weight_element ..................................... [min_cpus=4] OK
partitioners/petsc_partitioner.parmetis_weight_side ........................................ [min_cpus=4] OK
partitioners/petsc_partitioner.parmetis_weight_both ........................................ [min_cpus=4] OK
partitioners/petsc_partitioner.parmetis_presplit_mesh ...................................... [min_cpus=2] OK
--------------------------------------------------------------------------------------------------------
Ran 2333 tests in 462.2 seconds.
2333 passed, 32 skipped, 0 pending, 0 failed
```

Create your own MOOSE application

Create an Application

- MOOSE is designed for building custom applications
- Go out of the MOOSE repository (~/projects/moose)

```
$ cd ~/projects
```

- Run stock.sh to

```
$ ./moose/scripts/stork.sh panda
```

  - A folder of the application (panda) will be created.
  - The application will automatically link against MOOSE.

- Compile and Test Your Application

```
$ cd panda
$ make -j 4
$ ./run_tests -j 4
```

- Update MOOSE (weekly)

```
$ cd ~/projects/moose
$ git fetch origin
$ git rebase origin/master
```

Application directory structure

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

A First Example of Diffusion Problem

Solve a problem using MOOSE

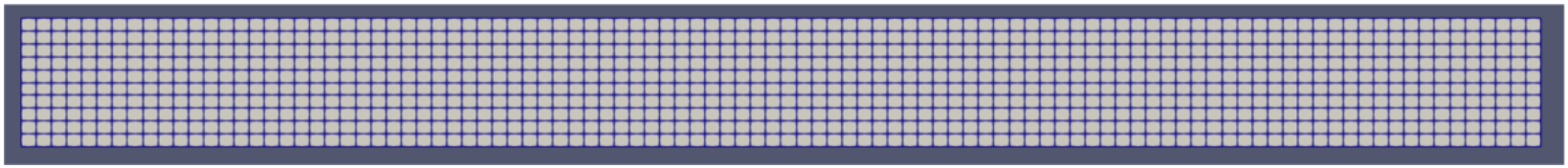
- Pre-processing
- Problem Statement
- Construct or apply objects from MOOSE
- (Post processing)

- Example: steady-state diffusion equation on the domain Ω

$$-\nabla \cdot \nabla u = 0 \in \Omega,$$

Solve a problem using MOOSE

- Pre-processing
  - Construct the suitable geometries



  - Simple mesh can be defined in the input.i

- Problem Statements
  - Strong Form
  - BCs
  - Weak Form

$$-\nabla \cdot \nabla u = 0 \in \Omega,$$

$u = 0$ on the left, $u = 1$ on the right

$\nabla u \cdot \hat{n} = 0$ on the remaining

$$(\nabla \psi_i, \nabla u_h) = 0 \quad \forall \psi_i$$

Compile

- moose/tutorials/darcy_thermo_mech/step01_diffusion

```
[long@Helix:step01_diffusion]$ conda activate moose
(moose) [long@Helix:step01_diffusion]$ make
```

```
(moose) [long@Helix:step01_diffusion]$ cd problems/
(moose) [long@Helix:problems]$ lt
total 208
-rw-r--r--  1 long  staff   1586 Jun  9 04:15 step1.i
-rwxr-xr-x  1 long  staff   1152 Jun  9 04:15 step1.py
-rw-r--r--  1 long  staff    248 Jun  9 04:15 tests
-rw-r--r--  1 long  staff   1507 Jun  9 09:34 peacock_run_exe_tmp_step1.i
```

Input File step1.i

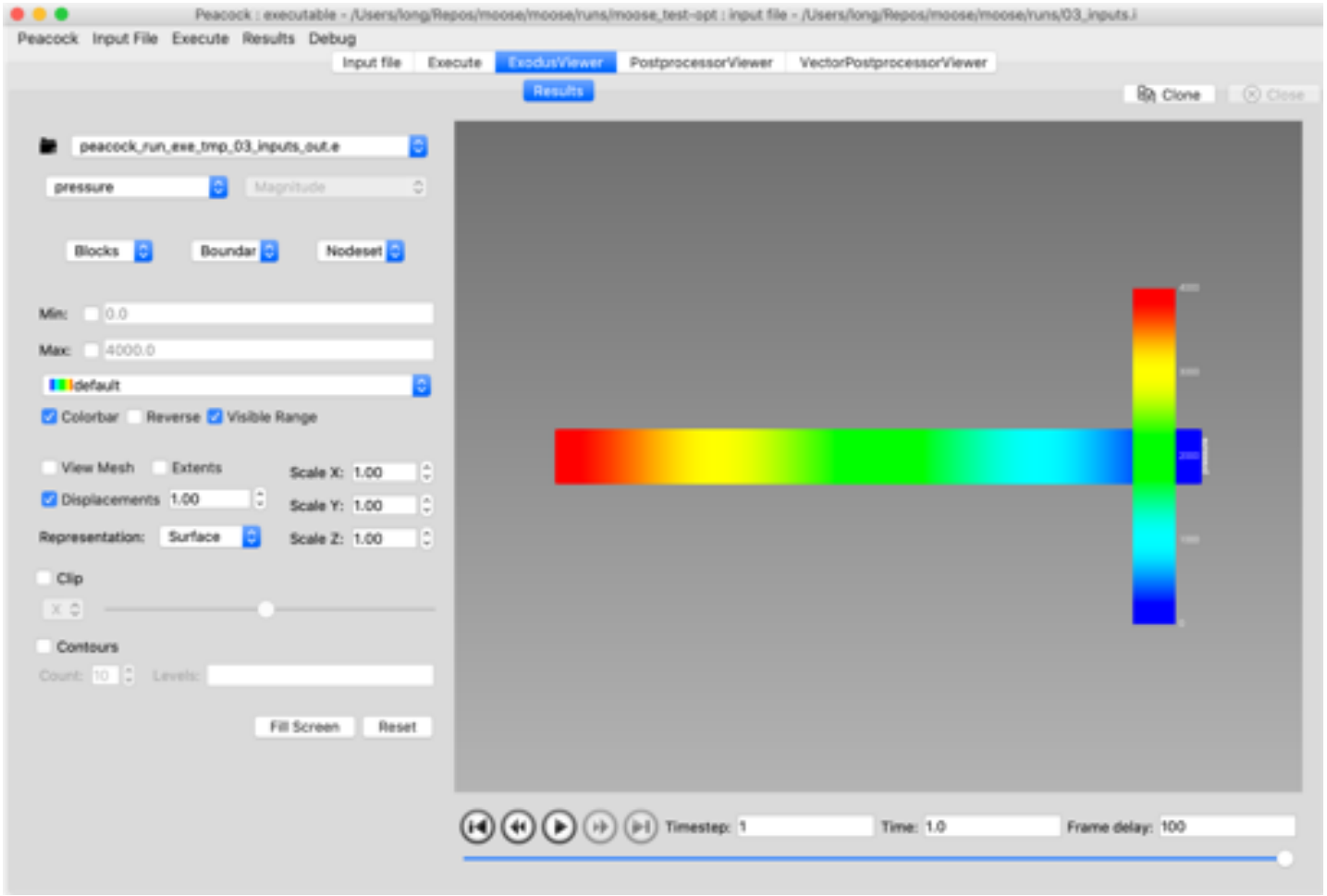
```
[Mesh]
  type = GeneratedMesh # Can generate simple lines,
rectangles and rectangular prisms
  dim = 2              # Dimension of the mesh
  nx = 100             # Number of elements in the x
direction
  ny = 10              # Number of elements in the y
direction
  xmax = 0.304         # Length of test chamber
  ymax = 0.0257        # Test chamber radius
[]


[Variables]
  [pressure]
    # Adds a Linear Lagrange variable by default
  []
[]


[Kernels]
  [diffusion]
    type = ADDiffusion  # Laplacian operator using
automatic differentiation
    variable = pressure # Operate on the "pressure"
variable from above
  []
[]

[Problem]
  type = FEProblem  # This is the "normal" type of
Finite Element Problem in MOOSE
  coord_type = RZ   # Axisymmetric RZ
  rz_coord_axis = X # Which axis the symmetry is around
[]
```

```
[BCs]
  [inlet]
    type = DirichletBC  # Simple u=value BC
    variable = pressure # Variable to be set
    boundary = left     # Name of a sideset in the mesh
    value = 4000        # (Pa) From Figure 2 from
paper.  First data point for 1mm spheres.
  []
  [outlet]
    type = DirichletBC
    variable = pressure
    boundary = right
    value = 0           # (Pa) Gives the correct pressure
drop from Figure 2 for 1mm spheres
  []
[]

[Executioner]
  type = Steady       # Steady state problem
  solve_type = NEWTON # Perform a Newton solve, uses AD
to compute Jacobian terms
  petsc_options_iname = '-pc_type -pc_hypre_type' # PETSc
option pairs with values below
  petsc_options_value = 'hypre boomeramg'
[]


[Outputs]
  exodus = true # Output Exodus format
[]
```

Run the simulation

- Method 1 – MOOSE GUI Peacock

```
$ ~/projects/moose/python/peacock/peacock -i step1.i
```



Run the simulation

- Method 2

```
$ ../darcy_thermo_mech-opt -i step1.i
```

- Method 3 – MPI

```
$ mpiexec -n 4 ../darcy_thermo_mech-opt -i step1.i
```

```
Framework Information:
MOOSE Version:           git commit 0575ba3cd9 on 2020-06-09
LibMesh Version:         6e07d0a45166892bdbeef78440bad7aa46e2a5b7
PETSc Version:           3.12.5
SLEPc Version:           3.12.1

…
…

      2 Linear |R| = 6.991677e-06
      3 Linear |R| = 3.761273e-07
      4 Linear |R| = 1.883098e-08
 2 Nonlinear |R| = 1.883097e-08
 Solve Converged!
```

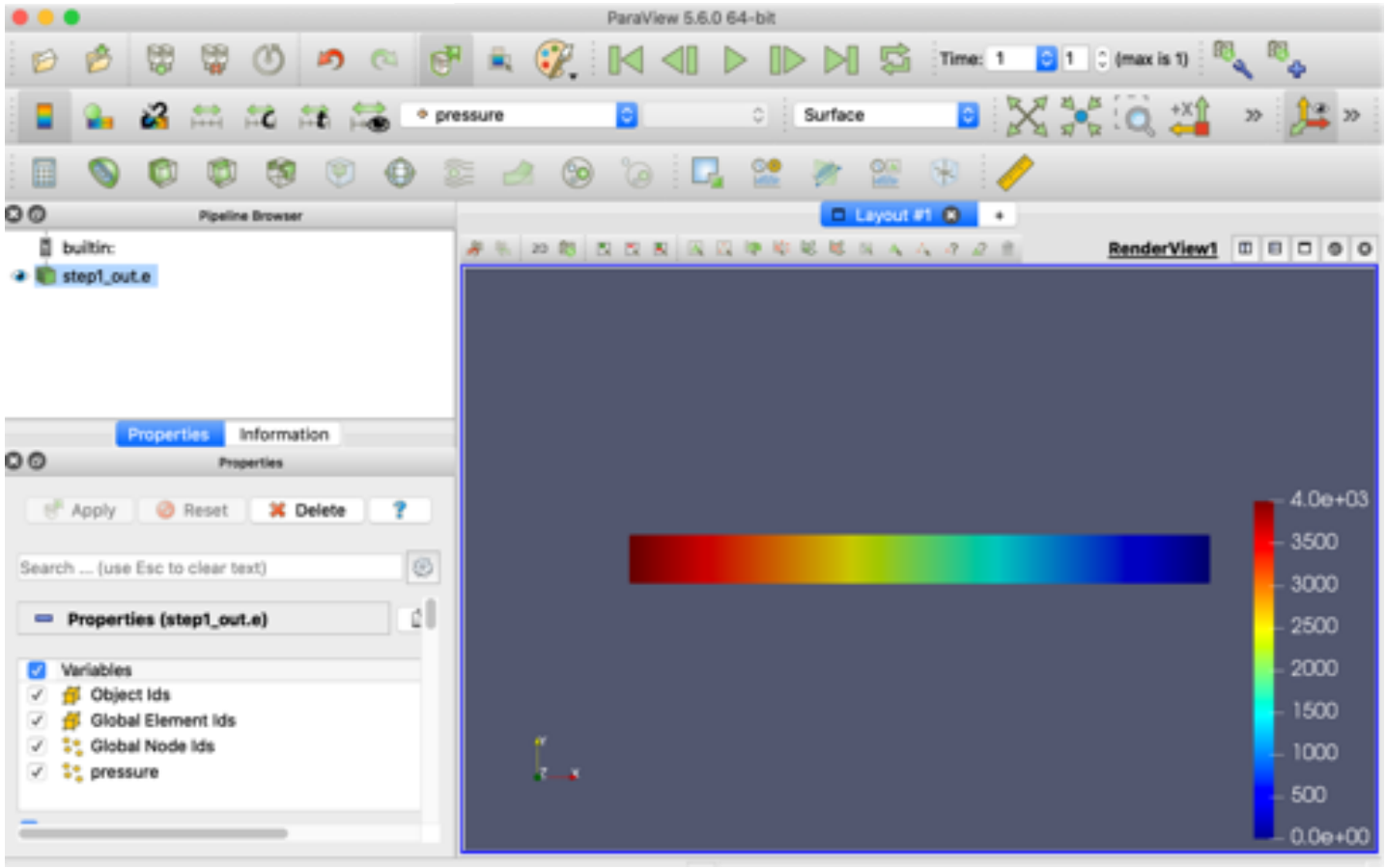
Visualization

- Pecock

```
$ ~/projects/moose/python/peacock/peacock -r step1_out.e
```

- Paraview

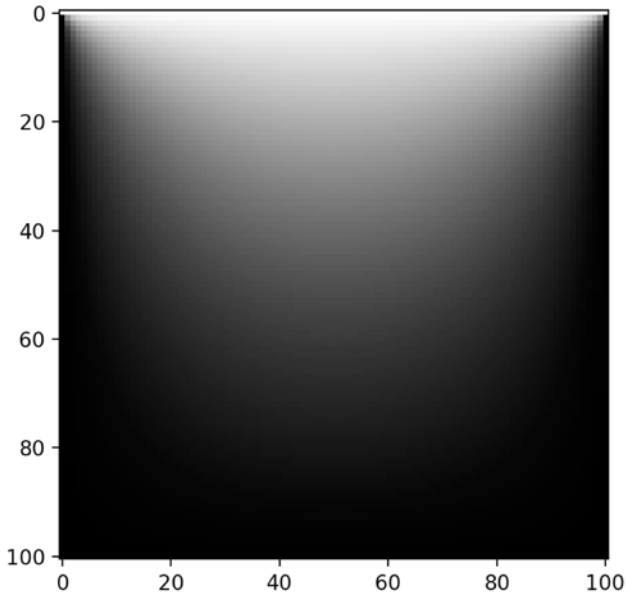
```
$ open step1_out.e
```

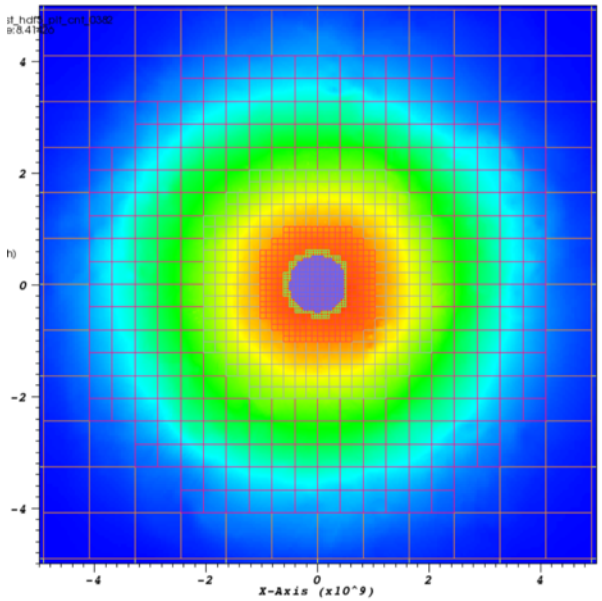


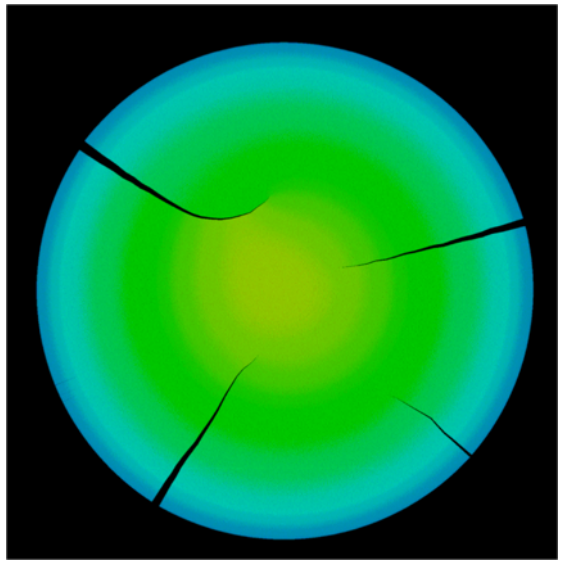
Finite Element Principles

FEM is one of approaches to solve PDEs

- Finite Difference Finite Volume Finite Element







PDEs $\Longrightarrow$ Algebraic Equations

Polynomial Fitting

- Fitting a polynomial to approximate a function f(x)

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

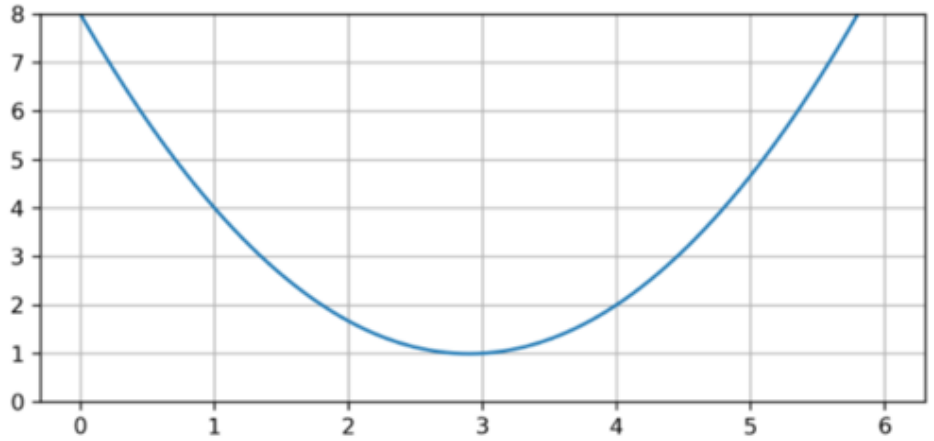
  - f(x) is known at a set of given points x_n
  - Solution would be defined if coefficients are determined
  - Basis functions: $1, x, x^2$
  - In general $f(x) = \sum_{i=0}^{d} c_i x^i,$

- Example: fit f(x) if

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

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

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



Simplified FEM

- FEM is a method for numerically approximating the solution to PDEs
- FEM finds a piecewise continuous solution function made up of "shape functions" multiplied by coefficients
- FEM provides weak form to approximate the solution to PDE
  - Weak: relax PDE's requirements and only requires it in an integral way
- Steps:
  - 1. Generating a weak form
  - 2. Discretization of the weak form
  - 3. Solve a set of simultaneous linear equations
  - 4. obtain solution function f

Generating a weak form

- Generally involves these steps:
  - Write down strong form of PDE.
  - Rearrange terms so that zero is on the right of the equals sign.
  - Multiply the whole equation by a "test" function ψ.
  - Integrate the whole equation over the domain Ω.
  - Integrate by parts and use the divergence theorem to get the desired derivative order on your functions and simultaneously generate boundary integrals.

Integration by Parts and Divergence Theorem

- Goal: reduce the order of derivatives
- Suppose φ is a scalar function, v is a vector function, and both are continuously differentialable functions, then the product rule states:

$$\nabla \cdot (\varphi \bar{v}) = \varphi(\nabla \cdot \bar{v}) + \bar{v} \cdot (\nabla \varphi)$$

- Integrate the function over the volume V

$$\int \varphi(\nabla \cdot \bar{v})\, dV = \int \nabla \cdot (\varphi \bar{v})\, dV - \int \bar{v} \cdot (\nabla \varphi)\, dV$$

- Divergence theorem transforms a volume integral into a surface integral

$$\int \nabla \cdot (\varphi \bar{v})\, \mathrm{d}V = \int \varphi \bar{v} \cdot \hat{n}\, \mathrm{d}s$$

- So we need to solve

$$\boxed{\int \varphi(\nabla \cdot \bar{v})\, dV = \int \varphi \bar{v} \cdot \hat{n}\, ds - \int \bar{v} \cdot (\nabla \varphi)\, dV}$$

Generate a Weak Form: Advection-Diffusion

- 1. Write the strong form of the equation:

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

- 2. Rearrange to get zero on the right-hand side:

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

- 3. Multiply by the test function ψ:

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

- 4. Integrate over the domain Ω:

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

- 5. Integrate by parts and apply the divergence theorem

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

- 5b. Write in inner product notation. Each term will inherit from an existing MOOSE type

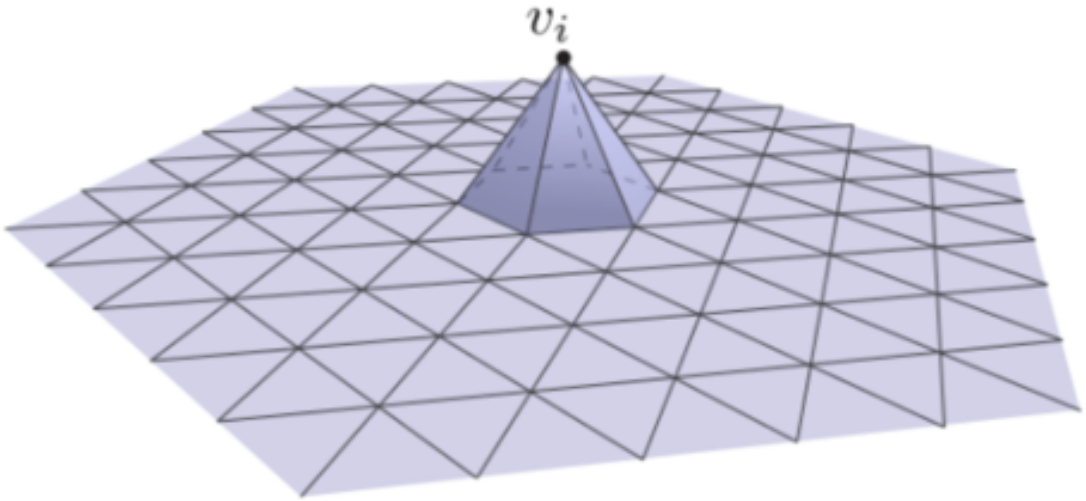
$$\underbrace{\left(\nabla\psi, k\nabla u\right)}_{Kernel} - \underbrace{\left\langle\psi, k\nabla u \cdot \hat{n}\right\rangle}_{BoundaryCondition} + \underbrace{\left(\psi, \bar{\beta} \cdot \nabla u\right)}_{Kernel} - \underbrace{\left(\psi, f\right)}_{Kernel} = 0$$

Discretization

- The weak form must be discretized using a set of "basis functions" amenable for numerical solution by a computer

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

  - u and gradient of u will be discretized using basis functions
- Basis Functions



Shape Functions

- The discretized expansion of *u* takes on the following form:

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

  - where ϕ_j are the "basis functions", which form the basis for the the "trial function", u_h.
  - *N* is the total number of functions for the discretized domain.
- The gradient of *u* can be expanded similarly:

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

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

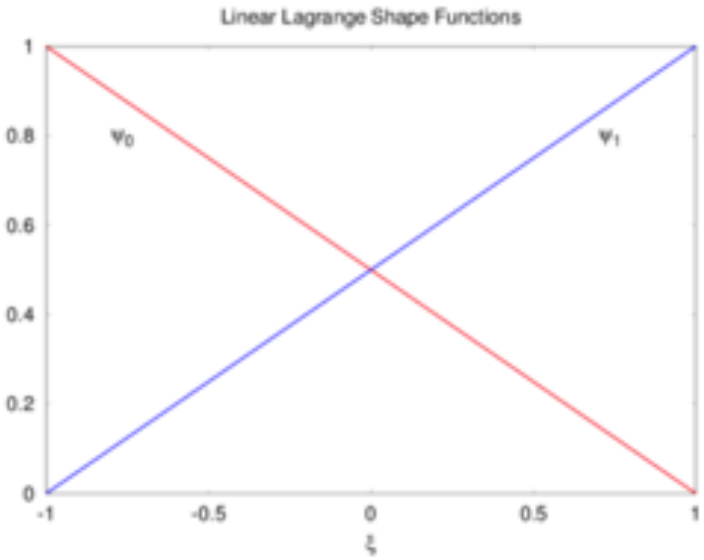
$$\psi = \{\phi_i\}_{i=1}^{N}$$

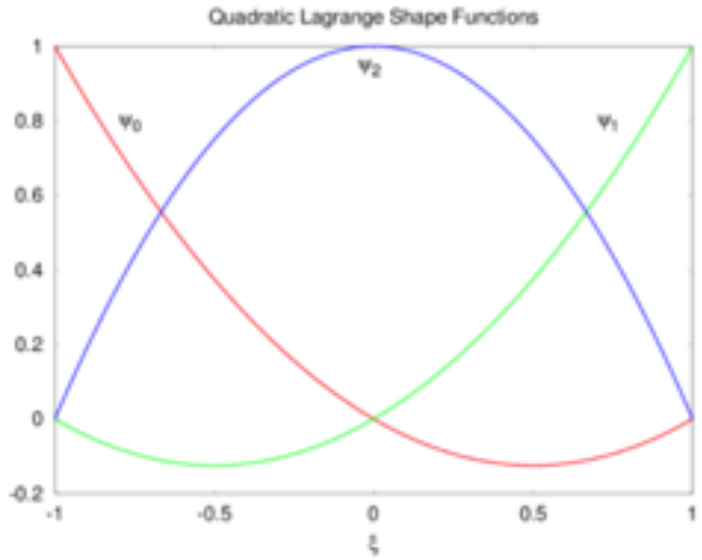
- *ith* component of the "residual vector,"

$$\bar{R}_i(u_h) = (\nabla \psi_i, k \nabla u_h) - \langle \psi_i, k \nabla u_h \cdot \hat{n} \rangle + (\psi_i, \bar{\beta} \cdot \nabla u_h) - (\psi_i, f) = 0$$

Shape functions

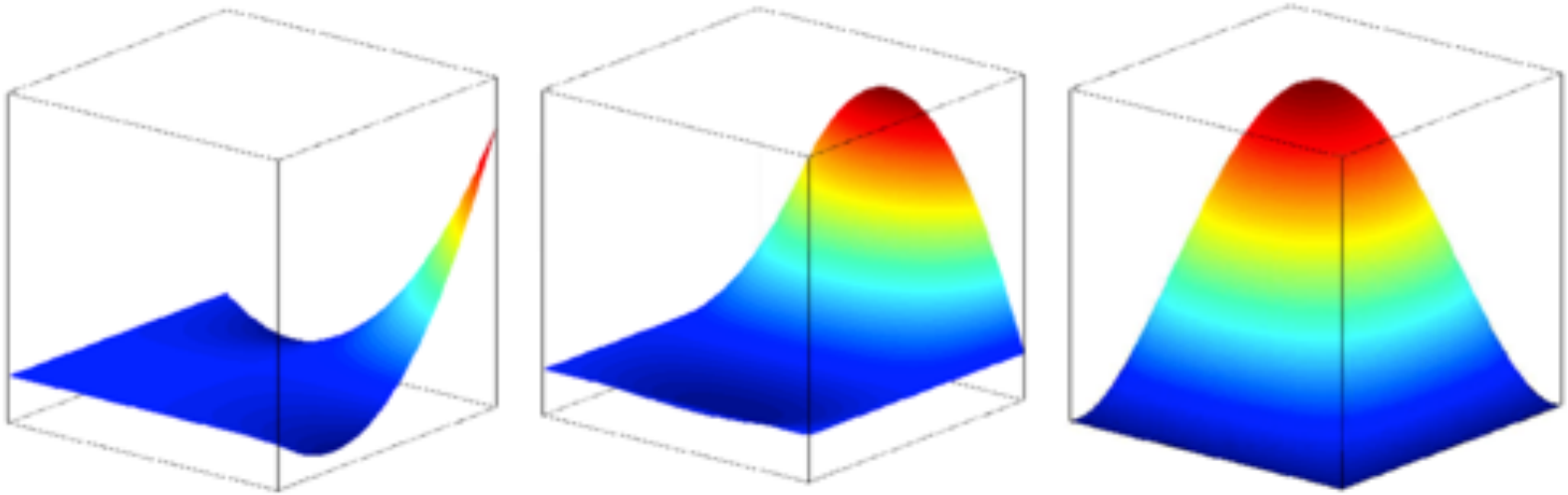
- Shape Functions are the functions that get multiplied by coefficients and summed to form the solution.
- Individual shape functions are restrictions of the global basis functions to individual elements.
- Typical shape function families: Lagrange, Hermite, Hierarchic, Monomial, Clough-Toucher
- Examples: 1D Shape Functions





Example: 2D Lagrange Shape Functions

- 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 \geq 0≥0 on the element.



Done with Discretization?

- Residual Vector

$$\bar{R}_i(u_h) = (\nabla\psi_i, k\nabla u_h) - \langle\psi_i, k\nabla u_h \cdot \hat{n}\rangle + (\psi_i, \bar{\beta}\cdot\nabla u_h) - (\psi_i, f) = 0$$

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

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

- Integrands -- Yes

- Integrals -- No

Discretization of integrals: reference element

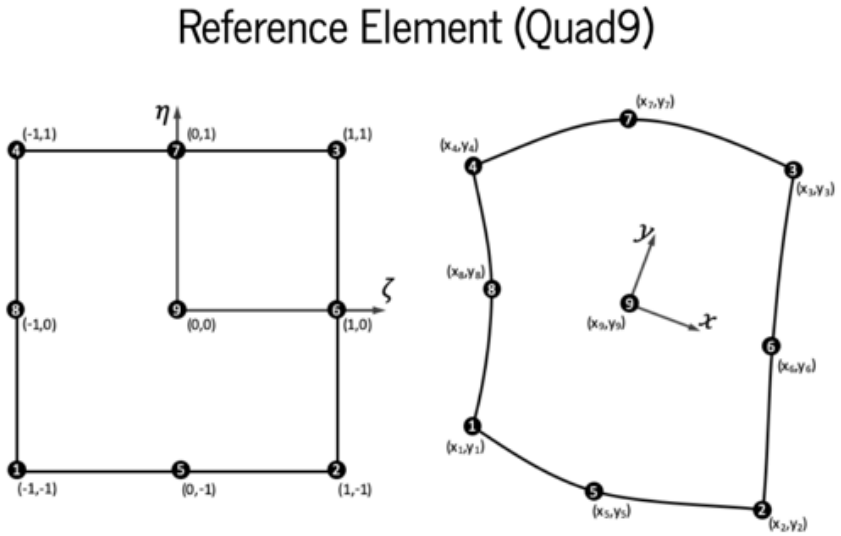
- First, split the domain integral into a sum of integrals over elements

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

- Second, map the element integrals to integrals over the "reference" elements

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

  - Je is the Jacobian of the map from the physical element to the reference element



Numerical Integration: Quadrature

- Gaussian quadrature is used to approximate the reference element integrals numerically.
  - That is, an integration can be represented as summation of weigh functions at quadrature point q.

$$\int_{\Omega} f(\bar{x})\, d\bar{x} \approx \sum_{q} w_q f(\bar{x}_q),$$

- Thus, integrals transforms to summations x_q, the spatial location of the q^{th} quadrature point

$$\int_{\Omega} f(\bar{x})\, \mathrm{d}\bar{x} = \sum_{e} \int_{\Omega_e} f(\bar{x})\, \mathrm{d}\bar{x} = \sum_{e} \int_{\hat{\Omega}_e} f(\bar{\xi})\, |\mathcal{J}_e|\, \mathrm{d}\bar{\xi} \approx \sum_{e} \sum_{q} w_q f(\bar{x}_q)\, |\mathcal{J}_e(\bar{x}_q)|$$

- MOOSE handles multiplication by the Jacobian (Je) and the weight (w_q) automatically, thus your `Kernel` object is only responsible for computing the $f(x_q)$ part of the integrand.

Discretization is done

- Sampling u_h at the quadrature points yields:

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

- Then,

$$\bar{R}_i(u_h) = (\nabla \psi_i, k \nabla u_h) - \langle \psi_i, k \nabla u_h \cdot \hat{n} \rangle + \left(\psi_i, \bar{\beta} \cdot \nabla u_h\right) - (\psi_i, f) = 0$$

$$\bar{R}_i(u_h) = \sum_e \sum_q w_q \left|\mathcal{J}_e\right| \underbrace{\left[\nabla \psi_i \cdot k \nabla u_h + \psi_i \left(\bar{\beta} \cdot \nabla u_h\right) - \psi_i f\right](\bar{x}_q)}_{\text{Kernel Object(s)}}$$
$$- \sum_f \sum_{q_{face}} w_{q_{face}} \left|\mathcal{J}_f\right| \underbrace{\left[\psi_i k \nabla u_h \cdot \bar{n}\right](\bar{x}_{q_{face}})}_{\text{BoundaryCondition Object(s)}}$$

- MOOSE Kenels and BCs can handle them. Problem solved.

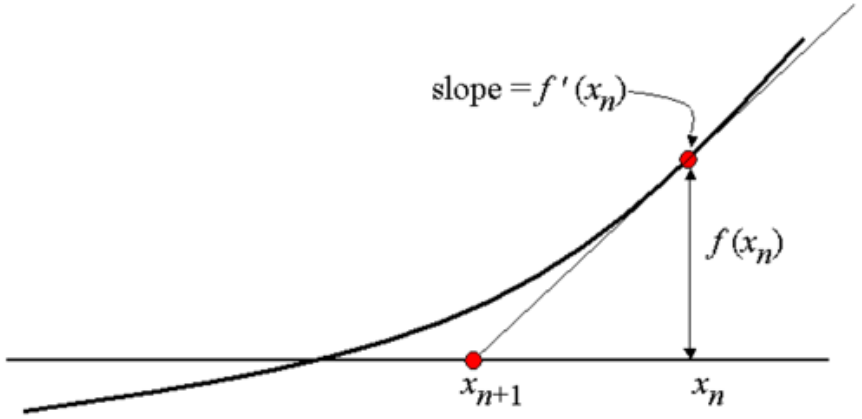
Jacobian Matrix = R'

- Recall Newton-Raphson Method in solving f(x)=0

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

$$\Downarrow$$

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



- Solving u in MOOSE, where $\bar{R}_i(u_h) = 0, \qquad i = 1, \ldots, N,$

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

  - Jacobian Matrix is the derivative of R $J_{ij}(\bar{u}_n) = \frac{\partial \bar{R}_i(\bar{u}_n)}{\partial u_j}$

Solvers in MOOSE

- The solve type is specified in the [Executioner] block within the input file:

```
[Executioner]
  solve_type = PJFNK
```

- Available options include:
  - PJFNK: Preconditioned Jacobian Free Newton Krylov (default)
    - improves convergence
  - JFNK: Jacobian Free Newton Krylov
    - approximates a Jacobian vector product
  - NEWTON: Performs solve using exact Jacobian for preconditioning
  - FD: PETSc computes terms using a finite difference method (debug)
- The Kernel method computeQpResidual is called to compute $\bar{R}(\bar{u}_n)$ during the nonlinear step
- Done!

Automatic and hand-coded Jacobians

- MOOSE uses forward mode automatic differentiation from the MetaPhysicL package.

- Able to develop entire apps without writing a single Jacobian statement

- AD Jacobians are slower to compute than hand-coded Jacobians, but they parallelize extremely well and can benefit from using a NEWTON solve, which often results in decreased solve time overall.

Input Files and Examples

Input File

- By default MOOSE uses a hierarchical, block-structured input file.
- Within each block, any number of name/value pairs can be listed.
- The syntax is completely customizable, or replaceable.
- To specify a simple problem, you will need to populate five or six top-level blocks.

Blocks required for any MOOSE problem

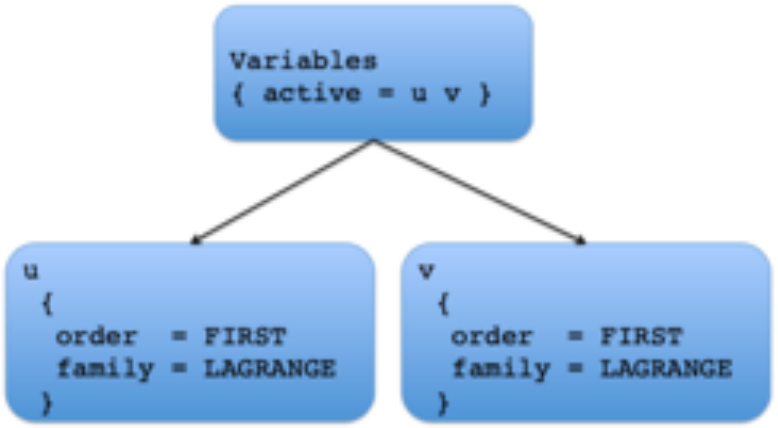
- **[Mesh]**
  - Establishes the FEM mesh, whether read from a file or generated.
- **[Variables]**
  - Defines the variables you will solve and the shape function you will use to approximate them
- **[Kernels]**
  - Defines the residual equation, divided into pieces
- **[BCs]**
  - Establishes the boundary conditions
- **[Executioner]**
  - Establishes the type of simulation (transient vs. steady state), the time behavior, and the solution method.
- **Output:**
  - Defines the outputs for the solution

Optional blocks

- **Materials**
  - Computes material properties required by the kernels.
- **ICs:**
  - Defines the initial state of the variables for transient solves
- **Global Parameters:**
  - Sets values for parameters that are used in multiple places in the input file
- **Auxvariables and Auxkernels**:
  - Auxvariables are dependent variables that are calculated with a corresponding auxkernel.
- **Postprocessors:**
  - Calculates scalar values over the entire mesh.
- **Preconditioners:**
  - Defines the preconditioner to be used during the nonlinear solve.
- **Functions:**
  - Defines a generic function (can be parsed from the input file) that is then used by something else, like a kernel, BC etc.

Hierarchical Block-structure

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

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

[Kernels]
  ...
[]
```



Mesh and Variables

- Mesh

```
[Mesh]
  # We use a pre-generated mesh file (in exodus format).
  # This mesh file has 'top' and 'bottom' named boundaries defined inside it.
  file = media/mug.e
[]
```

- Variables
  - In variables block, each variables represents a non-linear variables to be solved.
  - Each variables can be defined with orders and shape functions, as well as initial values.

```
[Variables]
  [./diffused]
    order = FIRST       # SECOND
    family = LAGRANGE   # HERMITE
  [../]
[]
```

Kernels and BCs

- Kernels
  - Each variables must associate with at least one Kernels block, which represents the equations to solve
  - In this u presents the pressure

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

$$-\nabla \cdot \nabla u = 0 \in \Omega,$$

- BCs

```
[BCs]
  [./bottom] # arbitrary user-chosen name
    type = DirichletBC
    variable = diffused
    boundary = 'bottom' # This must match a named boundary in the mesh file
    value = 1
  [../]
```

$u = 0$ on the left, $u = 1$ on the right

```
  [./top] # arbitrary user-chosen name
    type = DirichletBC
    variable = diffused
    boundary = 'top' # This must match a named boundary in the mesh file
    value = 0
  [../]
[]
```

Executioner and Outputs

- Executioner
  - Executioner block dictates the method that will be used in simulation. There are typically two types of executioner: Steady or Transient
  - we can choose the solver_type, i.e. Newton, PFJNK, FD, Linear.
  - Solve a steady state problem with Newton solver

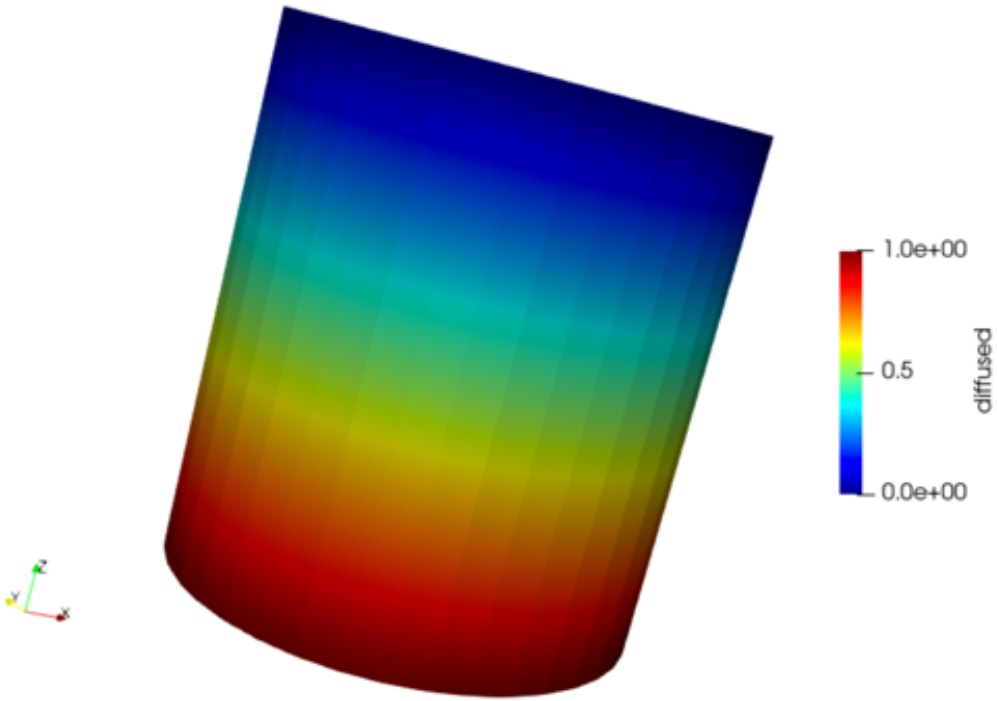
```
[Executioner]
  type = Steady
  solve_type = 'Newton'
[]
```

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

```
[Outputs]
  # file_base = output
  execute_on = 'timestep_end'
  exodus = true
[]
```

- Integrate all above to obtain a 04_input.i

Result: Steady Executioner 04_input.i

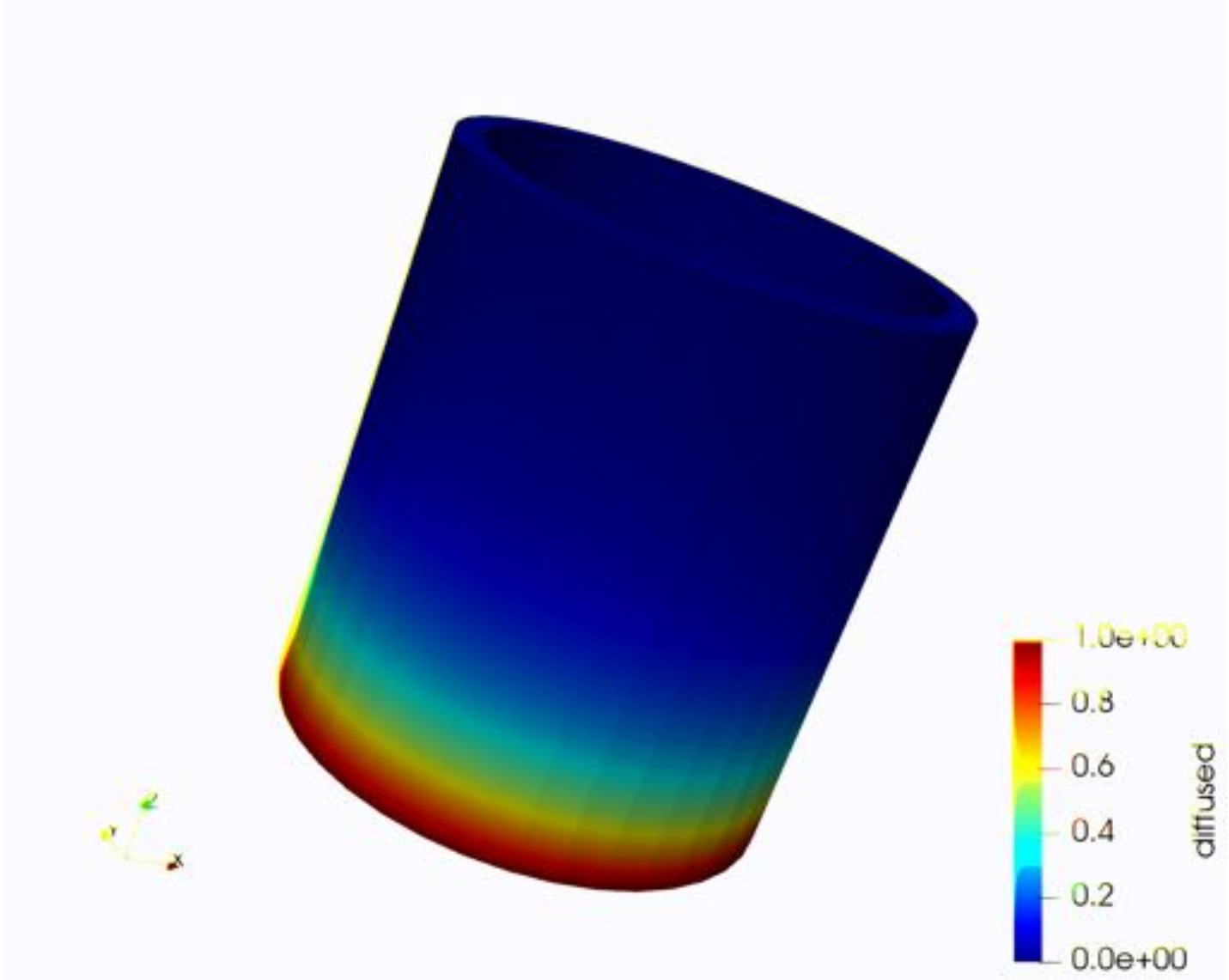


Transient Executioner: 04_Transisent.i

- However, we want to observe our simulations varies in time,
- Use the Transient solver which calculates the residual at every time steps.
  - We also have to included a new kernel block for time derivative.

```
[Kernels]
  [./diff]
    type = Diffusion
    variable = diffused
  [../]
  [./time]
    type = TimeDerivative
    variable = diffused
  [../]
[]

[Executioner]
  type = Transient
  solve_type = 'Newton'
  num_steps = 20
  dt = 1
[]
```

Result: Transient Executioner



Auxiliary System: 05_Aux.i

- If need more meaningful result or more fine control to the problem than just solving for "core" variables
- AuxVariables and AuxKernels
  - perform calculations on top of Variables and Kernels block
- Example: compute the flux of the diffusion equation

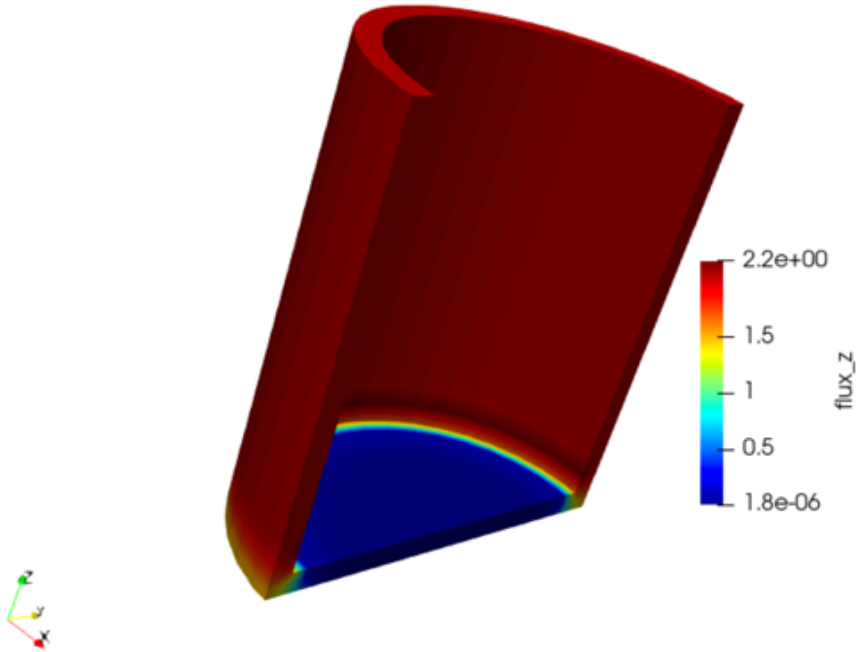
$$J = -D\nabla C$$

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

[AuxKernels]
  [./flux_z]
    type = DiffusionFluxAux
    diffusivity = 'thermal_conductivity'
    variable = flux_z
    diffusion_variable = diffused
    component = z
  [../]
[]
```

```
[Materials]
  [./k]
    type = GenericConstantMaterial
    prop_names = 'thermal_conductivity'
    prop_values = '10' # in W/mK
  []
[]
```

Auxiliary System: 05_Aux.i

- Results



PostProcessor: 05_PostProcessor.i

- post processor is used to aggregate data into a single scalar like maximum or average
  - There are a number of post processor available in MOOSE
    - ElementAverageValue - Average values
    - ElementL2Norm - L2 Norm of the elements

```
[Postprocessors]
  [./Average]
    type = ElementAverageValue
    variable = diffused
  [../]
[]

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

- Running this 05_PostProcessor.i will give us a csv file.

An Advanced Example: DarcyFLow

An advanced example: DarcyFlow

- moose/tutorials/darcy_thermo_mech

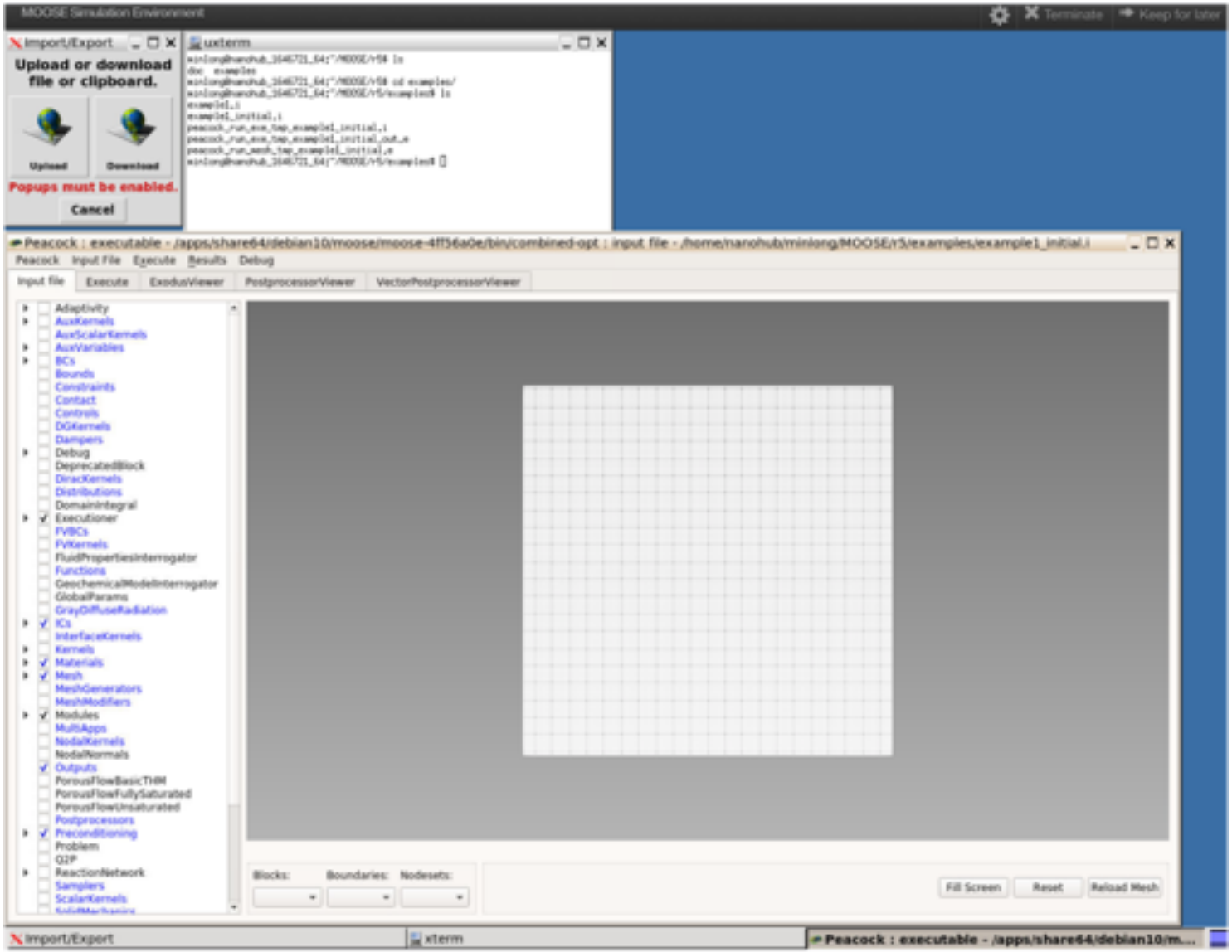
```
(moose) [long@Helix:darcy_thermo_mech]$ lt
total 24
-rwxr-xr-x   1 long  staff   395 Jun  9 04:15 build.sh
-rwxr-xr-x   1 long  staff  2243 Jun  9 04:15 check.py
drwxr-xr-x   5 long  staff   160 Jun  9 04:15 doc
-rwxr-xr-x   1 long  staff   180 Jun  9 04:15 run_tests
drwxr-xr-x  12 long  staff   384 Jun  9 04:15 step02_darcy_pressure
drwxr-xr-x  12 long  staff   384 Jun  9 04:15 step03_darcy_material
drwxr-xr-x  12 long  staff   384 Jun  9 04:15 step04_velocity_aux
drwxr-xr-x  12 long  staff   384 Jun  9 04:15 step05_heat_conduction
drwxr-xr-x  12 long  staff   384 Jun  9 04:15 step06_coupled_darcy_heat_conduction
drwxr-xr-x  12 long  staff   384 Jun  9 04:15 step07_adaptivity
drwxr-xr-x  12 long  staff   384 Jun  9 04:15 step08_postprocessors
drwxr-xr-x  12 long  staff   384 Jun  9 04:15 step09_mechanics
drwxr-xr-x  13 long  staff   416 Jun  9 04:15 step10_multiapps
drwxr-xr-x  12 long  staff   384 Jun  9 04:15 step11_action
drwxr-xr-x  18 long  staff   576 Jun 10 01:32 step01_diffusion
```

- Tutoring (typically 2 days)
  - https://mooseframework.org/workshop/

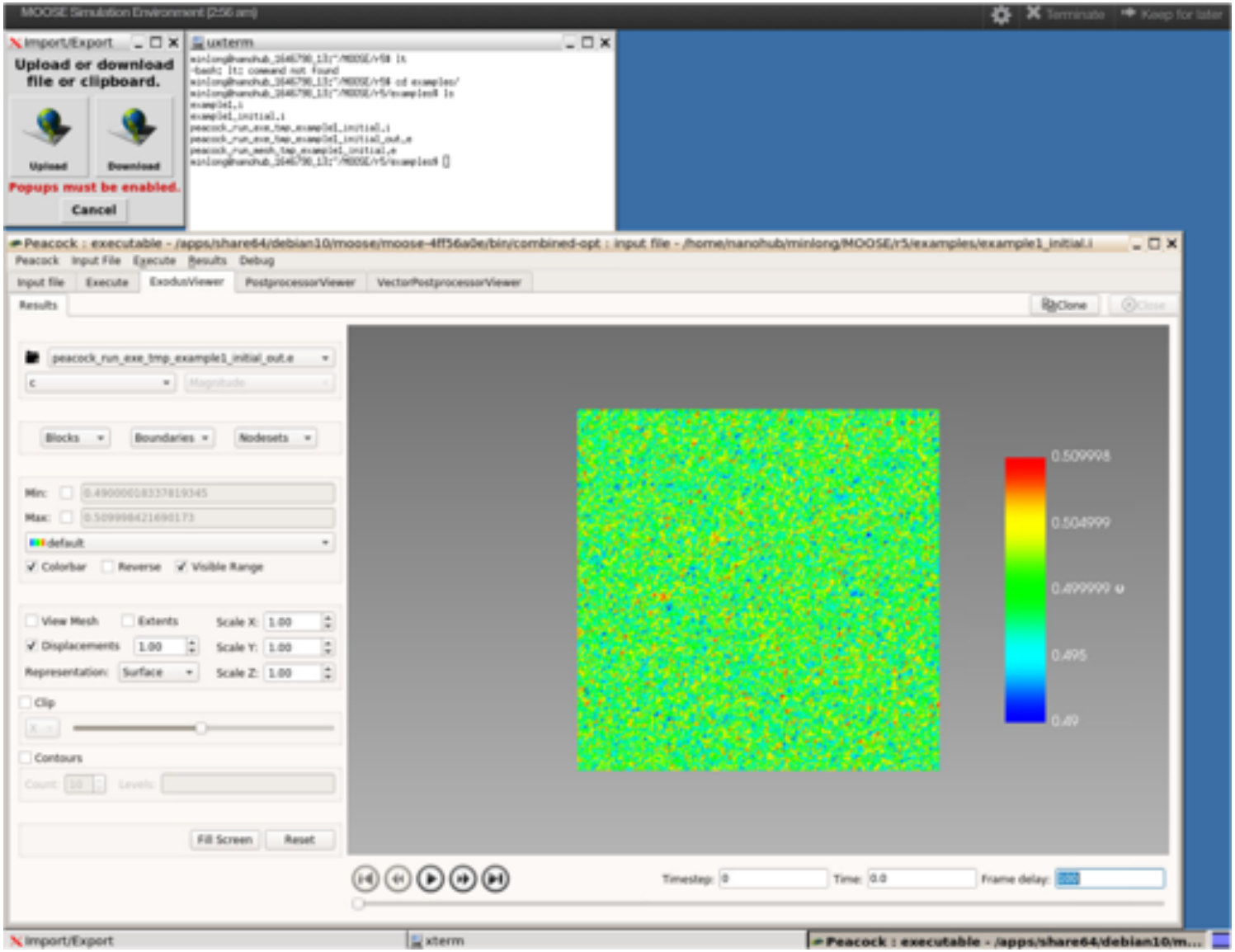
MOOSE Tools in NanoHub

https://nanohub.org/tools/moose

Click “launch tool”



Execute – Run -- ExodusViewer



Next: A hands-on training on how to apply the MOOSE framework to mesoscale materials