Introduction to the MOOSE PDE Solver Framework

NECTAR Lab, UC Santa Cruz

April 10, 2016

Introduction

This note introduces the Multiphysics Object-Oriented Simulation Environment (MOOSE) finite-element PDE solver framework [1]. Through an API, MOOSE provides an extensbible high-level framework to the PetSC nonlinear solver and libmesh mesher to solve a broad variety of partial differential equations typically encountered in engineering and physics. MOOSE was developed by the Idaho National Laboratory, while the PetSC [2] and libmemsh tools were developed at the Argrone and Sandia National Laboratories, respectively [3].

In contrast to commercial FEM tools, which are generally centered around a propreitary integrated design environment (IDE), the MOOSE workflow requires several additional applications, including a C compiler and visualization tools. A central feature of the MOOSE framework is the weak form specification of physics kernels in C, providing a flexibe mechanism for the formulation of complex anisotropic and nonlinear contiuum and hydrodynamic flow problems.

A MOOSE workflow is presented, including installation instructions. A simualtion example is provided to illustrate the application of MOOSE for self-consistent electrothermal simulation of a memristor structure.

What is MOOSE

MOOSE solves systems of coupled partical differential equations. Each PDE is cast into an associated weak formualtion of the fintie element method to represent and solve the coupled system on a discretized 2-D or 3-D mesh. For example, consider the heat equation with forcing term Q_f

$$\rho C_p \frac{\partial T}{\partial t} + \nabla \cdot \vec{k_T} \nabla T = Q_f \tag{1}$$

where ρ is mass density, C_p is (nonlinear) heat capacity, and $\vec{k_T}$ is (nonlinear and anisotropic) thermal conductance. The weak formulation is a variational statement on an element expressed, expressed in inner-product form as

$$\left(\psi, \rho C_p \dot{T}\right) + \left(\nabla\psi, \vec{k_T}\nabla T\right) = \left(\psi, Q_f\right) \tag{2}$$

where ψ is a test function that approximates the PDE solution over the element. It is the weak form Eq. 2 that is coded as a C object to represent physics of the problem that is solved by MOOSE. Each term of each PDE yields a kernel in C that is supplied to MOOSE and subsequently solved using iterative techniques.

Both steady-state and transient solvers are available. A major feature of MOOSE is the Newton-free Krylov solver, precluding the need for analytical Jacobian derivation or analytical computation. MOOSE outputs an industry-standard Exodus data file, suitable for visualization. Each of these steps are in general distinct and done with individual applications that make up the MOOSE workflow.

The MOOSE Workflow

This section discusses the minimum suite of applications to create an effective workflow, with a major feature being the broad discretion afforded in configuration. The MOOSE workflow depends largely on user preference, with a minimum workflow suite itemized below; each of these are disucssed in turn.

Following description of custom physics, using the weak form cast in C code, which requires compiling, the majority of the workflow centers around simulation specification by the MOOSE deck (text editor), executing the simulation (command line), and visualization (application).

- MOOSE application
- Text editor
- Mesh application

- Visualization application
- Compiler

The MOOSE Application

The MOOSE application is similar to Python or MATLAB in many ways, such as a command-line interface to an API (Application Program Interface). MOOSE must be complied when it is installed, unlike MATLAB, so a compiler is required. The resulting executable provides an API that executes a simulation described by a text-file in which custom physics are linked to MOOSE run-time by user-supplied C code. The MOOSE API is itself based on the PetSC PDE solver from Argonne National Lab and the libmesh framework from Sandia National Lab.

Text Editor

The text editor is the focal point of the MOOSE workflow, with the Atom being the endorsed text editor; a custom configuration file is avaiable for Atom for syntax highlighting for the MOOSE API. It is suggested to read `http://mooseframework.org/wiki/MooseTraining/InputFile/` for an introduction to basic elements of a MOOSE deck.

The Mesh Application

MOOSE requires that a mesh be supplied, inlcuding boundary and volume names. The libmesh framework from Sandia has a commercial version with academic pricing, which was evaluated. Simultaneously, the Gmsh open-source mesher was evaluated and found to be suitable for introductory research; it is this mesher that is the focus of the present note. The Gmsh application has stable builds for Mac, Linux, and Windows.

The Visualization Application

The MOOSE install provides a basic GUI interface the API and includes basic visualiztion. MOOSE exports an industry standard Exodus file that can be read by many third-party visualization applications; for the present note, open-source ParaView delivered exeception visualization capability.

Compliler

To invoke the MOOSE API, a machine-specific executable must be provided by a compiler. Moreover, custom physics in MOOSE must be compiled and linked to the MOOSE executable. Generally, once the physics are coded, no further compiling is necessary.

OSX Installation

This sections describes the El Captain OSX installation procedure for MOOSE, and the associated applications from the previous section, specifically the Atom text editor, the Gmsh mesher, the ParaView visualizer, and the gcc C compiler.

Optimum installation results with a clean OSX install, followed by the MOOSE install. The remaining applications can be installed in any order.

MOOSE Install

The El Captain OSX installation instructions are found at `http://mooseframework.org/getting-started/osx/`. Note that Xcode, with command line tools, must be installed prior to the MOOSE installation.

Following the MOOSE installation, go to `http://www.mooseframework.org/getting-started/` and follow the instructions for cloning MOOSE and and compiling libmesh. Compiling takes about an hour. Run the test suite to confirm MOOSE has been properly installed.

Atom Install

The Atom download is found at `https://atom.io`. It is recommended to install the MOOSE-specific syntax-highlighting package, found at `https://atom.io/packages/language-moose`.

Gmsh Install

The Gmsh download is found at `http://gmsh.info`. Note that Gmsh offers both 2-D and 3-D mesh support.

ParaView Install

The ParaView download is found at `http://www.paraview.org`.

Compiler Install

The X-Code installation will install gcc, an OSX-compatible C/C++ compiler. To verify correct installtion, type `cc --version` at an X-Term command line.

Simulation Example with MOOSE

This section provides the details necessary to self-consistently simulate transient and steady-state Joule self-heating of the memrisor structure illustrated in Figure 1.

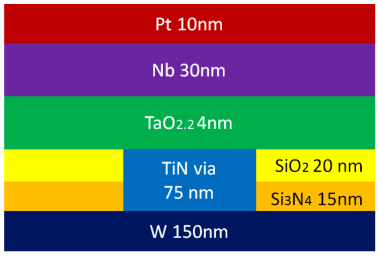
The Weak Formulation

To model Joule heating, the following PDEs are solved self-consistently

$$\rho C_p \frac{\partial T}{\partial t} + \nabla \cdot \vec{k_T} \nabla T - Q_f = 0 \tag{3a}$$

$$\nabla \cdot \sigma \nabla V = 0 \tag{3b}$$

where ρ is mass density, C_p is (nonlinear) heat capacity, $\vec{k_T}$ is (nonlinear and anisotropic) thermal conductance, and σ is (nonlinear) electrical conductance. Thermal and electric potential variables T and V, respecitvely, are 2-D functions of space and time-dependent. Equations 3a and 3b are solved subject to the Joule heating coupling term that relates instantaneous power density to the thernal forcing term



.

Figure 1: Approximate memristor cross-section and material stack-up

$$Q_f = \sigma |\nabla V|^2 \tag{4}$$

The diffusion operator is the only built-in function provided by the MOOSE API [1]. To solve self-consisently Equations 3a, 3b, and 4, requires a custom kernal for coupling, represetned by Q_f, and the intertial term, represented by $\rho C_p \dot{T}$. The following weak formulations obtain, respectively,

$$(\psi, \sigma\, u_h\, u_h) \tag{5a}$$

$$\left(\psi, \rho C_p \dot{T}\right) \tag{5b}$$

Kernel Implementation

Each kerna requires a header (.h) file with declarations and a code file (.c) that implements the weak formulation from the previous section. Each kernel must also be registered with the MOOSE (.c). A complete listing of MOOSE classes is available at `http://mooseframework.org/docs/doxygen/moose/classes.html`. Specfically the weak form Equation 5a uses the coupledValue class, described here `http://mooseframework.org/docs/doxygen/moose/classCoupleable.html#adb2f4c4446e2100f486abeb2513c4f01`.

The C code fragment represting the weak formulation Equation 5a is given immediately below, where u_h of Equation 5a is represented in the function by `some_variable`. Note that the weak form is expressed as the LHS, so the (-1) term is necessary [2]. The material conductivity is passed by the variable `_diffusivity` which represents σ.

```
#include "ExampleConvection.h"

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

  params.addRequiredCoupledVar("some_variable", "Some variable.");
  return params;
}

ExampleConvection::ExampleConvection(const InputParameters & parameters) :
    Kernel(parameters),
    _some_variable(coupledValue("some_variable")),
    _diffusivity(getMaterialProperty<Real>("diffusivity"))
{}

Real ExampleConvection::computeQpResidual()
{
  return _test[_i][_qp]*(_some_variable[_qp]*_some_variable[_qp])*(_diffusivity[_qp])*(-1);

}

Real ExampleConvection::computeQpJacobian()
{
  return _test[_i][_qp]*(_diffusivity[_qp]*_some_variable[_qp]*_some_variable[_qp]*_phi[_j][
      _qp])*(1);
}
```

The C code fragment representing the weak formulation Equation 5b is given immediately below. Specific heat and density are passed by the variables `_specific_heat` and `_density`. Note that the time derivative function is called in the code. Further, and significantly, note that an analytical Jacobian is not required to be specified, a major feature of MOOSE.

[1] See `http://mooseframework.com/docs/syntax/moose/`

[2] In this code fragment, and the following two, not all the code has been included from the .c file, what is shown are the most important fragments.

```
#include "HeatConductionTimeDerivative.h"

template<>
InputParameters validParams<HeatConductionTimeDerivative>()

HeatConductionTimeDerivative::HeatConductionTimeDerivative(const InputParameters &
    parameters)
  : TimeDerivative(parameters),
   _use_heat_capacity(getParam<bool>("use_heat_capacity")),
   _specific_heat(NULL),
   _density(NULL)
{
}

Real
HeatConductionTimeDerivative::computeQpResidual()
{
  return (*_specific_heat)[_qp] * (*_density)[_qp] * TimeDerivative::computeQpResidual();
}

Real
HeatConductionTimeDerivative::computeQpJacobian()
{
  return (*_specific_heat)[_qp] * (*_density)[_qp] * TimeDerivative::computeQpJacobian();
}
```

The C code fragment kernel registration is shown immediately below, under the comment section for self-consistent electrothermal includes, illustrating various custom kernels and material definitions. Kernel implementation is concluded by compling the various code framgments and linking them the MOOSE executable; this is generally done with the `make` that comes with the MOOSE install. Each of the files must also be placed in their respective directories.

Passing Material Parameters

Material parameters are be passed to to the kernels of the previous section by using the material functions `HeatConductionMaterial` for passing thermal conductivity, specific heat, and density and `ExampleMaterial` for passing electrical conductivity. Material parameters are expressed as vectors / tensors from the MOOSE deck

The MOOSE Simulation Deck

The MOOSE simulation is specified by a text file composed of the following elements, which are discussed in turn. A complete summarty of MOOSE simulation elements and systems is located at `http://mooseframework.org/wiki/MooseSystems/` with parameter definitions located at `http://mooseframework.com/docs/syntax/moose/`.The MOOSE simulation deck is listed in Appendix A.

- Mesh
- Variables
- Mesh application
- Kernels
- Boundary Conditions
- Materials
- Executioner
- Output

```
#include "ExampleApp.h"
#include "Moose.h"
#include "Factory.h"
#include "AppFactory.h"

// Self-consistent electrothermal includes, March 21 2016
#include "ExampleConvection.h"
#include "ExampleMaterial.h"
#include "HeatConductionMaterial.h"
#include "HeatConduction.h"
#include "HeatConductionTimeDerivative.h"

template<>
InputParameters validParams<ExampleApp>()
{
  InputParameters params = validParams<MooseApp>();

  params.set<bool>("use_legacy_uo_initialization") = false;
  params.set<bool>("use_legacy_uo_aux_computation") = false;
  return params;
}

ExampleApp::ExampleApp(InputParameters parameters) :
    MooseApp(parameters)
{
  srand(processor_id());

  Moose::registerObjects(_factory);
  ExampleApp::registerObjects(_factory);

  Moose::associateSyntax(_syntax, _action_factory);
  ExampleApp::associateSyntax(_syntax, _action_factory);
}

ExampleApp::~ExampleApp()
{
}

void
ExampleApp::registerObjects(Factory & factory)
{
  // Register ExampleConvection even though it is a forcing term.
  registerKernel(ExampleConvection);

  // Register our new material class so we can use it.
  registerMaterial(ExampleMaterial);

  // Register the heat conduction kernel.
  registerKernel(HeatConductionKernel);

  // Register thermal material properties.
  registerMaterial(HeatConductionMaterial);

  // Register transient thermal kernel.
  registerKernel(HeatConductionTimeDerivative);
}
```

[Mesh]

The mesh is specified here, as a .msh file. MOOSE has basic commands to construct simple objects, rename boundaries and volumes, and resample the mesh.

[Variables]

Variables are defeind here, as well as the family of approximation functions the test function ψ belings to, the approximation order, and initial conditions.

[Kernels]

Kernels are defined here, by `type` and `variable`. In the present example, `HeatConductionKernel` and `HeatConductionTransient` represents the diffusion inerial terms, respectively, of Equation 1. The forcing term of Equation 1 is represented by `couplingKernel`. The electric potential is also represented by `HeatConductionKernel`.

[Boundary Conditions]

Boundary conditions are defined here, with both Dirchlet (constant temperature) and Neumann (constant flux) supported; custom boundary can also be defined, e.g. periodic or absorbring. Boundary conditions are attached to mesh boundaries using `boundary`.

[Materials]

Material properties are specified here, with `block` used to attach a material property to a specific volume or doamin. Thermal properties are passed as a vector of terms specifying $\vec{k_T}$, C_p, and ρ. Electrical conductivity is passed a vector of idepdendent terms (T) and dependet terms.

[Executioner]

The simulation type is specified in the executioner block, with steady-state and transient specified by `steady` and `transient`, respectively. Additional options are also specified, similar to SPICE, e.g. time step and simulation duration.

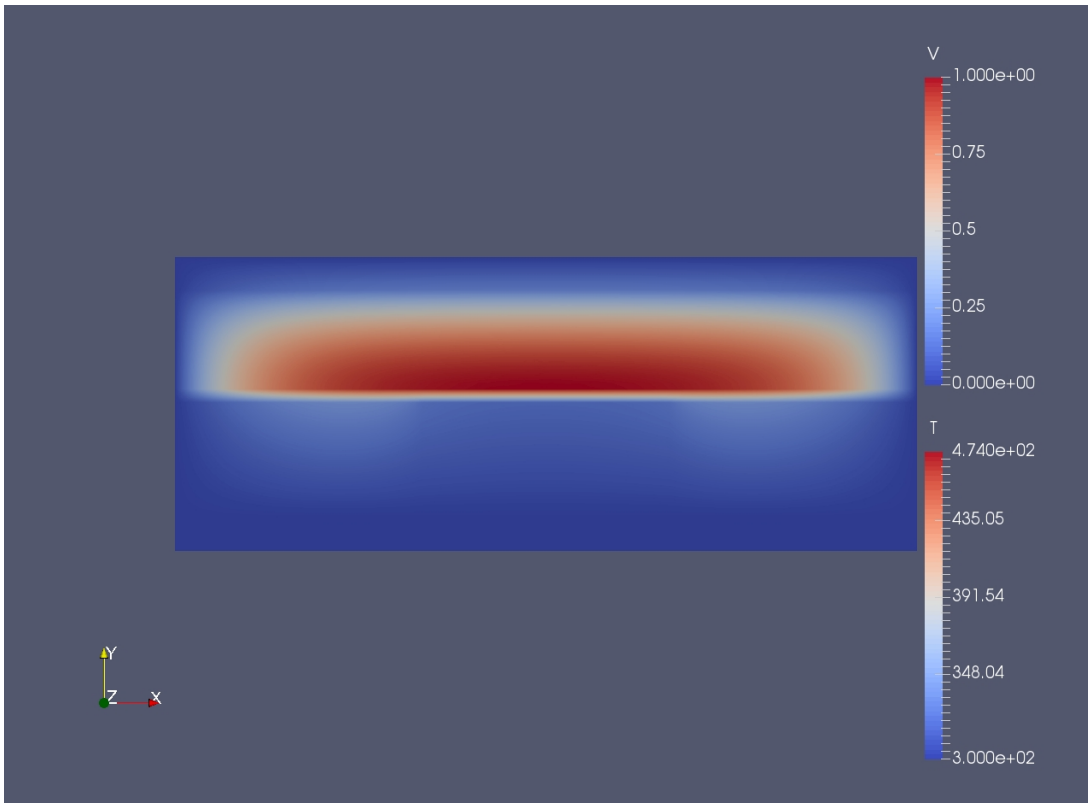
[Output]

Output parameters are specified here.

Simulation Result

The following embedded movie illustrates the the transient results of the simulation. Immediately below the movie is the steady-state result.

Figure 2: Self-consistent transient electrothermal solution to Equations 3a and 3b under 300 K boundary condition around the perimeter and a 1.0 V potential difference between the top and bottom electrodes



.

Figure 3: Self-consistent steady-state electrothermal solution to Equations 3a and 3b under 300 K boundary condition around the perimeter and a 1.0 V potential difference between the top and bottom electrodes

Appendix A: MOOSE Deck Listing

```
[Mesh]
  file = FullScaleStructure01.msh
  uniform_refine = 1.0
[]
[Variables]
  # This variable is the temperature field, T.
  [./T]
    order = FIRST
    family = LAGRANGE
    initial_condition = 300.0
  [../]
  # This variable is the electric potential field, V.
  [./V]
    order = FIRST
    family = LAGRANGE
    initial_condition = 1.0
  [../]
[]
[Kernels]
  active = 'laplaceEq couplingKernel HeatConduction HeatConductionTransient'
  [./laplaceEq]
    # This kernel solves the Laplace equation for V.
    type = HeatConductionKernel
    variable = V
  [../]
  [./HeatConduction]
    # This kernel solves the heat equation for T.
    type = HeatConductionKernel
    variable = T
  [../]
  [./couplingKernel]
    # This kernel couples the thermal potential to the electrical potential.
    type = ExampleConvection
    variable = T
    some_variable = V
  [../]
  [./HeatConductionTransient]
    type = HeatConductionTimeDerivative
    variable = T
    use_heat_capacity = false
  [../]
[]
[BCs]
  [./thermalDiffBottom]
    type = DirichletBC
    variable = T
    boundary = 'bottom'
    value = 300 # Assume the substrate is room temperature.
  [../]
  [./thermalDiffTop]
```

```
    type = DirichletBC
    variable = T
    boundary = 'top'
    value = 300
  [../]
  [./thermalDiffLeft]
    type = DirichletBC
    variable = T
    boundary = 'left'
    value = 300 # Assume the substrate is room temperature.
  [../]
  [./thermalDiffRight]
    type = DirichletBC
    variable = T
    boundary = 'right'
    value = 300
  [../]
  [./laplaceEqBottom]
    type = DirichletBC
    variable = V
    boundary = 'bottom'
    value = 0 # Hold the substrate at 0 V potential.
  [../]
  [./laplaceEqTop]
    type = DirichletBC
    variable = V
    boundary = 'top'
    value = 1.0 # Bias voltage.
  [../]
[]
[Materials]
  [./gW]
    # W thermal properties.
    type = GenericConstantMaterial
    block = 'W'
    prop_names = 'thermal_conductivity specific_heat density'
    prop_values = '173 5 1' # W/m*K, J/kg-K, kg/m^3 @ 296K
  [../]
  [./gTaO]
    # TaO thermal properties.
    type = GenericConstantMaterial
    block = 'TaO'
    prop_names = 'thermal_conductivity specific_heat density'
    prop_values = '0.05 1 1' # W/m*K, J/kg-K, kg/m^3 @ 296K
  [../]
  [./gSiN_Left]
    # SiN thermal properties, left side.
    type = GenericConstantMaterial
    block = 'SiN_Left'
    prop_names = 'thermal_conductivity specific_heat density'
    prop_values = '1.0 0.1 1' # W/m*K, J/kg-K, kg/m^3 @ 296K
  [../]
```

```
[./gSiN_Right]
  # SiN thermal properties, right side.
  type = GenericConstantMaterial
  block = 'SiN_Right'
  prop_names = 'thermal_conductivity specific_heat density'
  prop_values = '1.0 0.1 1' # W/m*K, J/kg-K, kg/m^3 @ 296K
[../]
[./gTiN]
  # TiN thermal properties.
  type = GenericConstantMaterial
  block = 'TiN'
  prop_names = 'thermal_conductivity specific_heat density'
  prop_values = '4 5 1' # W/m*K, J/kg-K, kg/m^3 @ 296K
[../]
[./gSiO_Left]
  # SiO thermal properties, left side.
  type = GenericConstantMaterial
  block = 'SiO_Left'
  prop_names = 'thermal_conductivity specific_heat density'
  prop_values = '1.0 0.1 1' # W/m*K, J/kg-K, kg/m^3 @ 296K
[../]
[./gSiO_Right]
  # SiO thermal properties, right side.
  type = GenericConstantMaterial
  block = 'SiO_Right'
  prop_names = 'thermal_conductivity specific_heat density'
  prop_values = '1.0 0.1 1' # W/m*K, J/kg-K, kg/m^3 @ 296K
[../]
[./gNb]
  # Nb thermal properties.
  type = GenericConstantMaterial
  block = 'Nb'
  prop_names = 'thermal_conductivity specific_heat density'
  prop_values = '30 2 1' # W/m*K, J/kg-K, kg/m^3 @ 296K
[../]
[./gPt]
  # Pt thermal properties.
  type = GenericConstantMaterial
  block = 'Pt'
  prop_names = 'thermal_conductivity specific_heat density'
  prop_values = '100 5 1' # W/m*K, J/kg-K, kg/m^3 @ 296K
[../]
[./W]
  # W electrical conductivity.
  type = ExampleMaterial
  block = 'W'
  diffusion_gradient = 'T'
  independent_vals = '1 1000'
  dependent_vals = '1e2 1e2'
[../]
[./TaO]
  # TaO electrical conductivity.
```

```
  type = ExampleMaterial
  block = 'TaO'
  diffusion_gradient = 'T'
  independent_vals = '1 1000'
  dependent_vals = '1e-2 1e-2'
[../]
[./SiN_Left]
  # SiN electrical conductivity, left side.
  type = ExampleMaterial
  block = 'SiN_Left'
  diffusion_gradient = 'T'
  independent_vals = '1 1000'
  dependent_vals = '1e-6 1e-6'
[../]
[./SiN_Right]
  # SiN electrical conductivity, right side.
  type = ExampleMaterial
  block = 'SiN_Right'
  diffusion_gradient = 'T'
  independent_vals = '1 1000'
  dependent_vals = '1e-6 1e-6'
[../]
[./TiN]
  # TiN electrical conductivity.
  type = ExampleMaterial
  block = 'TiN'
  diffusion_gradient = 'T'
  independent_vals = '1 1000'
  dependent_vals = '1e-0 1e-0'
[../]
[./SiO_Left]
  # SiO electrical conductivity, left side.
  type = ExampleMaterial
  block = 'SiO_Left'
  diffusion_gradient = 'T'
  independent_vals = '1 1000'
  dependent_vals = '1e-6 1e-6'
[../]
# Specifiy right SiO alpha here (sigma / kappa).
[./SiO_Right]
  # SiO electrical conductivity, right side.
  type = ExampleMaterial
  block = 'SiO_Right'
  diffusion_gradient = 'T'
  independent_vals = '1 1000'
  dependent_vals = '1e-6 1e-6'
[../]
[./Nb]
  # Nb electrical conductivity.
  type = ExampleMaterial
  block = 'Nb'
  diffusion_gradient = 'T'
```

```
    independent_vals = '1 1000'
    dependent_vals = '1e1 1e1'
  [../]
  [./Pt]
    # Pt electrical conductivity.
    type = ExampleMaterial
    block = 'Pt'
    diffusion_gradient = 'T'
    independent_vals = '1 1000'
    dependent_vals = '1e0 1e0'
  [../]
[]
[Executioner]
  type = Transient
  num_steps = 500
  dt = 0.5
  solve_type = PJFNK
[]
[Outputs]
  exodus = true
[]
```

References

[1] D. Gaston, C. Newman, G. Hansen, and D. Lebrun, "Moose: A parallel computational framework for coupled systems of nonlinear equations," *Nuclear Engineering and Design*, vol. 239, no. 10, pp. 1768 – 1778, 2009.

[2] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang, "PETSc Web page," http://www.mcs.anl.gov/petsc, 2015. [Online]. Available: http://www.mcs.anl.gov/petsc

[3] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey, "`libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations," *Engineering with Computers*, vol. 22, no. 3–4, pp. 237–254, 2006, http://dx.doi.org/10.1007/s00366-006-0049-3.