Serac Documentation

Release 0.1

Dec 07, 2022

Contents

Serac is a 3D implicit nonlinear thermal-structural simulation code. Its primary purpose is to investigate multiphysics abstraction strategies and implicit finite element-based algorithm development for emerging computing architectures. It also serves as a proxy-app for LLNL's Smith code and heavily leverages the MFEM finite element library.

Note: This project is under heavy development and is currently a pre-alpha release. Functionality and interfaces may change rapidly as development progresses.

CHAPTER 1

Copyright and License Information

Please see the LICENSE file in the repository.

LLNL-CODE-805541

1.1 Quickstart Guide

1.1.1 Getting Serac

Serac is hosted on GitHub. Serac uses git submodules, so the project must be cloned recursively. Use either of the following commands to pull Serac's repository:

```
# Using SSH keys setup with GitHub
$ git clone --recursive git@github.com:LLNL/serac.git

# Using HTTPS which works for everyone but is slightly slower and will require␣
↪username/password
# for some commands
$ git clone --recursive https://github.com/LLNL/serac.git
```

1.1.2 Overview of the Serac build process

The Serac build process has been broken into three phases with various related options:

1. (Optional) Build the developer tools
2. Build the third party libraries
3. Build the serac source code

The developer tools are only required if you wish to contribute to the Serac source code. The first two steps involve building all of the third party libraries that are required by Serac. Two options exist for this process: using the Spack HPC package manager via the uberenv wrapper script or building the required dependencies on your own. We recommend the first option as building HPC libraries by hand can be a tedious process. Once the third party libraries are built, Serac can be built using the cmake-based BLT HPC build system.

Note: If you get the following error `ERROR: pip version 19.0.3 is too old to install clingo`, run the following command to upgrade your pip: `python3 -m pip install --user --upgrade pip`. This error will not necessarily be the last error on the screen.

1.1.3 Building Serac's Developer Tools

Note: This can be skipped if you are not doing Serac development or if you are on an LC machine. They are installed in a group space defined in `host-config/<machine name>-<SYS_TYPE>-<compiler>.cmake`

Serac developers utilize some industry standard development tools in their everyday work. We build these with Spack and have them installed in a public space on commonly used LC machines. These are defined in the host-configs in our repository.

If you wish to build them yourself (which takes a long time), use one of the following commands:

For LC machines:

```
$ python3 scripts/llnl/build_devtools.py --directory=<devtool/build/path>
```

For other machines:

```
$ python3 scripts/uberenv/uberenv.py --project-json=scripts/spack/devtools.json --
↪spack-config-dir=<spack/config/dir> --prefix=<devtool/build/path>
```

For example on **Ubuntu 20.04**:

```
python3 scripts/uberenv/uberenv.py --project-json=scripts/spack/devtools.json --spack-
↪config-dir=scripts/spack/configs/linux_ubuntu_20 --prefix=../path/to/install
```

Unlike Serac's library dependencies, our developer tools can be built with any compiler because they are not linked into the serac executable. We recommend GCC 8 because we have tested that they all build with that compiler.

1.1.4 Building Serac's Dependencies via Spack/uberenv

Note: This is optional if you are on an LC machine as we have previously built the dependencies. You can see these machines and configurations in the `host-configs` repository directory.

Serac only directly requires MFEM, Axom, and Conduit. Through MFEM, Serac also depends on a number of other third party libraries (Hypre, METIS, NetCDF, ParMETIS, SuperLU, and zlib).

The easiest path to build Serac's dependencies is to use Spack. This has been encapsulated using Uberenv. Uberenv helps by doing the following:

- Pulls a blessed version of Spack locally

- If you are on a known operating system (like TOSS3), we have defined Spack configuration files to keep Spack from building the world
- Installs our Spack packages into the local Spack
- Simplifies whole dependency build into one command

Uberenv will create a directory containing a Spack instance with the required Serac dependencies installed.

Note: This directory **must not** be within the Serac repo - the example below uses a directory called `serac_libs` at the same level as the Serac repository. The `--prefix` input argument is required.

It also generates a host-config file (`<config_dependent_name>.cmake`) at the root of Serac repository. This host-config defines all the required information for building Serac.

```
$ python3 scripts/uberenv/uberenv.py --prefix=../serac_libs
```

Note: On LC machines, it is good practice to do the build step in parallel on a compute node. Here is an example command: `salloc -ppdebug -N1-1 python3 scripts/uberenv/uberenv.py`

Unless otherwise specified Spack will default to a compiler. This is generally not a good idea when developing large codes. To specify which compiler to use add the compiler specification to the `--spec` Uberenv command line option. On TOSS3, we recommend and have tested `--spec=%clang@10.0.0`. More compiler specs can be found in the Spack compiler files in our repository: `scripts/spack/configs/<platform>/compilers.yaml`.

We currently regularly test the following Spack configuration files:

- Linux Ubuntu 18.04 (via Windows WSL 1)
- Linux Ubuntu 20.04 (via Windows WSL 2)
- TOSS 3 (On Quartz at LC)
- BlueOS (On Lassen at LC)

To install Serac on a new platform, it is a good idea to start with a known Spack configuration directory (located in the Serac repo at `scripts/spack/configs/<platform>`). The `compilers.yaml` file describes the compilers and associated flags required for the platform and the `packages.yaml` file describes the low-level libraries on the system to prevent Spack from building the world. Documentation on these configuration files is located in the Spack docs.

Some helpful uberenv options include :

- `--spec=+debug` (build the MFEM and Hypre libraries with debug symbols)
- `--spec=+profiling` (build the Adiak and Caliper libraries)
- `--spec=+devtools` (also build the devtools with one command)
- `--spec=%clang@10.0.0` (build with a specific compiler as defined in the `compiler.yaml` file)
- `--spack-config-dir=<Path to spack configuration directory>` (use specific Spack configuration files)
- `--prefix=<Path>` (required, build and install the dependencies in a particular location) - this *must be outside* of your local Serac repository

The modifiers to the Spack specification `spec` can be chained together, e.g. `--spec=%clang@10.0.0+debug+devtools`.

If you already have a Spack instance from another project that you would like to reuse, you can do so by changing the uberenv command as follows:

```
$ python3 scripts/uberenv/uberenv.py --upstream=</path/to/my/spack>/opt/spack
```

1.1.5 Building Serac's Dependencies by Hand

To build Serac's dependencies by hand, use of a `host-config` CMake configuration file is stongly encouraged. A good place to start is by copying an existing host config in the `host-config` directory and modifying it according to your system setup.

1.1.6 Using a Docker Image with Preinstalled Dependencies

As an alternative, you can build Serac using preinstalled dependencies inside a Docker container. Instructions for this process are located *here*.

1.1.7 Building Serac

Serac uses a CMake build system that wraps its configure step with a script called `config-build.py`. This script creates a build directory and runs the necessary CMake command for you. You just need to point the script at the generated or a provided host-config. This can be accomplished with one of the following commands:

```
# If you built Serac's dependencies yourself either via Spack or by hand
$ python3 ./config-build.py -hc <config_dependent_name>.cmake

# If you are on an LC machine and want to use our public pre-built dependencies
$ python3 ./config-build.py -hc host-configs/<machine name>-<SYS_TYPE>-<compiler>.
↪cmake

# If you'd like to configure specific build options, e.g., a release build
$ python3 ./config-build.py -hc /path/to/host-config.cmake -DCMAKE_BUILD_TYPE=Release
↪<more CMake build options...>
```

If you built the dependencies using Spack/uberenv, the host-config file is output at the project root. To use the pre-built dependencies on LC, you must be in the appropriate LC group. Contact Jamie Bramwell for access.

Some build options frequently used by Serac include:

- `CMAKE_BUILD_TYPE`: Specifies the build type, see the CMake docs
- `ENABLE_BENCHMARKS`: Enables Google Benchmark performance tests, defaults to `OFF`
- `ENABLE_WARNINGS_AS_ERRORS`: Turns compiler warnings into errors, defaults to `ON`
- `ENABLE_ASAN`: Enables the Address Sanitizer for memory safety inspections, defaults to `OFF`
- `SERAC_ENABLE_CODEVELOP`: Enables local development build of MFEM/Axom, see *Co-Developing with Axom and MFEM*, defaults to `OFF`

Once the build has been configured, Serac can be built with the following commands:

```
$ cd build-<system-and-toolchain>
$ make -j16
```

Note: On LC machines, it is good practice to do the build step in parallel on a compute node. Here is an example command: `srun -ppdebug -N1 --exclusive make -j16`

We provide the following useful build targets that can be run from the build directory:

- `test`: Runs our unit tests
- `docs`: Builds our documentation to the following locations:
  - Sphinx: `build-*/src/docs/html/index.html`
  - Doxygen: `/build-*/src/docs/html/doxygen/html/index.html`
- `style`: Runs styling over source code and replaces files in place
- `check`: Runs a set of code checks over source code (CppCheck and clang-format)

1.1.8 Preparing Windows WSL/Ubuntu for Serac installation

For faster installation of the Serac dependencies via Spack on Windows WSL/Ubuntu systems, install cmake, MPICH, openblas, OpenGL, and the various developer tools using the following commands:

Ubuntu 20.04

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install cmake libopenblas-dev libopenblas-base mpich mesa-common-dev␣
↪libglu1-mesa-dev freeglut3-dev cppcheck doxygen libreadline-dev python3-sphinx␣
↪python3-pip clang-format-10 m4
$ sudo ln -s /usr/lib/x86_64-linux-gnu/* /usr/lib
```

Ubuntu 18.04

```
$ sudo apt-get update
$ sudo apt-get upgrade
$ sudo apt-get install g++-8 gcc-8
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-8 800 --slave /usr/
↪bin/g++ g++ /usr/bin/g++-8
$ sudo apt-get install cmake libopenblas-dev libopenblas-base mpich mesa-common-dev␣
↪libglu1-mesa-dev freeglut3-dev cppcheck doxygen libreadline-dev python3-distutils␣
↪python3-pip
$ sudo ln -s /usr/lib/x86_64-linux-gnu/* /usr/lib
```

Note that the last line is required since Spack expects the system libraries to exist in a directory named `lib`. During the third party library build phase, the appropriate Spack config directory must be specified using either:

Ubuntu 20.04

```
python3 scripts/uberenv/uberenv.py --spack-config-dir=scripts/spack/configs/
linux_ubuntu_20 --prefix=../path/to/install
```

Ubuntu 18.04

```
python3 scripts/uberenv/uberenv.py --spack-config-dir=scripts/spack/configs/
linux_ubuntu_18 --prefix=../path/to/install
```

1.1.9 Building Serac dependencies on MacOS

Warning: These instructions are in development.

Meeting base dependency requirements

One way to install the required depedencies is with the MacPorts package manager. Install the required ports as follows:

```
$ sudo port install clang-12 openmpi-clang12 gcc12 bzip2 cmake autoconf automake␣
↪gettext graphviz pkgconfig xorg-libX11 lapack readline zlib
```

(Note: The port `gcc12` is included only to have a fortran compiler.)

Activate the particular compiler packages with MacPorts:

```
$ sudo port select clang mp-clang-12
$ sudo port select mpi openmpi-clang12-fortran
$ sudo port select gcc mp-gcc12
```

This step tells MacPorts to make symbolic links in your path so that, for example, the command `clang` will invoke the compiler installed by the MacPorts package and not the one shipped by Apple. It also sets up a set of symlinks so that Clang, GCC, and the MPI wrappers all work without you having to muck with environment variables to locate header files and libraries. It may be possible to skip this step and give full paths to your compilers in `compilers.yaml` (instead of the symlinks `/opt/local/bin/clang`, etc.), but we haven't tried this.

Note: If you want to remove these symlinks, use the `port select` command with `none` as the desired port; e.g., `sudo port select clang none`

While building ParMetis, Spack invokes the MPI compiler wrapper with the `mpic++` command, but MacPorts does not create this particular synonym for OpenMPI. It does create `mpicxx`. This can be solved by making a symlink:

```
$ cd /opt/local/bin
$ sudo ln -s mpicxx mpic++
```

MacPorts will automatically update the `mpicxx` symlink in `/opt/local/bin` to point to the correct executable when you use the `port select` command to activate a partiular MPI package. By making `mpic++` point to `mpicxx`, this command will also automatically point to the correct executable if you change the global MPI package through MacPorts in the future.

The Serac build scripts will install the `clingo` package in your Python environment (and may even *uninstall* it if it finds it with a version it considers too old). If you don't want the install to modify your Python environment, you may wish to conisder using tools like virtual environments or conda to isolate this change.

Configuring Spack

Next, you must tailor the Spack configuration files. We will modify the `compilers.yaml` and `packages.yaml` files in `scripts/spack/configs/darwin/`. Instead of modifying them directly, you may wish to copy these files to another location outside of the Serac repo, use them as templates for the customization, and use the `--spack-config-dir` option to use them when invoking uberenv as described above.

Example `compilers.yaml`:

```
compilers:
- compiler:
    environment: {}
    extra_rpaths: []
    flags: {}
    modules: []
    operating_system: bigsur
    paths:
      cc: /opt/local/bin/clang
      cxx: /opt/local/bin/clang++
      f77: /opt/local/bin/gfortran
      fc: /opt/local/bin/gfortran
    spec: clang@12.0.1
    target: x86_64
```

NOTES:

- The `operating_system` field should be set according to your macOS version. (For example, `mojave`, `catalina`, `bigsur`, `monterey`, `ventura`).
- By default, MacPorts installs packages in `/opt/local`; the above `paths` need to be adjusted if you choose a different location. This of course applies to the packages in `packages.yaml` as well.
- As noted above, the `port select ...` commands will set which version of clang gets invoked by the executables `/opt/local/bin/clang`, etc. The paths above are thus valid only if you activated the `clang` package that matches the compiler spec. Alternatively, you could set the full name and path of the executables of the desired compilers if you don't want the operation of Spack to be influenced by your MacPorts settings.
- You should set `spec` to the actual version of the compiler you installed.
- The `target` entry should be set to `x86_64` or `m1` depending on which architecture your machine uses.

Here is an example of `packages.yaml`:

```
packages:
  all:
    compiler: [clang, gcc]
    providers:
      blas: [netlib-lapack]
      lapack: [netlib-lapack]
      mpi: [openmpi]

  mpi:
    buildable: false
  openmpi:
    externals:
    - spec: openmpi@4.1.4
      prefix: /opt/local

  netlib-lapack:
    buildable: false
    externals:
    - spec: netlib-lapack@3.10.1
      prefix: /opt/local
  autoconf:
    buildable: false
    externals:
    - spec: autoconf@2.71
      prefix: /opt/local
```

(continues on next page)

(continued from previous page)

```
  automake:
    buildable: false
    externals:
    - spec: automake@1.16.5
      prefix: /opt/local
  bzip2:
    buildable: false
    externals:
    - spec: bzip2@1.0.8
      prefix: /opt/local
  cmake:
    version: [3.22.4]
    buildable: false
    externals:
    - spec: cmake@3.22.4
      prefix: /opt/local
  gettext:
    buildable: false
    externals:
    - spec: gettext@0.21
      prefix: /opt/local
  graphviz:
    buildable: false
    externals:
    - spec: graphviz@2.50.0
      prefix: /opt/local
  libtool:
    buildable: false
    externals:
    - spec: libtool@2.4.6
      prefix: /opt/local
  libx11:
    buildable: false
    externals:
    - spec: libx11@1.8.1
      prefix: /opt/local
  m4:
    buildable: false
    externals:
    - spec: m4@1.4.6
      prefix: /usr
  perl:
    buildable: false
    externals:
    - spec: perl@v5.30.2
      prefix: /usr
  pkg-config:
    buildable: false
    externals:
    - spec: pkg-config@0.29.2
      prefix: /opt/local
  tar:
    buildable: false
    externals:
    - spec: tar@3.3.2
      prefix: /usr
  readline:
```

(continues on next page)

(continued from previous page)

```
    buildable: false
    externals:
    - spec: readline@8.1.2.000
      prefix: /opt/local
  unzip:
    buildable: false
    externals:
    - spec: unzip@6.0
      prefix: /usr
  zlib:
    buildable: false
    externals:
    - spec: zlib@1.2.12
      prefix: /opt/local
```

Notes:

- The version specs should be set to the actual versions of the packages you have, which will not neccesarily be the same as the above. This can be discovered for the packages installed with MacPorts using the following command:

```
$ port info clang-12 openmpi-clang12 gcc12 bzip2 autoconf automake gettext graphviz␣
↪pkgconfig xorg-libX11 lapack readline zlib
```

- Use the version number provided, taking the values up to, but excluding, any underscore.
- The packages not installed by MacPorts are the ones that have `/usr` as the prefix. The versions already present on the system are sufficient for the build.

The above Spack settings and MacPorts packages will cover the basic installation of Serac. If you want to build the optional devtools, you should install the additional packages with MacPorts:

Then, append the following to `packages.yaml`:

```
cppcheck:
  version: [2.3]
  buildable: false
  externals:
  - spec: cppcheck@2.3
    prefix: /usr/local
doxygen:
  version: [1.8.13]
  buildable: false
  externals:
  - spec: doxygen@1.8.13
    prefix: /usr/local
llvm:
  version: [10.0.0]
  buildable: false
  externals:
  - spec: llvm+clang@10.0.0
    prefix: <path/to/llvm/10>
python:
  buildable: false
  externals:
  - spec: python@3.9
    prefix: <path/to/python/venv>
```

Notes:

- LLVM/Clang is needed for the style check tools. The *exact* version 10.0.0 is apparently highly recommended, since other versions may format the code slightly differently, which will mean that pull requests formatted with them may trigger style errors in the CI checks.
- The placeholders `<path/to/llvm/10>` and `<path/to/python/venv>` need to be filled in with actual paths. See the following two notes.
- LLVM 10.0.0 has been superseded as the version for llvm-10; so this package is not easily installable with MacPorts. You must build it yourself and then point to the build location.
- For `<path/to/python/venv>`, specify the the virtual environment directory created above.

Building dependencies

The invocation of `uberenv.py` is slightly modified from the standard instructions above in order to force the use of the MacPorts-installed MPI:

```
$ ./scripts/uberenv/uberenv.py --prefix=../path/for/TPLs --spec="%clang@12.0.1 ^
↪openmpi@4.1.4"
```

Notice the caret with the MPI spec. Without this, current versions of Spack ignore the `packages.yaml` file and try to build a version of MPI from source. You can add additional specs as noted in the section *Building Serac's Dependencies via Spack/uberenv*.

1.2 User Guide

1.2.1 Simple Heat Transfer Tutorial

This tutorial provides an introduction to running simulations with Serac and demonstrates the setup of a simple steady-state thermal conduction problem. The tutorial also provides a side-by-side comparison of Serac's simulation configuration methods, namely, the C++ API and Lua input files.

The full source code for this tutorial is available in `examples/simple_conduction/with_input_file.cpp` and `examples/simple_conduction/without_input_file.cpp`, which demonstrate Lua and C++ configuration, respectively. The input file used for the Lua configuration is `examples/simple_conduction/conduction.lua`.

The thermal conduction modeled in this section is based on the formulation discussed in *Heat Transfer*.

Setting Up Includes and Initializing

The most important parts of Serac are its physics modules, each of which corresponds to a particular discretization of a partial differential equation (e.g., continuous vs. discontinuous Galerkin finite element methods). In this example, we are building a thermal conduction simulation, so we include Serac's thermal conduction module:

```
#include "serac/physics/thermal_conduction_legacy.hpp"
```

The following header provides access to the *StateManager* class which manages the individual finite element states and the mesh:

```
#include "serac/physics/state/state_manager.hpp"
```

Serac also provides a set of setup/teardown functions that encapsulate much of the boilerplate setup required for each simulation, e.g., MPI initialization/finalization and logger setup/teardown, so we include their headers:

```
#include "serac/infrastructure/initialize.hpp"
#include "serac/infrastructure/terminator.hpp"
```

Finally, we include the header for Serac's mesh utilities, which includes support for reading meshes from a file and for generating meshes of common solids, like cuboids, rectangles, disks, and cylinders:

```
#include "serac/mesh/mesh_utils.hpp"
```

We're now ready to start our `main()` function by initializing Serac, which performs the setup described above:

```
int main(int argc, char* argv[])
{
  /*auto [num_procs, rank] = */serac::initialize(argc, argv);
  axom::sidre::DataStore datastore;
  serac::StateManager::initialize(datastore, "without_input_file_example");
```

To simplify saving to an output file and restarting a simulation, Serac stores critical state information, like the mesh and fields, in a single `StateManager` object, which is initialized here.

Warning: Since Serac's initialization helper initializes MPI, you should not call `MPI_Init` directly.

Setting Up Inlet

This section is specific to configuration with Lua input files. If you're just interested in using the C++ API, you can skip to *Constructing the Mesh*.

Serac uses Axom's Inlet component for defining and extracting information from input files. Inlet is based on Axom's Sidre component, which provides a uniform in-memory layout for simulation data. We instantiate Inlet with a DataStore instance (the top-level Sidre building block) and the path to the Lua input file, which in this case is `examples/simple_conduction/conduction.lua`:

```
  axom::sidre::DataStore datastore;
  serac::StateManager::initialize(datastore, "with_input_file_example");
  auto inlet = serac::input::initialize(datastore, input_file);
```

We then define the schema for the input file. Instead of defining the structure of the input file in one place, Inlet allows Serac to separate its schema definition logic into functions that are responsible for defining just one component of the schema. Since our input file contains information required for mesh construction, for Serac's `ThermalConduction` module, we use Inlet to define the corresponding schemas:

```
  auto& mesh_schema = inlet.addStruct("main_mesh", "The main mesh for the problem");
  serac::mesh::InputOptions::defineInputFileSchema(mesh_schema);

  auto& thermal_schema = inlet.addStruct("thermal_conduction", "Thermal conduction␣
↪module");
  serac::ThermalConductionLegacy::InputOptions::defineInputFileSchema(thermal_schema);
```

Note: Since Serac's schema definition functions are independent of their location in the input file, we add a struct whose name corresponds to the location in `conduction.lua` and pass that to the appropriate schema definition function.

Because input file parsing happens when the schema is defined, there is no need to call a separate `parse` function. We conclude Inlet's setup by calling the `verify` method, which ensures that all required data is present and meets any other constraints:

```
  SLIC_ERROR_ROOT_IF(!inlet.verify(), "Input file contained errors");
```

Hint: The `SLIC_ERROR_ROOT_IF` macro is part of Serac's *Logging* functionality, which is enabled as part of the `serac::initialize` function described above.

Constructing the Mesh

In this introductory example, we will use a simple square mesh with 10 quadrilateral elements in each space dimension for 100 elements total. Once created, the primary mesh must always be registered with the `StateManager`:

Using C++

```
  auto mesh = serac::mesh::refineAndDistribute(serac::buildRectangleMesh(10, 10));
  serac::StateManager::setMesh(std::move(mesh));
```

After constructing the serial mesh, we call `refineAndDistribute` to distribute it into a parallel mesh.

Using Lua

```
  auto mesh_options = inlet["main_mesh"].get<serac::mesh::InputOptions>();
  auto mesh = serac::mesh::buildParallelMesh(mesh_options);
  serac::StateManager::setMesh(std::move(mesh));
```

This snippet queries Inlet's internal hierarchy at the location where we defined the mesh schema (`main_mesh`), and reads the data into the `struct` Serac uses for storing mesh creation options, which is all we need to construct the mesh. The Lua representation is as follows:

```
main_mesh = {
    type = "box",
    elements = {x = 10, y = 10}
}
```

Constructing the Physics Module

Using C++

```
  constexpr int order = 2;
  serac::ThermalConductionLegacy conduction(order,␣
↪serac::ThermalConductionLegacy::defaultQuasistaticOptions());
```

When using the C++ API, the `ThermalConduction` constructor requires the polynomial order of the elements and the solver options to be used when inverting the stiffness matrix, in addition to the mesh. Since we're setting up a steady-state simulation, we can just use the `defaultQuasistaticOptions`.

Using Lua

Once the configuration options are read from Inlet, we can use them to construct the `ThermalConduction` object:

```
  auto conduction_opts = inlet["thermal_conduction"].get
↪<serac::ThermalConductionLegacy::InputOptions>();
  serac::ThermalConductionLegacy conduction(conduction_opts);
```

Unlike the C++-only version, we don't need to specify the order or the solver options in the constructor because they're in the input file:

```
    equation_solver = {
        linear = {
            type = "iterative",
            iterative_options = {
                rel_tol     = 1.0e-6,
                abs_tol     = 1.0e-12,
                max_iter    = 200,
                -- WARNING: Undocumented in MFEM, but ranges from -1 to 3,
                -- -1 is no output, 0 is warnings/errors only,
                -- 1 is per-iteration norm only, 2 is final iteration count only,
                -- 3 includes extra debug info
                print_level = 0,
                solver_type = "cg",
                prec_type   = "JacobiSmoother",
            },
        },

        nonlinear = {
            rel_tol     = 1.0e-4, -- w.r.t. residual
            abs_tol     = 1.0e-8,
            max_iter    = 500,
            -- WARNING: Undocumented in MFEM, but ranges from -1 to 3,
            -- -1 is no output, 0 is warnings/errors only,
            -- 1 is per-iteration norm only, 2 is final iteration count only,
            -- 3 includes extra debug info
            print_level = 1,
        },
    },

    order = 2,
```

Configuring the Physics Module

The following sections demonstrate a subset of the configuration options available with Serac's `ThermalConduction` module.

Note: The C++-only API requires method calls for configuration, while the Lua-based approach typically only requires changes to the input file. This is because the configuration options are all part of the `ThermalConduction::InputOptions` extracted from Inlet.

Configuring Material Conductivity

Instead of using a monolithic material model, the `ThermalConduction` module currently allows for material parameters like conductivity, specific heat capacity, and density to be configured individually.

Using C++

```
  constexpr double kappa = 0.5;
  auto kappa_coef = std::make_unique<mfem::ConstantCoefficient>(kappa);
  conduction.setConductivity(std::move(kappa_coef));
```

Using Lua

```
    kappa = 0.5,
```

Setting Thermal (Dirichlet) Boundary Conditions

The following snippets add two Dirichlet boundary conditions:

- One that constrains the temperature to 1.0 at boundary attribute 1, which for this mesh corresponds to the side of the square mesh along the x-axis, i.e., the "bottom" of the mesh.
- One that constrains the temperature to $x^2 + y - 1$ at boundary attributes 2 and 3, which for this mesh correspond to the right side and top of the mesh, respectively.

Using C++

```
  const std::set<int> boundary_constant_attributes = {1};
  constexpr double boundary_constant = 1.0;
  auto boundary_constant_coef = std::make_unique<mfem::ConstantCoefficient>(boundary_
↪constant);
  conduction.setTemperatureBCs(boundary_constant_attributes, std::move(boundary_
↪constant_coef));

  const std::set<int> boundary_function_attributes = {2, 3};
  auto boundary_function_coef = std::make_unique<mfem::FunctionCoefficient>([](const␣
↪mfem::Vector& vec){
    return vec[0] * vec[0] + vec[1] - 1;
  });
  conduction.setTemperatureBCs(boundary_function_attributes, std::move(boundary_
↪function_coef));
```

Using Lua

```
    boundary_conds = {
        ['temperature_1'] = {
            attrs = {1},
            constant = 1.0
        },
        ['temperature_2'] = {
            attrs = {2, 3},
            scalar_function = function(v)
                return v.x * v.x + v.y - 1
            end
        }
    },
```

Note: The exact names here are not critical, any entry whose name contains the string `temperature` will be applied as a Dirichlet condition to the temperature field.

Running the Simulation

Now that we've configured the `ThermalConduction` instance using a few of its configuration options, we're ready to run the simulation. We call `completeSetup` to "finalize" the simulation configuration, and then save off the initial state of the simulation. This also allocates and builds all of the internal finite element data structures.

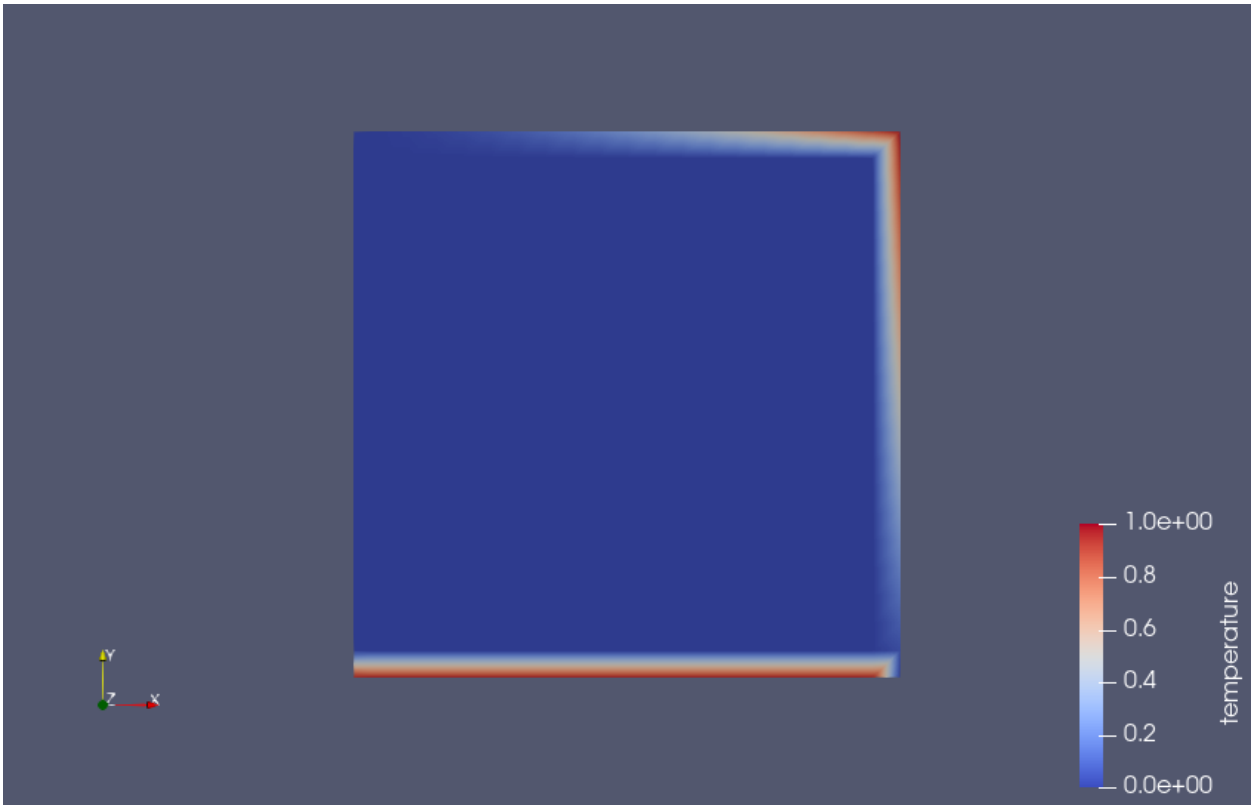
We can then perform the steady-state solve and save the end result:

```
conduction.completeSetup();
conduction.outputState();

double dt;
conduction.advanceTimestep(dt);
conduction.outputState();
```

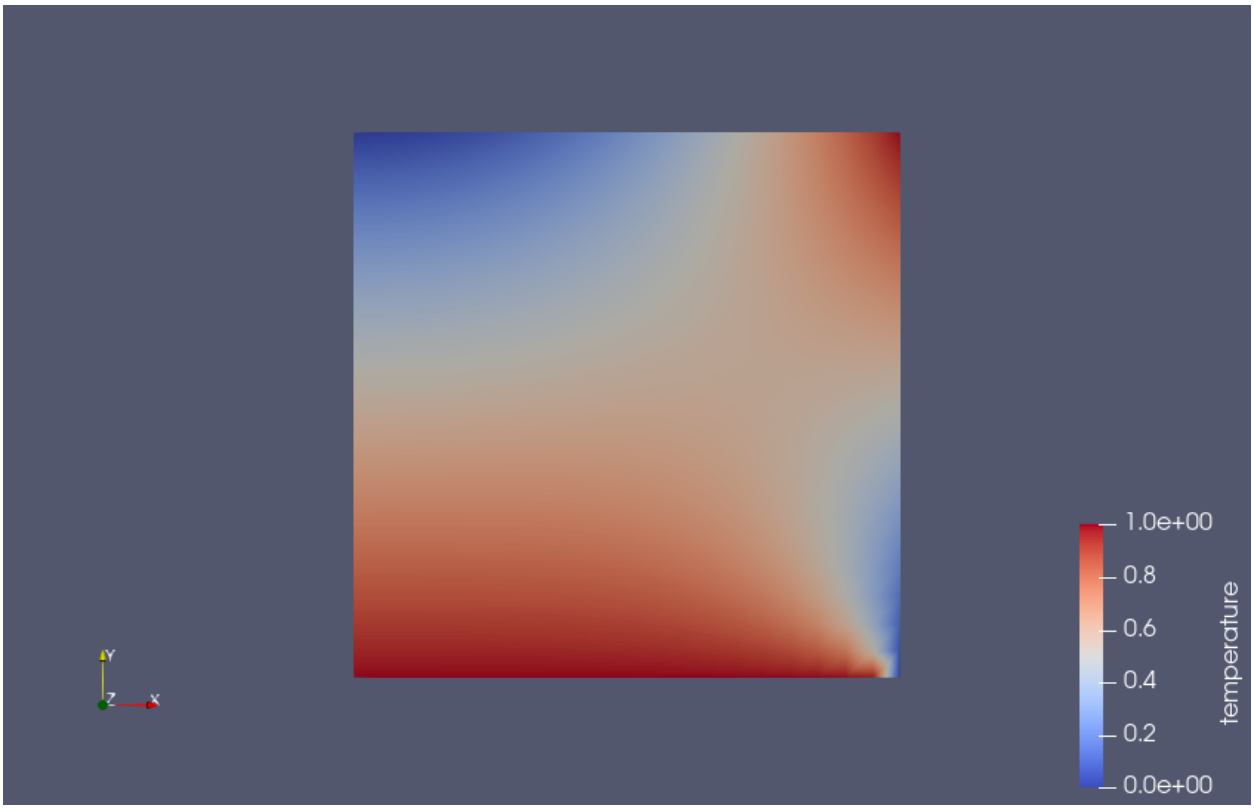
Note: The `dt` variable does not actually get used in a quasistatic simulation.

This should produce the following initial state:



Note the areas of the mesh where the boundary conditions were applied.

The final (steady-state) solution should look like the following:



The end result is not particularly impressive, but should be fairly intuitive.

Cleaning Up

To make sure Serac terminates properly, don't forget to call its exit function at the very end of your program:

```
  serac::exitGracefully();
}
```

1.2.2 Command Line Options

Below is the documentation for Serac's command line options:

Table 1: Options

Long form	Short form	Variable Type	Description
–help	-h	N/A	Print this help message and exit
–input-file	-i	Path	Input file to use
–restart-cycle	-c	Integer	Cycle to restart from
–create-input-file-docs	-d	N/A	Writes Sphinx documentation for input file, then exits
–output-directory	-o	Path	Directory to put outputted files
–paraview	-p	N/A	Enable ParaView output
–version	-v	N/A	Print version and provenance information, then exits

1.2.3 Input File Structure

Below is the documentation for Serac input files, generated automatically by Axom's inlet component.

input_file

Table 2: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
dt	Time step.	0.250000		
t_final	Final time for simulation.	1.000000		

thermal_solid

Description: Thermal solid module

coef_thermal_expansion

Description: Coefficient of thermal expansion for isotropic thermal expansion

Table 3: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 4: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 5: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-component of vector			
y	y-component of vector			
x	x-component of vector			

reference_temperature

Description: Coefficient for the reference temperature for isotropic thermal expansion

Table 6: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 7: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 8: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-component of vector			
y	y-component of vector			
x	x-component of vector			

thermal_conduction

Description: Thermal conduction module

Table 9: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
rho	Density	1.000000		
cp	Specific heat capacity	1.000000		
kappa	Thermal conductivity	0.500000		
order	Order degree of the finite elements.	1	1 to 8	

boundary_conds

Collection contents:

source

Description: Scalar source term (RHS of the thermal conduction PDE)

Table 10: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 11: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 12: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-component of vector			
y	y-component of vector			
x	x-component of vector			

initial_temperature

Description: Coefficient for initial condition

Table 13: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 14: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 15: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
y	y-component of vector			
z	z-component of vector			
x	x-component of vector			

nonlinear_reaction

Description: Nonlinear reaction term parameters

Table 16: Functions

Function Name	Description	Signature	Required
d_reaction_function	Derivative of the nonlinear reaction function dq = dq / dTemperature	Double(Double)	
reaction_function	Nonlinear reaction function q = q(temperature)	Double(Double)	

scale

Description: Spatially varying scale factor for the reaction

Table 17: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 18: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 19: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-component of vector			
y	y-component of vector			
x	x-component of vector			

equation_solver

Description: Linear and Nonlinear stiffness Solver Parameters.

nonlinear

Description: Newton Equation Solver Parameters

Table 20: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
solver_type	Solver type (MFEMNewton|KINFullStep|KINLineSearch)	MFEMNewton		
rel_tol	Relative tolerance for the Newton solve.	0.010000		
abs_tol	Absolute tolerance for the Newton solve.	0.000100		
max_iter	Maximum iterations for the Newton solve.	500		
print_level	Nonlinear print level.	0		

linear

Description: Linear Equation Solver Parameters

Table 21: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
type	The type of solver parameters to use (iterative|direct)		iterative, direct	

direct_options

Description: Direct solver parameters

Table 22: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
print_level	Linear print level.	0		

iterative_options

Description: Iterative solver parameters

Table 23: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
prec_type	Preconditioner type (JacobiSmoother|L1JacobiSmoother|AMG|BlockILU).	JacobiSmoother		
solver_type	Solver type (gmres|minres|cg).	gmres		
rel_tol	Relative tolerance for the linear solve.	0.000001		
abs_tol	Absolute tolerance for the linear solve.	0.000000		
max_iter	Maximum iterations for the linear solve.	5000		
print_level	Linear print level.	0		

dynamics

Description: Parameters for mass matrix inversion

Table 24: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
enforcement_method	Time-varying constraint enforcement method to use			
timestepper	Timestepper (ODE) method to use			

solid

Description: Finite deformation solid mechanics module

Table 25: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
viscosity	Viscosity constant	0.000000		
density	Initial mass density	1.000000		
material_nonlin	Flag to include material nonlinearities (linear elastic vs. neo-Hookean material model).	True		
order	Order degree of the finite elements.	1	1 to 8	
mu	Shear modulus in the Neo-Hookean hyperelastic model.	0.250000		
geometric_nonlin	Flag to include geometric nonlinearities in the residual calculation.	True		
K	Bulk modulus in the Neo-Hookean hyperelastic model.	5.000000		

initial_velocity

Description: Coefficient for initial condition

Table 26: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 27: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 28: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-component of vector			
y	y-component of vector			
x	x-component of vector			

equation_solver

Description: Linear and Nonlinear stiffness Solver Parameters.

nonlinear

Description: Newton Equation Solver Parameters

Table 29: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
solver_type	Solver type (MFEMNewton|KINFullStep|KINLineSearch)	MFEMNewton		
rel_tol	Relative tolerance for the Newton solve.	0.010000		
abs_tol	Absolute tolerance for the Newton solve.	0.000100		
print_level	Nonlinear print level.	0		
max_iter	Maximum iterations for the Newton solve.	500		

linear

Description: Linear Equation Solver Parameters

Table 30: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
type	The type of solver parameters to use (iterative|direct)		iterative, direct	

direct_options

Description: Direct solver parameters

Table 31: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
print_level	Linear print level.	0		

iterative_options

Description: Iterative solver parameters

Table 32: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
prec_type	Preconditioner type (JacobiSmoother|L1JacobiSmoother|AMG|BlockILU).	JacobiSmoother		
solver_type	Solver type (gmres|minres|cg).	gmres		
rel_tol	Relative tolerance for the linear solve.	0.000001		
abs_tol	Absolute tolerance for the linear solve.	0.000000		
max_iter	Maximum iterations for the linear solve.	5000		
print_level	Linear print level.	0		

boundary_conds

Collection contents:

dynamics

Description: Parameters for mass matrix inversion

Table 33: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
enforcement_method	Time-varying constraint enforcement method to use			
timestepper	Timestepper (ODE) method to use			

initial_displacement

Description: Coefficient for initial condition

Table 34: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 35: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 36: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-component of vector			
y	y-component of vector			
x	x-component of vector			

thermal_conduction

Description: Thermal conduction module

Table 37: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
cp	Specific heat capacity	1.000000		
rho	Density	1.000000		
kappa	Thermal conductivity	0.500000		
order	Order degree of the finite elements.	1	1 to 8	

initial_temperature

Description: Coefficient for initial condition

Table 38: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 39: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 40: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-component of vector			
y	y-component of vector			
x	x-component of vector			

boundary_conds

Collection contents:

The input schema defines a collection of this container. For brevity, only one instance is displayed here.

Table 41: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 42: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 43: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-component of vector			
y	y-component of vector			
x	x-component of vector			

attrs

Collection contents:

Table 44: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
1				

source

Description: Scalar source term (RHS of the thermal conduction PDE)

Table 45: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 46: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 47: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-component of vector			
y	y-component of vector			
x	x-component of vector			

nonlinear_reaction

Description: Nonlinear reaction term parameters

Table 48: Functions

Function Name	Description	Signature	Required
d_reaction_function	Derivative of the nonlinear reaction function dq = dq / dTemperature	Double(Double)	
reaction_function	Nonlinear reaction function q = q(temperature)	Double(Double)	

scale

Description: Spatially varying scale factor for the reaction

Table 49: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 50: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 51: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-component of vector			
y	y-component of vector			
x	x-component of vector			

dynamics

Description: Parameters for mass matrix inversion

Table 52: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
enforcement_method	Time-varying constraint enforcement method to use			
timestepper	Timestepper (ODE) method to use			

equation_solver

Description: Linear and Nonlinear stiffness Solver Parameters.

nonlinear

Description: Newton Equation Solver Parameters

Table 53: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
rel_tol	Relative tolerance for the Newton solve.	0.010000		
solver_type	Solver type (MFEMNewton|KINFullStep|KINLineSearch)	MFEMNewton		
abs_tol	Absolute tolerance for the Newton solve.	0.000100		
max_iter	Maximum iterations for the Newton solve.	500		
print_level	Nonlinear print level.	0		

linear

Description: Linear Equation Solver Parameters

Table 54: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
type	The type of solver parameters to use (iterative|direct)		iterative, direct	

direct_options

Description: Direct solver parameters

Table 55: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
print_level	Linear print level.	0		

iterative_options

Description: Iterative solver parameters

Table 56: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
prec_type	Preconditioner type (JacobiSmoother|L1JacobiSmoother|AMG|BlockILU).	JacobiSmoother		
solver_type	Solver type (gmres|minres|cg).	gmres		
rel_tol	Relative tolerance for the linear solve.	0.000001		
max_iter	Maximum iterations for the linear solve.	5000		
abs_tol	Absolute tolerance for the linear solve.	0.000000		
print_level	Linear print level.	0		

solid

Description: Finite deformation solid mechanics module

Table 57: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
density	Initial mass density	1.000000		
viscosity	Viscosity constant	0.000000		
material_nonlin	Flag to include material nonlinearities (linear elastic vs. neo-Hookean material model).	True		
order	Order degree of the finite elements.	1	1 to 8	
K	Bulk modulus in the Neo-Hookean hyperelastic model.	5.000000		
mu	Shear modulus in the Neo-Hookean hyperelastic model.	0.250000		
geometric_nonlin	Flag to include geometric nonlinearities in the residual calculation.	True		

initial_velocity

Description: Coefficient for initial condition

Table 58: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 59: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 60: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-component of vector			
y	y-component of vector			
x	x-component of vector			

equation_solver

Description: Linear and Nonlinear stiffness Solver Parameters.

nonlinear

Description: Newton Equation Solver Parameters

Table 61: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
solver_type	Solver type (MFEMNewton|KINFullStep|KINLineSearch)	MFEMNewton		
rel_tol	Relative tolerance for the Newton solve.	0.010000		
abs_tol	Absolute tolerance for the Newton solve.	0.000100		
max_iter	Maximum iterations for the Newton solve.	500		
print_level	Nonlinear print level.	0		

linear

Description: Linear Equation Solver Parameters

Table 62: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
type	The type of solver parameters to use (iterative|direct)		iterative, direct	

direct_options

Description: Direct solver parameters

Table 63: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
print_level	Linear print level.	0		

iterative_options

Description: Iterative solver parameters

Table 64: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
prec_type	Preconditioner type (JacobiSmoother|L1JacobiSmoother|AMG|BlockILU).	JacobiSmoother		
solver_type	Solver type (gmres|minres|cg).	gmres		
rel_tol	Relative tolerance for the linear solve.	0.000001		
max_iter	Maximum iterations for the linear solve.	5000		
print_level	Linear print level.	0		
abs_tol	Absolute tolerance for the linear solve.	0.000000		

dynamics

Description: Parameters for mass matrix inversion

Table 65: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
enforcement_method	Time-varying constraint enforcement method to use			
timestepper	Timestepper (ODE) method to use			

boundary_conds

Collection contents:

The input schema defines a collection of this container. For brevity, only one instance is displayed here.

Table 66: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 67: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 68: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
y	y-component of vector			
z	z-component of vector			
x	x-component of vector			

attrs

Collection contents:

Table 69: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
1				

initial_displacement

Description: Coefficient for initial condition

Table 70: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
constant	The constant scalar value to use as the coefficient			
component	The vector component to which the scalar coefficient should be applied			

Table 71: Functions

Function Name	Description	Signature	Required
scalar_function	The function to use for an mfem::FunctionCoefficient	Double(Vector, Double)	
vector_function	The function to use for an mfem::VectorFunctionCoefficient	Vector(Vector, Double)	

vector_constant

Description: The constant vector to use as the coefficient

Table 72: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-component of vector			
y	y-component of vector			
x	x-component of vector			

main_mesh

Description: The main mesh for the problem

Table 73: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
approx_elements	Approximate number of elements in an n-ball mesh			
ser_ref_levels	Number of times to refine the mesh uniformly in serial.	0		
mesh	Path to Mesh file			
par_ref_levels	Number of times to refine the mesh uniformly in parallel.	0		
type	Type of mesh		ball, box, disk, file	

size

Table 74: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
y	Size in the y-dimension			
z	Size in the z-dimension			
x	Size in the x-dimension			

elements

Table 75: Fields

Field Name	Description	Default Value	Range/Valid Values	Required
z	z-dimension			
y	y-dimension			
x	x-dimension			

Serac can be used either by providing input files to the main executable or through a C++ API. Example lua input files are located in the data directory and examples of how to use the C++ API are located in the tests directory.

1.2.4 Physics Module C++ Interface

A fundamental data structure in Serac is BasePhysics. Classes derived from `BasePhysics` are expected to encapsulate a specific partial differential equation and all of the state data and parameters associated with it. Currently, Serac contains the following physics modules:

- Solid mechanics
- Thermal conduction
- Thermal solid mechanics

If you would like to include Serac's simulation capabilities in your software project, these are the classes to include. To set up and use a physics module:

1. Construct the appropriate physics module class using a `mfem::ParMesh` and a polynomial order of approximation.
2. Set the material properties via `mfem::Coefficients`.
3. Set the boundary conditions via a `std::set` of boundary attributes and a `mfem::Coefficient`.
4. Set the right hand side source terms (e.g. body forces).
5. Set the time integration scheme (e.g. quasi-static or backward Euler). Note that not all time integrators are available for all physics modules.
6. Complete the setup of the physics module by calling `completeSetup()`. This allocates and builds all of the underlying linear algebra data structures.
7. Advance the timestep by calling `advanceTimestep(double dt)`.

8. Output the state variables in VisIt, and optionally ParaView, format by calling `outputState()`. You can also access the underlying state data via the generic `getState()` or physics-specific calls (e.g. `temperature()`).

1.3 Developer Guide

1.3.1 Style Guide

Code Style

This project follows Google's C++ Style Guide with the following amendments:

1. `camelCase` should be used for function names
2. `ALL_CAPS` should be used for constants (in addition to macros)

If a class/function could feasibly be upstreamed to MFEM or implements an MFEM interface, it should be part of the `serac::mfem_ext` namespace and use MFEM's `PascalCase` naming convention.

The Google style guide is meant for style enforcement only. The design principles outlined in the C++ Core Guidelines should be followed.

Of particular importance are the guidelines proposed for managing object lifetime:

1. Use raw pointers and references [only] to denote non-ownership (R.3 - R.4)
2. Prefer `unique_ptr` to `shared_ptr` (R.21)

For example, if an object `A` creates a subobject `B` whose constructor requires a reference to one of `A`'s instance variables `Foo f`, `B` should store a non-owning reference to `f`, either `Foo&` or `Foo*`. This should be `const` if at all possible. In this case, shared ownership is not required because the lifetime of `B` is entirely dependent on the lifetime of `A`.

Documentation

Functions, structs, classes, and critical member variables should be annotated with Doxygen comments. These comments should be enclosed in Javadoc-style comment blocks. For example, a variable can be documented as follows:

```
/**
 * The MPI communicator
 */
MPI_Comm m_comm;
```

When annotating code, especially functions, Doxygen's special commands can come in handy to provide additional information:

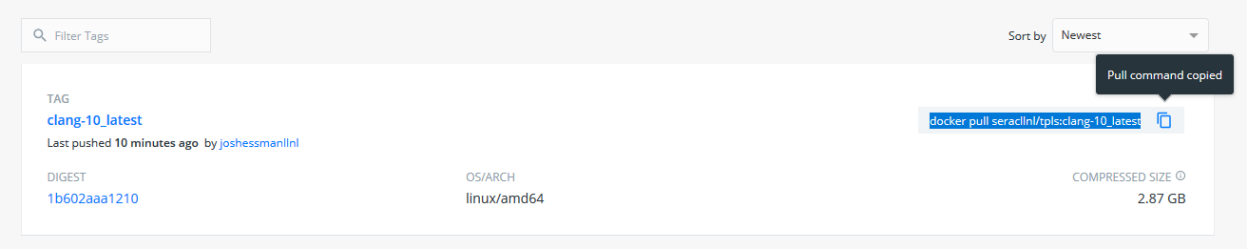
```
/**
 * Calculate du_dt = M^-1 (-Ku + f).
 * This is all that is needed for explicit methods
 * @param[in] u The state vector (input to the differentiation)
 * @param[out] du The derivative of @p u with respect to time
 * @see https://mfem.github.io/doxygen/html/classmfem_1_1TimeDependentOperator.html
 */
virtual void Mult(const mfem::Vector &u, mfem::Vector &du_dt) const;
```

For non-`void` functions, the `@return` command should be used to describe the return value.

1.3.2 Using a Docker Image for Development

If you haven't used Docker before, it is recommended that you check out the Docker tutorial before proceeding.

1. Clone a copy of the Serac repo to your computer: `git clone --recursive https://github.com/LLNL/serac.git`
2. Once you've installed `docker`, navigate to our Dockerhub page and select the most recent image corresponding to the compiler you'd like to use. Clang 10 and GCC 8 images are currently offered.
3. Copy the pull command corresponding to the image you've selected (in this case, it's `docker pull seracllnl/tpls:clang-10_latest`):



4. Next, run the copied command. Our images are around 2.5 GB, so it may take a while for the image to be downloaded to your machine. When the download completes, you will see something like the following:

```
110  ffb110ee0241: Pull complete
111  3b9a3928c208: Pull complete
112  72783f915408: Pull complete
113  3037fd02d56a: Pull complete
114  85774a0705c9: Pull complete
115  2032a5147bcb: Pull complete
116  6b76dc12b07a: Pull complete
117  Digest: sha256:1b602aaa121004d38d09adc162bfa4d041db8b2d6917613fa5c019654da31325
118  Status: Downloaded newer image for seracllnl/tpls:clang-10_latest
119  docker.io/seracllnl/tpls:clang-10_latest
120  15a940775c7938e0b9aa0d63fb1af539eabe0766b9ac592b94fb636d6faeb8b5
```

5. You can now run the image. Run `docker run -it -u serac -v /your/serac/repo:/home/serac/serac seracllnl/tpls:clang-10_latest bash`, replacing the tag (the compiler name following the `tpls:`) with the tag you used in the `docker pull` command and replacing `/your/serac/repo` with the path to the Serac repo you cloned in the first step. This will open a terminal into the image.

Note: The `-v` option to `docker run` mounts a Docker volume into the container. This means that part of your filesystem (in this case, your copy of the Serac repo) will be accessible from the container.

6. Follow the build instructions detailed in the *quickstart guide*, using the host-config in `host-configs/docker` that corresponds to the compiler you've selected. These commands should be run using the terminal you opened in the previous step. Due to issues with the docker bind-mount permissions, it is suggested that you set the build and install directories to be outside of the repository.

```
$ cd /home/serac/serac
$ python ./config-build.py -hc host-configs/docker/<container-host-config>.cmake -
↪bp ../build -ip ../install
```

(continues on next page)

(continued from previous page)

```
$ cd ../build
$ make -j4
$ make test
```

7. You can now make modifications to the code from your host machine (e.g., via a graphical text editor), and use the Docker container terminal to recompile/run/test your changes.

1.3.3 Frequently Used Modern C++ Features

Serac currently uses C++17. Several modern C++ features and library components are used heavily throughout Serac.

Smart pointers are used to avoid directly using `operator new` and `operator delete` except when absolutely necessary. `std::unique_ptr<T>` is used to denote **exclusive** ownership of a pointer to `T` - see this article for more info. Because `unique_ptr` implies unique/exclusive ownership, instances cannot be copied. For example, if a function has a `unique_ptr` argument, a caller must utilize *move semantics* to transfer ownership at the call site. The linked article provides an example of this, and move semantics are discussed in a more general sense here.

`std::shared_ptr<T>` is used to denote **shared** ownership of a pointer to `T` - see this article for example uses. `shared_ptr` s should be used sparingly. Often, when two objects need to share a resource, it is sufficient for only one of the objects to be responsible for the lifetime of the shared resource; the other object can store a reference to the resource.

`std::optional<T>` is used to express the idea of `Maybe T`, a.k.a. a nullable type. An `optional` is optionally a `T`, which is useful as a return type for functions that can fail. It is preferable to values that are implied to be invalid or represent failure, e.g., `std::optional<int>` should be used instead of -1 to represent an invalid array index. It is also preferred as an alternative to functions that return `nullptr` on failure. You can read more about `optional` here.

`std::variant<T1, T2, T3, ...>` is use to express the idea of `Either T1 or T2 or T3 or ...`. It is the type- and memory-safe version of a `union`. This article goes into more detail, but typically this is used to "tie" together classes that are used in the same context but are not conducive to an inheritance hierarchy.

Lambdas are also used frequently to declare small functions immediately before they are used, e.g., before they are passed to another function. Lambdas are very useful with `std::algorithm` s (introduced here), which are often preferable to traditional `for` loops as they more clearly express intent. Lambdas can also *capture* variables available in the scope in which they are declared - see this page for more info.

Finally, range-based `for` loops (described here) should be used whenever possible instead of integer-iterator-based indexing. This is supported for all standard library containers.

For a comprehensive overview of modern C++ (C++11 onwards), Scott Meyer's *Effective Modern C++* is quite useful.

1.3.4 Testing

Serac has two levels of tests, unit and integration. Unit tests are used to test individual components of code, such as a class or function. While integration tests are for testing the code as a whole. For example, testing the *serac* driver with an input file against blessed answers.

Unit Tests

Unit Tests can be ran via the build target `test` after building the code.

Integration Tests

Note: Integration testing is in development and not fully featured.

Requirements:

- Installed ATS
- `ATS_EXECUTABLE` defined in the host-config (added automatically to Spack generated host-configs) or on command line via `-DATS_EXECUTABLE=/path/to/ats`.
- If using a personal machine, check the `ats-config` directory in the serac repo and create a json file `<your_machine_name>.json` if you haven't already. Your machine's name can be found by running the following lines of code:

```
$ python3
>>> import socket
>>> socket.gethostname().rstrip('1234567890')
>>> exit()
```

  Currently, there are configuration json files for Toss3 and BlueOS which can be used as reference.

1. **Build the code.** Build code with the normal steps. More info in the *Quickstart Guide*. This generates a script in the build directory called `ats.sh`.
2. **Run integration tests.** Run the corresponding command for the system you are on:

```
# BlueOS
$ lalloc 2 ./ats.sh

# Toss3
$ salloc -N2 ./ats.sh

# Personal Machine (currently runs subset of tests)
$ ./ats.sh
```

   Append `--help` to the command to see the current options.

3. **View results.** ATS gives a running summary and the final results. ATS also outputs the following helpful files in the platform and timestamp specific created log directory:
   - `ats.log` - All output of ATS
   - `atss.log` - Short summary of the run
   - `atsr.xml` - JUnit test summary

   ATS also outputs both a `.log` and `.log.err` for each test and checker that is run.

Installing ATS

ATS can be installed via the normal devtools install process. More info on *Building Serac's Developer Tools*. This method is useful because it builds all development tools in one process.

If you want to install ATS by itself, ATS provides multiple methods to install in their Getting Started section.

ATS Test Helper Functions

We provide the following test helper functions to make defining integration tests easier in `tests/test.ats`.

- `tolerance_test`

```
Adds a test and dependent tolerance check for the given arguments

Parameters
----------
name : str
    The name of the test
input_file : str
    Path to the input file
num_mpi_tasks : int, optional
    The number of MPI tasks (default is 1)
tolerance : str, optional
    The allowed tolerance in test results (default is 0.00001)
    Note: If using options 2 or 3 a "default" value is required unless you
↪'ve
          specified all parameters
    Three options:
        1.) Either single value such as 0.00001
        2.) Multiple values in this format: default:0.1,velocity:0.22
        3.) Path to a tolerance JSON file
restart_cycle: int, optional
    The cycle to run from and test
    Note: If given a value, it will perform a tolerance test for both the␣
↪normal
          serac run and also the restart serac run using the specified␣
↪cycle
    Note: If ats.sh has the baseline option set, it will copy the restart
↪'s summary
          file to the restart baseline location.

Note: Paths in this function are relative to where the function is called␣
↪from
      not this file
```

1.3.5 Logging

Logging is done through Axom's SLIC component. SLIC provides a lot of configurable logging functionality which we have consolidated the header `src/infrastructure/logger.hpp` and implemented in `src/infrastructure/logger.cpp`.

Note: On parallel runs, messages can be out of order if flush is not called often enough.

Logging Streams

SLIC has a concept of logging streams. Logging streams controls the following:

- How each message is formatted. More info here .
- Where each messages are output, such as `std::cout`, `std::cerr`, or to a file stream.

- Logic for handling and filtering of messages, based on message level or content.

Serac creates the following logging streams under different conditions:

- GenericOutputStream
  - Serial
  - Debug and info messages to `std::cout`
  - Warning and error messages to `std::cerr`
  - Logs all messages directly to given streams.
- LumberjackStream
  - Parallel
  - Debug and info messages to `std::cout`
  - Warning and error messages to `std::cerr`
  - Flushing causes messages to be scalably passed and filtered down to rank 0 then outputted.
  - On error, SLIC does a local flush on the aborting rank, then does an `MPI_Abort`. Messages on other ranks since the last flush will be lost.

Message Levels

SLIC has 4 message levels to help indicate the important of messages. Descriptions are as follows:

- Debug - messages that help debugging runs, only on when `SERAC_DEBUG` is defined
- Info - basic informational messages
- Warning - message indicating that something has gone wrong but not enough to end the simulation
- Error - message indicating a non-recoverable error has occurred

Logging Macros

SLIC provides many helper macros that assist in logging messages. Here is a list of them but more information can be found in the Axom documentation :

- `SLIC_INFO(msg)` - Logs info message
- `SLIC_INFO_IF(expression, msg)` - Logs info message if expression is true
- `SLIC_INFO_ROOT(msg)` - Logs info message if on rank 0
- `SLIC_INFO_ROOT_IF(expression, msg)` - Logs info message if on rank 0 and expression is true
- `SLIC_WARNING(msg)` - Logs warning message
- `SLIC_WARNING_IF(expression, msg)` - Logs warning message if expression is true
- `SLIC_WARNING_ROOT(msg)` - Logs warning message if on rank 0
- `SLIC_WARNING_ROOT_IF(expression, msg)` - Logs warning message if on rank 0 and expression is true
- `SLIC_ERROR(msg)` - Logs error message
- `SLIC_ERROR_IF(expression, msg)` - Logs error message if expression is true
- `SLIC_ERROR_ROOT(msg)` - Logs error message if on rank 0

- `SLIC_ERROR_ROOT_IF(expression, msg)` - Logs error message if on rank 0 and expression is true

The following macros are compiled out if not in a debug build:

- `SLIC_ASSERT(expression)` - Logs an error if expression is not true
- `SLIC_ASSERT_MSG(expression, msg)` - Logs an error with a custom message if expression is not true
- `SLIC_CHECK(expression)` - Logs an warning if expression is not true
- `SLIC_CHECK_MSG(expression, msg)` - Logs an warning with a custom message if expression is not true
- `SLIC_DEBUG(msg)` - Logs debug message on rank 0
- `SLIC_DEBUG_IF(expression, msg)` - Logs debug message if expression is true
- `SLIC_DEBUG_ROOT(msg)` - Logs debug message if on rank 0
- `SLIC_DEBUG_ROOT_IF(expression, msg)` - Logs debug message if on rank 0 and expression is true

1.3.6 Expression Templates

Expression templates are a C++ technique that leverages static polymorphism (with the Curiously Recurring Template Pattern) to build an expression tree at compile time. The lazy evaluation afforded by this approach minimizes the number of temporary intermediate results and in some circumstances allows for increased optimization due to compile-time knowledge of the expression tree.

Serac provides expression templates for operations on `mfem::Vector` s via operator overloads. This approach allows for a user to add two vectors by simply writing `mfem::Vector c = a + b;` which is more natural and readable than `add(a, b, c)`.

In particular, Serac currently provides the following operations:

1. Vector addition with `a + b`
2. Vector subtraction with `a - b`
3. Vector negation with `-a`
4. Scalar multiplication with `s * a` or `a * s` for scalar `s` and vector `a`
5. Application of an `mfem::Operator` with `op * a` for `mfem::Operator op` and vector `a`

Note: Because `mfem::Matrix` inherits from `mfem::Operator` this functionality includes matrix-vector multiplication.

Note: All of these expressions can be composed, that is, an expression can be used as an argument to another expression. This allows for chaining, e.g., `-a + b + 0.3 * c`.

Memory Safety

Because these expression objects are imperfect closures (as with lambdas in C++), care should be taken to ensure that objects are not used after they go out of scope.

Consider a case where an intermediate expression is assigned to a variable (perhaps for readability):

```
auto lambda = [](const auto& a, const auto& b) {
    auto a3 = a * 3.0;
    return a3 - b;
};
```

This code does not compile as lvalue references to expression objects are prohibited. In this case the intermediate result must be *moved* into the return statement so the returned expression can take ownership:

```
auto lambda = [](const auto& a, const auto& b) {
    auto a3 = a * 3.0; // a is of expression type
    return std::move(a3) - b; // return value is of expression type
};
```

Consider this snippet where an intermediate expression is evaluated:

```
auto lambda = [](const auto& a, const auto& b) {
    auto a3 = evaluate(a * 3.0); // a is of type mfem::Vector
    return a3 - b; // return value is of expression type
};
```

In this case a reference to a function-scope variable `a3` is returned, which is incorrect as `a3` goes out of scope when the function returns.

Note: The above example WILL compile but will result in a runtime crash.

As with the previous example, this should be resolved by moving the `mfem::Vector` into the expression:

```
auto lambda = [](const auto& a, const auto& b) {
    auto a3 = evaluate(a * 3.0); // a is of type mfem::Vector
    return std::move(a3) - b; // return value is of expression type
};
```

Avoiding Allocations

Currently, assigning an expression template to a vector, e.g. `mfem::Vector c = a + b` will result in an allocation. To avoid this, use the `evaluate` function:

```
// Preallocate
mfem::Vector c(size);
evaluate(a + b, c);
```

Distributed Expression Templates

Note: This functionality is under active development and may change significantly.

Distributed expression templates (with `mfem::HypreParVector`) are **experimentally** supported. Mixed operations (where some operands are global `mfem::Vector` s and others are distributed vectors) are not supported. Additionally, an expression cannot be assigned to a `HypreParVector`. Use the `evaluate` function described above.

1.3.7 Profiling Serac using Adiak and Caliper

Introduction to Adiak

Adiak is a library developed at LLNL for collecting metadata that can be used to compare multiple runs across programs. For more information, read Adiak's documentation. Note that Serac provides some wrapper functions to initialize and finalize Adiak metadata collection.

Introduction to Caliper

Caliper is a framework developed at LLNL for measuring the performance of programs. To find out more, read Caliper's documentation. Serac also provides convenient macros that make it easy to instrument and assess the performance of simulation code.

Introduction to SPOT

SPOT is a framework developed at LLNL for vizualizing performance data. SPOT is an external tool and does not need to be linked into Serac.

Build Instructions

To use Adiak and Caliper with Serac, install the `profiling` variant of `serac` with Spack, i.e., `serac+profiling`. Note that these libraries are pre-built as part of the installed set of libraries on LC.

Instrumenting Code

To use the functions and macros described in the remainder of this section, the `serac/infrastructure/profiling.hpp` header must be included.

To enable Adiak and Caliper for a program, call `serac::profiling::initialize()`. This will begin the collection of metadata and performance data. Optionally, an MPI communicator can be passed to configure Adiak and a Caliper ConfigManager configuration string can be passed to configure Caliper. Note that you must still annotate regions to be profiled and provide any custom metadata.

Call `serac::profiling::finalize()` to conclude metadata and performance monitoring and to write the data to a `.cali` file.

To provide custom metadata for comparing program runs, call `SERAC_SET_METADATA(name, data)` after `serac::profiling::initialize()` and before `serac::profiling::finalize`. This will add extra metadata into the `.cali` file. Supported metadata types are integrals, floating points, and strings. Note that this macro is a no-op if the `profiling` variant is not used.

```
SERAC_SET_METADATA("dimensions", 2);
SERAC_SET_METADATA("mesh", "../data/star.mesh");
```

To add profile regions and ensure that Caliper is only used when it has been enabled through Spack, only use the macros described below to instrument your code:

Use `SERAC_MARK_FUNCTION` at the very top of a function to mark it for profiling.

Use `SERAC_MARK_BEGIN(name)` at the beginning of a region and `SERAC_MARK_END(name)` at the end of the region.

Use `SERAC_MARK_LOOP_BEGIN(id, name)` before a loop to mark it for profiling, `SERAC_MARK_LOOP_ITER(id, i)` at the beginning of the `i` th iteration of a loop, and `SERAC_MARK_LOOP_END(id)` immediately after the loop ends:

```
SERAC_MARK_BEGIN("region_name");

SERAC_MARK_LOOP_BEGIN(doubling_loop, "doubling_loop");
for (int i = 0; i < input.size(); i++)
{
  SERAC_MARK_LOOP_ITER(doubling_loop, i);
  output[i] = input[i] * 2;
}
SERAC_MARK_LOOP_END(doubling_loop);

SERAC_MARK_END("region_name");
```

Note that the `id` argument to the `SERAC_MARK_LOOP_*` macros can be any identifier as long as it is consistent between all uses of `SERAC_MARK_LOOP_*` for a given loop.

To reduce the amount of annotation for regions bounded by a particular scope, use `SERAC_PROFILE_SCOPE(name)`. This will follow RAII and works with graceful exception handling. When `SERAC_PROFILE_SCOPE` is instantiated, profiling of this region starts, and when the scope exits, profiling of this region will end.

```
 // Refine once more and utilize SERAC_PROFILE_SCOPE
{
  SERAC_PROFILE_SCOPE("RefineOnceMore");
  pmesh->UniformRefinement();
}
```

Alternatively, for single line expressions, use `SERAC_PROFILE_EXPR(name, expr)`. In the following example, only the call to `buildMeshFromFile` will be profiled (tag = "LOAD_MESH").

```
auto pmesh = mesh::refineAndDistribute(SERAC_PROFILE_EXPR("LOAD_MESH",␣
↪buildMeshFromFile(mesh_file)), 0, 0);
```

Note: `SERAC_PROFILE_EXPR` creates a lambda and the expression is evaluated within that scope, and then the result is returned.

Performance Data

The metadata and performance data are output to a `.cali` file. To analyze the contents of this file, use cali-query.

To view this data with SPOT, open a browser, navigate to the SPOT server (e.g. LC), and open the directory containing one or more `.cali` files. For more information, watch this recorded tutorial.

1.3.8 Memory Checking

There are two commonly available memory checkers available to use with Serac on LC: AddressSanitizer and Valgrind.

AddressSanitizer

AddressSanitizer (aka Asan) is memory error detection tool that is a part of LLVM. It very fast and easy to use but doesn't seem as robust as Valgrind. It requires compile and link flags which are enabled via the CMake option ENABLE_ASAN. Anything in our CMake system will get those flags after that is enabled but our third-party libraries (like MFEM) will not. After that just run your built executable and Asan will output a log to the screen after your program is done running. Asan's behavior can be modified with a set of environment variables .

Note: Asan only works with the Clang and GCC compiler chains. Our build system will throw and error if you try to build with anything else while ENABLE_ASAN is ON.

Here is a recommended workflow:

```
./config-build.py -hc host-configs/rzgenie-toss_3_x86_64_ib-gcc@8.1.0.cmake -DENABLE_
↪ASAN=ON
cd build-rzgenie-toss_3_x86_64_ib-gcc@8.1.0-debug
srun -N1 --exclusive --mpi-bind=off make -j
LSAN_OPTIONS=suppressions=../suppressions.asan ASAN_OPTIONS=log_path=asan.out:log_exe_
↪name=true srun -n2 bin/serac
```

This will output files in the current directory for each process that follow the pattern: `asan.out.<exe name>.<pid>`. It also sets your return code to a non-zero value if there were any non-suppressed memory errors.

LSAN_OPTIONS and ASAN_OPTIONS are delimited by ':'.

Here is an explanation of the given options (all should be added to ASAN_OPTIONS unless noted):

- `suppressions`: Location of memory leak suppression file (LSAN_OPTIONS)
- `log_path`: Logs to the given file instead of to the screen. This is very helpful to avoid intermingled lines on the screen from every process
- `log_exe_name`: Adds executable name to log_path

Helpful options:

- `fast_unwind_on_malloc=0`: This improves Asan's stack tracing ability but also greatly slows down the run
- `exitcode=0`: This stops Asan from returning a a non-zero exit code from your executable (defaults to 23) (LSAN_OPTIONS)

Debugging with Address Sanitizer enabled

If a program results in address sanitizer emitting an error (gcc example, here):

```
==45800==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x61a00000a1c0 at␣
↪pc 0x5561996372b7 bp 0x7fff89f707e0 sp 0x7fff89f707d0
READ of size 8 at 0x61a00000a1c0 thread T0
    #0 0x5561996372b6 in serac::Functional<double (serac::H1<2, 1>),␣
↪(serac::ExecutionSpace)0>::Gradient::operator mfem::Vector() (/home/sam/code/serac/
↪build/tests/functional_qoi+0x57e2b6)
    #1 0x55619962df8c in void check_gradient<double (serac::H1<2, 1>)>
↪(serac::Functional<double (serac::H1<2, 1>), (serac::ExecutionSpace)0>&,␣
↪mfem::Vector&) (/home/sam/code/serac/build/tests/functional_qoi+0x574f8c)
    #2 0x556199624144 in void functional_qoi_test<2, 2>(mfem::ParMesh&, serac::H1<2,␣
↪1>, serac::Dimension<2>) (/home/sam/code/serac/build/tests/functional_qoi+0x56b144)
```

(continues on next page)

(continued from previous page)

```
    #3 0x556199614ee3 in main /home/sam/code/serac/src/serac/physics/utilities/
↪functional/tests/functional_qoi.cpp:206
```

the information provided (e.g. invalid read at /home/sam/code/serac/build/tests/functional_qoi+0x57e2b6) doesn't precisely reveal where the error takes place. We can set a breakpoint on the address sanitizer error function to find out exactly where the problem lies, but first we need to find out the name of that error reporting symbol. One way to do this is to filter the symbols in the executable:

```
sam@provolone:~/code/serac/build/tests$ nm -g ./functional_qoi | grep asan
                 U __asan_after_dynamic_init
                 U __asan_before_dynamic_init
                 U __asan_handle_no_return
                 U __asan_init
0000000001646450 B __asan_option_detect_stack_use_after_return
                 U __asan_poison_stack_memory
                 U __asan_register_globals
                 U __asan_report_load1
                 U __asan_report_load16
                 U __asan_report_load2
                 U __asan_report_load4
                 U __asan_report_load8
                 U __asan_report_load_n
                 U __asan_report_store1
                 U __asan_report_store16
                 U __asan_report_store4
                 U __asan_report_store8
                 U __asan_report_store_n
                 U __asan_stack_free_5
                 U __asan_stack_free_6
                 U __asan_stack_free_7
                 U __asan_stack_free_8
                 U __asan_stack_free_9
                 U __asan_stack_malloc_0
                 U __asan_stack_malloc_1
                 U __asan_stack_malloc_2
                 U __asan_stack_malloc_3
                 U __asan_stack_malloc_4
                 U __asan_stack_malloc_5
                 U __asan_stack_malloc_6
                 U __asan_stack_malloc_7
                 U __asan_stack_malloc_8
                 U __asan_stack_malloc_9
                 U __asan_unpoison_stack_memory
                 U __asan_unregister_globals
                 U __asan_version_mismatch_check_v8
00000000016466a1 B __odr_asan.mesh2D
00000000016466a0 B __odr_asan.mesh3D
00000000016466a3 B __odr_asan.myid
00000000016466a2 B __odr_asan.nsamples
00000000016466a4 B __odr_asan.num_procs
```

and select the one that corresponds to the the original error (read of size 8 => `__asan_report_load8`).

Valgrind

Valgrind is a very powerful set of tools that help with dynamic analysis tools. We will focus on memcheck which is a memory error detection tool.

Unlike Asan, valgrind does not need any special compiler flags. Just build your executable and run your executable with `valgrind`. Valgrind's suppression files are easily generated by valgrind with `--gen-suppressions=all` and are more customizable than Asan's.

Here is a recommended workflow:

```
./config-build.py -hc host-configs/rzgenie-toss_3_x86_64_ib-gcc@8.1.0.cmake
cd build-rzgenie-toss_3_x86_64_ib-gcc@8.1.0-debug
srun -N1 --exclusive --mpi-bind=off make -j
srun -n2 valgrind --tool=memcheck --log-file=valgrind.out --leak-check=yes --show-
↪leak-kinds=all --num-callers=20 --suppressions=../suppressions.valgrind bin/serac
```

This will will produce a file called `valgrind.out` in the current directory with a valgrind report.

Here is an explanation of the given options:

- `--tool=memcheck`: valgrind is a tool-suite so this runs the memcheck tool
- `--log-file=valgrind.out`: Logs report to the given file
- `--leak-check=yes`: Enables memory leak checks
- `--show-leak-kinds=all``: Enables showing all memory leak kinds
- `--num-callers=20`: Limits the size of the stack traces to 20
- `--suppressions=../suppressions.valgrind`: Location of memory leak suppression file

1.3.9 Building a Docker Image

The following instructions apply to the creation of a new compiler image.

Create New Docker File

Note: If a Dockerfile for the desired compiler already exists, you can skip this section and go to *update-docker-image-label* .

---

1. Start by cloning the `serac` repository and creating a branch off `develop`.
2. Ensure that an Axom image exists on Dockerhub for the desired compiler. If no corresponding Axom compiler image exists, it should be created before proceeding. Here is the Axom Docker repository.

   Note: A new docker repository has been created and is being actively maintained for RADIUSS unlike the Axom Docker repo. We should migrate to it but it may be missing system packages.

   ---

3. Go to the `scripts/docker` directory and run `build_new_dockerfile.sh`, passing the compiler name and version, e.g. for Clang 10, run `./build_new_dockerfile.sh clang 10`. Minor versions can also be specified, for example, GCC 9.3 can be specified with `./build_new_dockerfile.sh gcc 9.3`. This will create a Dockerfile whose name corresponds to a specific compiler, e.g., `dockerfile_clang-10`. This may require modifications depending on the compiler and base image - for example, an extra system package might be installed so Spack doesn't need to build it from source.

4. Edit `./github/workflows/docker_build_tpls.yml` to add new job for the new compiler image. This can be copy-pasted from one of the existing jobs - the only things that must be changed are the job name and `TAG`, which should match the name of the compiler/generated `Dockerfile`. For example, a build for `dockerfile_clang-10` must set `TAG` to `clang-10`. For clarity, the `name` field for the job should also be updated.
5. Commit and push the added YAML file and new Dockerfile.

Update/Add Docker Image

1. Go to the Actions tab on GitHub, select the "Docker TPL Build" action, and run the workflow on the branch to which the above changes were pushed.
2. Once the "Docker TPL Build" action completes, it will produce artifacts for each of the generated host-configs. Download these artifacts and rename them to just the compiler spec. For example, `buildkitsandbox-linux-clang@10.0.0.cmake` to `clang@10.0.0.cmake` and commit them to your branch under `host-configs/docker`. You will also have to update `azure-pipelines.yml` if you added or change the existing compiler specs. These are all in variables called `HOST_CONFIG`.
3. Copy the new docker image names from each job under the `Get dockerhub repo name` step. For example, `seracllnl/tpls:clang-10_06-02-22_04h-11m`. This will replace the previous image name at the top of `azure-pipelines.yml` under the `matrix` section or add a new entry if you are adding a new docker image.
4. To include the new image in CI jobs, add/update the `matrix` entry to `azure-pipelines.yml`, modifying its attributes with the appropriate new image name (which is timestamped) and new host-config file.

1.3.10 Tensor Class

`tensor` is a class template for doing arithmetic on small, statically-sized vectors, matrices, and tensors. To create one, specify the underlying type in the first template argument, followed by a list of integers for the shape. For example, `tensor<float,3,2,4>` is conceptually similar to the type `float[3][2][4]`.

Here are some examples and features:

- `tensor` has value semantics (in contrast to C++ multidimensional arrays):

```
// c-style arrays are okay for storage, but can't do much else
{
  double u[3] = {1.0, 2.0, 3.0};
  double v[3] = u; // does not compile
  double * w  = u; // does compile, but size information is lost and w is a␣
↪shallow copy
}

// tensors store their own data and can be copied, assigned, referenced, and␣
↪transformed
{
  tensor < double, 3 > u = {1.0, 2.0, 3.0};
  tensor < double, 3 > v = u; // make a copy of u
  tensor < double, 3 > & w = u; // make a reference to u
}
```

- `tensor` supports operator overloading:

```
tensor < double, 3 > u = {1.0, 2.0, 3.0};
tensor < double, 3 > v = {4.0, 5.0, 6.0};
tensor < double, 3 > sum = u + v;
tensor < double, 3 > weighted_sum = 3.0 * u - v / 7.0;
weighted_sum += sum;
```

- `tensor` supports many common mathematical functions:

```
tensor < double, 3, 3 > I = Identity<3>();
tensor < double, 3, 3 > grad_u = {...};
tensor < double, 3, 3 > strain = 0.5 * (grad_u + transpose(grad_u)); // or,␣
↪equivalently, sym(grad_u)
tensor < double, 3, 3 > stress = lambda * tr(strain) * I + 2.0 * mu * strain;

tensor < double, 3 > traction = dot(stress, normal);
tensor < double, 2, 3 > J = {...};
double dA = sqrt(det(dot(J, transpose(J))));
tensor < double, 3 > force = traction * dA;
```

- `tensor` supports useful shortcuts (like class template argument deduction):

```
// class template argument deduction: A is deduced to have type tensor<double,2,2>
tensor A = {{{1.0, 2.0}, {2.0, 3.0}}};

// create tensors from index notation expressions, B has type tensor<double,3,3,3,
↪3>
tensor B = make_tensor<3,3,3,3>([](int i, int j, int k, int l){
  return 0.5 * ((i == j) * (k == l) + (i == l) * (k == j));
});

// slicing: get and set specific subtensors
tensor< double, 3, 3 > C = B[0][2];
B[2][0][0] = C[1];

// access by operator() or operator[]
C[1][1] = 2.0;
C(2, 2) = 3.0;
```

- `tensor` arithmetic supports `constexpr` evaluation:

```
constexpr tensor change_of_basis_matrix = {{
  { 3.0, -2.0, 1.7},
  { 2.0,  8.0, 1.7},
  {-1.0,  4.0, 6.7}
}};

// express a quantity in a new basis
tensor v = dot(change_of_basis_matrix, u);

// modify the components in the new basis
v = f(v);

// precompute the inverse basis transformation at compile time
constexpr tensor inverse_change_of_basis_matrix = inv(change_of_basis_
↪matrix);

// convert the modified values back to the original basis
u = dot(inverse_change_of_basis_matrix, v);
```

- `tensor` only allows operations between operands of appropriate shapes

```
tensor< double, 3, 2 > A{};
tensor< double, 3 > u{};
tensor< double, 2 > v{};

auto uA = dot(u, A); // works, returns tensor< double, 2 >
auto Av = dot(A, v); // works, returns tensor< double, 3 >
auto Au = dot(A, u); // compile error: incompatible dimensions for dot product
auto vA = dot(v, A); // compile error: incompatible dimensions for dot product

auto w = u + v; // compile error: can't add tensors of different shapes

A[0] = v; // works, assign a new value to the first row of A
A[1] = u; // compile error: can't assign a vector with 3 components to a vector␣
↪of 2 components
```

Dual Number Class

`dual` is a class template that behaves like a floating point value, but also stores information about derivatives. For example, say we have a function, $f(x) = \frac{x \sin(\exp(x)-2)}{1+x^2}$. In C++, one might implement this function as:

```
auto f = [](auto x){ return (x * sin(exp(x) - 2.0)) / (1 + x*x); };
```

If $f(x)$ is used in a larger optimization or root-finding problem, we will likely also need to be able to evaluate $f\,'(x)$. Historically, the two most common ways to get this derivative information were

1. Finite Difference Stencil:

   ```
   static constexpr double epsilon = 1.0e-9;
   auto dfdx = [](double x) { return (f(x + epsilon) - f(x - epsilon)) / (2.0 *␣
   ↪epsilon); }
   ```

   This approach is simple, but requires multiple function invocations and the accuracy suffers due to catastrophic cancellation in floating point arithmetic.

2. Derive the expression for $f\,'(x)$, either by hand or with a computer algebra system, and manually implement the result. For example, using Mathematica we get

   $$f\,'(x) = \frac{\exp(x)(x+x^3)\cos(2-\exp(x)) - (x^2-1)\sin(2-\exp(x))}{(1+x^2)^2},$$

   which must then be manually implemented in C++ code:

   ```
   auto dfdx = [](double x) {
     return (exp(x) * (x + x*x*x) * cos(2 - exp(x)) - (x*x - 1) * exp(2 - sin(x))) /␣
   ↪((1 + x*x) * (1 + x*x));
   };
   ```

   This approach can give very accurate results, and allows the derivative implementations to be individually optimized for performance. The downside is that the symbolic differentiation and manual implementation steps can be error prone: mistakes in transcription, differentiation, or implementation can be hard to notice.

   To emphasize this point, the expression for $f\,'(x)$ given above is actually incorrect, and the subsequent C++ implementation of that incorrect expression for $f\,'(x)$ is itself incorrect. But if you only skimmed the content above, you likely didn't notice.

The `dual` class template provides a 3rd option that improves on the accuracy and performance of finite difference stencil, without sacrificing accuracy. In addition, it doesn't require the developer to manually differentiate and write new code that might contain errors. An example:

```
double answer = f(x); // evaluate f at x
dual< double > answer_and_derivative = f(make_dual(x)); // evaluate f and f' at x
double just_the_answer = answer.value;
double just_the_gradient = answer.gradient;
```

Internally, the implementation is remarkably simple:

```
template <typename gradient_type>
struct dual {
  double        value;
  gradient_type gradient;
};
```

That is, `dual` just stores a `double` value and a specified type for the gradient term. Then, the basic rules of differentiation are encoded in the corresponding operator overloads:

$$\frac{d}{dx}(a + b) = \frac{da}{dx} + \frac{db}{dx}$$

```
template <typename gradient_type_a, typename gradient_type_b>
constexpr auto operator+(dual<gradient_type_a> a, dual<gradient_type_b> b)
{
  return dual{a.value + b.value, a.gradient + b.gradient};
}
```

$$\frac{d}{dx}(a\ b) = \frac{da}{dx}\ b + a\frac{db}{dx}$$

```
template <typename gradient_type_a, typename gradient_type_b>
constexpr auto operator*(dual<gradient_type_a> a, dual<gradient_type_b> b)
{
  return dual{a.value * b.value, a.gradient * b.value + a.value * b.gradient};
}
```

and so on. In this way, when a dual number is passed in to a function, each of the intermediate values keep track of gradient information as well. The downside to this approach is that doing that arithmetic to track the gradients of intermediate values is more expensive than manually writing code for the derivatives.

However, by supporting both manually-written derivatives and `dual` numbers, users can choose to calculate derivatives in whatever manner is appropriate for their problem: manually-written gradients for performance-critical code-paths, and automatic differentiation for iterating quickly on prototypes and research.

Some additional resources on the theory and implementation of automatic differentiation are given below:

Slides on AD Theory

Article demonstrating how AD applies to a computational graph

C++ tools and libraries for AD

Using `tensor` and `dual` together

In the previous example, f was a function with a scalar input and scalar output. In practice, most of the functions we care about are more interesting. For example, an isotropic linear elastic material in solid mechanics has the following

stress-strain relationship:

$$\sigma = \lambda \, \mathrm{tr}(\epsilon) \, \mathbf{I} + 2 \, \mu \, \epsilon$$

or, in C++:

```
double lambda = 2.0;
double mu = 1.0;
static constexpr auto I = Identity<3>();
auto stress = [=](auto strain){ return lambda * tr(strain) * I + 2 * mu * strain; };
```

That is, `stress()` takes a `tensor<double,3,3>` as input, and outputs a `tensor<double, 3, 3>`:

```
tensor< double, 3, 3 > epsilon = {...};
tensor< double, 3, 3 > sigma = stress(epsilon);
```

In general, each part of a function's output can depend on each part of its inputs. So, in this example the gradient could potentially have up to 81 components:

$$\frac{\partial \sigma_{ij}}{\partial \epsilon_{kl}}, \qquad i,j,k,l \in 1,2,3$$

If we promote the input argument to a tensor of dual numbers, we can compute these derivatives automatically:

```
tensor< double, 3, 3 > epsilon = {...};
tensor< dual< tensor< double, 3, 3 > >, 3, 3 > sigma = stress(make_dual(epsilon));
```

Now, `sigma` contains value and gradient information that can be understood in the following way:

$$\texttt{sigma[i][j].value} = \sigma_{ij} \qquad \texttt{sigma[i][j].gradient[k][l]} = \frac{\partial \sigma_{ij}}{\partial \epsilon_{kl}}$$

There are also convenience routines to extract all the values and gradient terms into their own tensors of the appropriate shape:

```
// as before
tensor< dual< tensor< double, 3, 3 > >, 3, 3 > sigma = stress(make_dual(epsilon));

// extract the values
tensor< double, 3, 3 > sigma_values = get_value(sigma);

// extract the gradient
tensor< double, 3, 3, 3, 3 > sigma_gradients = get_gradient(sigma);
```

Differentiating Functions with Multiple Inputs and Outputs

Now let's consider a function that has multiple inputs and multiple outputs:

```
double mu = 1.0;
double rho = 2.0;
static constexpr auto I = Identity<3>();
auto f = [=](auto p, auto v, auto L){
   auto strain_rate = 0.5 * (L + transpose(L));
   auto stress = - p * I + 2 * mu * strain_rate;
   auto kinetic_energy_density = 0.5 * rho * dot(v, v);
   return serac::tuple{stress, kinetic_energy_density};
};
```

Here, `f` calculates the stress, σ, and local kinetic energy density, q, of a fluid in terms of the pressure `p` (scalar), velocity `v` (3-vector), and velocity gradient `L` (3x3 matrix). So, there are 2 outputs and 3 inputs, resulting in potentially 6 derivatives with different order tensors:

$$\frac{\partial \sigma}{\partial p}, \frac{\partial \sigma}{\partial v}, \frac{\partial \sigma}{\partial L}, \frac{\partial q}{\partial p}, \frac{\partial q}{\partial v}, \frac{\partial q}{\partial L}$$

All of these derivatives can be calculated in a single function invocation by following the same pattern as before:

```
double p = ...;
tensor<double,3> v = ...;
tensor<double,3,3> L = ...;

// promote the arguments to dual numbers with make_dual()
tuple dual_args = make_dual(serac::tuple{p, v, L});

// then call the function with the dual arguments
auto outputs = apply(f, dual_args);

// note: serac::apply is a way to pass an n-tuple to a function that expects n␣
↪arguments
//
// i.e. the two following lines have the same effect
// f(p, v, L);
// serac::apply(f, serac::tuple{p, v, L});
```

Like before, `outputs` will now contain the actual output values, but also all gradient terms (6, in this case). To get the gradient tensors, we call the same `get_gradient()` function:

```
auto gradients = get_gradient(outputs);
```

The 6 gradient terms for this example can be thought of in a "matrix" where the i, j entry is the derivative of the i^{th} output with respect to the j^{th} input:

$$\left[\frac{\partial f_i}{\partial x_j}\right] = \begin{bmatrix} \frac{\partial \sigma}{\partial p} & \frac{\partial \sigma}{\partial v} & \frac{\partial \sigma}{\partial L} \\ \frac{\partial q}{\partial p} & \frac{\partial q}{\partial v} & \frac{\partial q}{\partial L} \end{bmatrix}$$

The type returned by `get_gradient()` reflects this structure: returning a `serac::tuple` of `serac::tuple`. So for this example, the return type will be of the form:

```
serac::tuple<
  serac::tuple< df1_dx1_type, df1_dx2_type, df1_dx2_type >,
  serac::tuple< df2_dx1_type, df2_dx2_type, df2_dx2_type >
>;
```

The individual blocks can be accessed by using `serac::get()`.

Finally, if we look at the actual types contained in `get_gradient(output)` we see a few interesting details:

```
serac::tuple<
  serac::tuple<tensor<double, 3, 3>, zero,              tensor<double, 3, 3, 3, 3> >,
  serac::tuple<zero,                 tensor<double, 3>, zero                       >
> gradients = get_gradient(outputs);
```

First, the tensor shapes of the individual blocks are are in agreement with what we expect (e.g. $\frac{\partial \sigma}{\partial p}$ is 3x3, $\frac{\partial \sigma}{\partial L}$ is 3x3x3x3, etc).

Second, some of the derivative blocks seem to be missing! Instead of actual tensors, a mysterious type `zero` appears in three of the blocks of our derivative. What does that mean?

It means that if we look back at the original definition of our function, we see that the stress tensor does not depend on `v` at all. Similarly, the kinetic energy density only depends on `v`, while having no dependence on `p` or `L`. The implementation of the `tensor` and `dual` class templates automatically detects and optimizes away unnecessary storage and calculations associated with these derivative blocks that are identically zero.

1.3.11 Functional

`Functional` is a class that is used specify and evaluate finite-element-type calculations. For example, the weighted residual for a solid mechanics simulation may look something like:

$$r(u) := \underbrace{\int_\Omega \sigma(\nabla u) \cdot \nabla \psi \, \mathrm{d}v}_{\text{stress response}} + \underbrace{\int_\Omega b(\mathbf{x}) \, \psi \, \mathrm{d}v}_{\text{body forces}} + \underbrace{\int_{\partial\Omega} \mathbf{t}(\mathbf{x}) \, \psi \, \mathrm{d}a}_{\text{surface loads}},$$

where ψ are the test basis functions. To describe this residual using `Functional`, we first create the object itself, providing a template parameter that expresses the test and trial spaces (i.e. the respective "outputs" and "inputs" of the residual function, r). In this case, solid mechanics uses nodal displacements and residuals (i.e. H1 test and trial spaces), so we write:

```
constexpr int order = 2; // the polynomial order of the basis functions
constexpr int dim = 3; // the number of components per node
using test = H1<order, dim>;
using trial = H1<order, dim>;
Functional< test(trial) > residual(&test_fes, {&trial_fes});
```

where `test_fes`, `trial_fes` are the `mfem::FiniteElementSpaces` for the problem. The template argument follows the same convention of `std::function`: the output-type appears outside the parentheses, and the input-type(s) appear, inside the parentheses (in order). So, the last line of code in the snippet above is saying that `residual` is going to represent a calculation that takes in an H1 field (displacements), and returns weighted residual vectors for each node, using H1 test functions.

Now that the `Functional` object is created, we can use the following functions to define integral terms (depending on their dimensionality). Here, we use s to denote the "source" term (integrated against test functions), and f to denote the "flux" term (integrated against test function gradients).

1. Integrals of the form: $\iint_\Omega \psi \cdot s + \nabla\psi : f \; da$

```
residual.AddAreaIntegral(
  DependsOn< ... >{},
  [](auto x, auto ... args){
    auto s = ...;
    auto f = ...;
    return serac::tuple{s, f};
  },
  domain_of_integration
);
```

2. Integrals of the form: $\iiint_\Omega \psi \cdot s + \nabla\psi : f \; dv$

```
residual.AddVolumeIntegral(
  DependsOn< ... >{},
  [](auto x, auto ... args){
    auto s = ...;
    auto f = ...;
    return serac::tuple{s, f};
```

(continues on next page)

(continued from previous page)

```
  },
  domain_of_integration
);
```

3. Integrals of the form: $\iint_{\partial\Omega} \psi \cdot s \; da$

```
residual.AddSurfaceIntegral(
  DependsOn< ... >{},
  [](auto ... args){
    auto s = ...;
    return s;
  },
  domain_of_integration
);
```

Note: the first argument `DependsOn< ... >{}` is a way to specify which of the trial spaces (if any) are required by that integral. e.g. `DependsOn< 1, 2 >{}` will indicate that the values from trial spaces 1 and 2 (zero-based indexing) will be passed in to the provided q-function.

Going back to our example problem (since we assumed 3D earlier) we can make an `Add****Integral()` call for each of the integral terms in the original residual. In each of these functions, the first argument tells which trial spaces the calculation depends on, the second argument is the integrand (a lambda function or functor returning $\{s, f\}$), and the third argument is the domain of integration. Let's start with the stress response term:

```
// The integrand lambda function is passed the spatial position of the quadrature␣
↪point,
// as well as a {value, derivative} tuple for the trial space.
residual.AddVolumeIntegral(

  // this calculation depends on the displacement field, which is the 0th trial space
  DependsOn<0>{},

  [](auto x, auto disp){

    // Here, we unpack the {value, derivative} tuple into separate variables
    auto [u, grad_u] = disp;

    // call some constitutive model for the material in this domain
    auto stress = material_model(grad_u);

    // Functional::AddVolumeIntegral() expects us to return a tuple of the form {s, f}
↪,
    // but this integral has no term that get integrated against the test functions,
    // so the "source" term is just zero
    return serac::tuple{zero{}, stress};

  },
  mesh
);
```

The other terms follow a similar pattern. For the body force:

```
residual.AddVolumeIntegral(

  // this calculation doesn't require values from any trial space
  // so there is nothing between the angle brackets
```

(continues on next page)

(continued from previous page)

```
  DependsOn</* nothing in here */>{},

  [](auto x){

    // evaluate the body force function at the location of the quadrature point
    auto body_force = b(x);

    // Functional::AddVolumeIntegral() expects us to return a tuple of the form {s, f}
↪,
    // but this integral has no term that get integrated against the test function␣
↪gradients,
    // so the "flux" term is just zero
    return std::tuple{body_force, zero{}};

  },
  mesh
);
```

And finally, for the surface tractions:

```
// Functional::AddSurfaceIntegral() only expects us to return s, so we don't need a␣
↪tuple
residual.AddSurfaceIntegral(

  // this calculation doesn't require values from any trial space
  // so there is nothing between the angle brackets
  DependsOn</* nothing in here */>{},

  // evaluate the traction at the location of the quadrature point
  // note: the q-function for boundary integrals is also passed
  // the unit surface normal as the second argument
  [](auto x, auto n){ return t(x); },

  surface_mesh
);
```

Now that we've finished describing all the integral terms that appear in our residual, we can carry out the actual calculation by calling `Functional::operator()`:

```
auto r = residual(displacements);
```

Putting these snippets together without the verbose comments, we have (note: the two AddVolumeIntegrals were fused into one):

```
using test = H1<order, dim>;
using trial = H1<order, dim>;
Functional< test(trial) > residual(test_fes, trial_fes);

// note: the first two AddVolumeIntegral calls can be fused
// into one, provided they share the same domain of integration
residual.AddVolumeIntegral(
  DependsOn<0>{}, // depends on the displacement field
  [](auto x, auto disp){
    auto [u, grad_u] = disp;
    return serac::tuple{b(x), material_model(grad_u)};
  },
```

(continues on next page)

(continued from previous page)

```
  mesh
);

residual.AddSurfaceIntegral([](auto x, auto disp /* unused */){ return traction(x); },
↪ surface_mesh);

auto r = residual(displacements);
```

So, in only a few lines of code, we can create optimized, custom finite element kernels!

Quantities of Interest

`Functional` can also be used to represent scalar-valued integral expressions. These can be used to represent objective functions, constraints, or other "quantities of interest". To make a `Functional` with a scalar-valued output, use `double` as the test space in its function signature:

```
using trial = H1<order, dim>;

// this indicates that the calculation will
// return a scalar, rather than a residual vector
using test = double;

Functional< test(trial) > qoi(&test_fes, {&trial_fes});

...
```

Like before, the actual integral calculations are defined by calling the following member functions:

1. Integrals of the form: $\iint_\Omega s \, da$

```
qoi.AddAreaIntegral(
  DependsOn< ... >{},
  [](auto x, auto ... args){
    auto s = ...;
    return s;
  },
  domain_of_integration
);
```

2. Integrals of the form: $\iiint_\Omega s \, dv$

```
qoi.AddVolumeIntegral(
  DependsOn< ... >{},
  [](auto x, auto ... args){
    auto s = ...;
    return s;
  },
  domain_of_integration
);
```

3. Integrals of the form: $\iint_{\partial\Omega} s \, da$

```
qoi.AddSurfaceIntegral(
  DependsOn< ... >{},
  [](auto ... args){
    auto s = ...;
    return s;
  },
  domain_of_integration
);
```

Note: since there aren't really test functions in this case (or equivalently, $\phi(x) = 1$), there is never a "flux" term, so these q-functions all just return a scalar. Here's an example of how to use `Functional` to implement a strain-energy calculation to accompany our solid mechanics example:

Strain energy: $U(u) = \frac{1}{2}\iiint_{\Omega} \sigma : \epsilon \, dv$

```
using displacement_field = H1<order,dim>

Functional< double(displacement_field) > strain_energy(&test_fes, {&trial_fes});
strain_energy.AddVolumeIntegral(
  DependsOn<0>{}, // depends on displacement
  [](auto x, auto displacement){
    auto [u, dudx] = displacement;
    auto epsilon = 0.5 * (transpose(dudx) + dudx);
    auto sigma = my_material_model(epsilon);
    auto strain_energy_density = 0.5 * double_dot(sigma, epsilon);
    return strain_energy_density;
  },
  mesh
);
```

Implementation

For the most part, the `Functional` class is just a container of `Integral` objects, and some prolongation and restriction operators to get the data they need:

```
template <typename test, typename trial>
struct Functional<test(trial)> : public mfem::Operator {
  ...
  std::vector< Integral<test(trial)> > domain_integrals;
  std::vector< Integral<test(trial)> > boundary_integrals;
};
```

The calls to `Functional::Add****Integral` forward the integrand and mesh information to an `Integral` constructor and add it to the appropriate list (either `domain_integrals` or `boundary_integrals`). MFEM treats domain and boundary integrals differently, so we maintain them in separate lists.

From there, the `Integral` constructor uses the integrand functor to specialize a highly templated finite element kernel (simplified implementation given below).

```
template < ::Geometry g, typename test, typename trial, int geometry_dim, int spatial_
↪dim, int Q,
           typename derivatives_type, typename lambda>
void evaluation_kernel(const mfem::Vector& U, mfem::Vector& R, derivatives_type*␣
↪derivatives_ptr,
                       const mfem::Vector& J_, const mfem::Vector& X_, int num_
↪elements, lambda qf)
```

(continues on next page)

(continued from previous page)

```
{
  ...

  // for each element in the domain
  for (int e = 0; e < num_elements; e++) {

    // get the values for this particular element
    tensor u_elem = detail::Load<trial_element>(u, e);

    // this is where we will accumulate the element residual tensor
    element_residual_type r_elem{};

    // for each quadrature point in the element
    for (int q = 0; q < static_cast<int>(rule.size()); q++) {
      // get the position of this quadrature point in the parent and physical space,
      // and calculate the measure of that point in physical space.
      auto   xi  = rule.points[q];
      auto   dxi = rule.weights[q];
      auto   x_q = make_tensor<spatial_dim>([&](int i) { return X(q, i, e); });
      auto   J_q = make_tensor<spatial_dim, geometry_dim>([&](int i, int j) { return␣
↪J(q, i, j, e); });
      double dx  = detail::Measure(J_q) * dxi;

      // evaluate the value/derivatives needed for the q-function at this quadrature␣
↪point
      auto arg = detail::Preprocess<trial_element>(u_elem, xi, J_q);

      // evaluate the user-specified constitutive model
      //
      // note: make_dual(arg) promotes those arguments to dual number types
      // so that qf_output will contain values and derivatives
      auto qf_output = qf(x_q, make_dual(arg));

      // integrate qf_output against test space shape functions / gradients
      // to get element residual contributions
      r_elem += detail::Postprocess<test_element>(get_value(qf_output), xi, J_q) * dx;

    }

    // once we've finished the element integration loop, write our element residuals
    // out to memory, to be later assembled into global residuals by mfem
    detail::Add(r, r_elem, e);
  }
}
```

Then, the call to that specialized finite element kernel is wrapped inside a `std::function` object with the appropriate signature. This `std::function` is used to implement the action of `Mult()`:

```
template < typename spaces >
struct Integral {

  ...

  template <int geometry_dim, int spatial_dim, typename lambda_type>
  Integral(...) {
```

(continues on next page)

(continued from previous page)

```
    ...

    evaluation = [=](const mfem::Vector& U, mfem::Vector& R) {
      evaluation_kernel<geometry, test_space, trial_space, geometry_dim, spatial_dim,␣
↪Q>(...);
    };

    ...

  };

  void Mult(const mfem::Vector& input, mfem::Vector& output) const { evaluation(input,
↪ output); }

  std::function<void(const mfem::Vector&, mfem::Vector&)> evaluation;

}
```

Finally, when the user calls `Functional::operator()`, it loops over the domain and surface integrals, calling `Integral::Mult()` on each one to compute the weighted residual contribution from each term.

1.3.12 Co-Developing with Axom and MFEM

Occasionally, Serac developers may want to patch fixes and useful features back to MFEM and Axom. A co-development build environment is available to expedite this process via git submodules.

To enable a co-develop workflow, use the following steps from the root serac directory:

1. **Initialize submodules.** Axom and MFEM are submodules of the Serac repository. The following command recursively updates all submodules:

```
$ git submodule update --init --recursive
```

2. **Generate Build System via CMake.** Configure our CMake build with the co-develop option using a local host-config (CMake initial cache) generated by Spack. You can do this one of two ways:

```
# Option 1: This uses CMake to configure Axom, MFEM, and Serac in one step.
$ ./config-build.py -hc <host config file> -DSERAC_ENABLE_CODEVELOP=ON
$ cd <build directory> # this is autogenerated by script based on host-config name

# Option 2:
$ mkdir <build directory>
$ cd <build directory>
$ cmake -C <host config file> -DSERAC_ENABLE_CODEVELOP=ON ..
```

3. **Build/Test code.** Build MFEM, Axom, and Serac:

```
$ make -j
$ make -j test
```

1.3.13 StateManager

StateManager is a global interface for (as the name suggests) managing state - specifically, the following core components of a simulation:

1. Meshes - instances of `mfem::ParMesh`
2. Fields - instances of `serac::FiniteElementState`
3. Duals - instances of `serac::FiniteElementDual`
4. Material state - instances of `serac::QuadratureData<T>`

`StateManager` acts as both a factory for creating the latter three kinds of state and a means of restarting a simulation, where state is saved to a file and later reloaded to an identical point in the simulation. Critically, this abstraction means that implementers of physics modules do not have to worry about restart vs. non-restart logic, as these factory methods can be used identically in either case.

Under the hood, `StateManager` is implemented in terms of Axom's `axom::sidre::MFEMSidreDataCollection`, whose user documentation is available here.

`MFEMSidreDataCollection` is an implementation of MFEM's DataCollection interface, which allows for instances of `mfem::GridFunction` and `mfem::QuadratureFunction` to be associated with a single instance of `mfem::Mesh`. Because Serac supports multiple meshes within a given simulation, `StateManager` acts as an abstraction over multiple `MFEMSidreDataCollection` instances.

Nominal (non-restart) Workflow

The first interaction a user must always make with `StateManager` is a call to its static `initialize()` method. `StateManager` is implemented as a global singleton so that its contained data can be accessed from physics modules anywhere in the simulation. The singleton is initialized with a non-owning reference to an `axom::sidre::DataStore` and an output directory to which data will be saved.

Note: `StateManager` does not own its `DataStore` because Serac uses a single datastore to store different kinds of data - that is, data unrelated to the state defined above. In particular, input file data is also stored in the per-simulation `DataStore` instance.

Before any other kinds of state can be created, a mesh must be registered via `SetMesh()`. In order for a restart to work properly, all state data must be owned by the underlying `StateManager`, so ownership of the mesh is transferred via a `unique_ptr`. In the case of multi-mesh simulations, a name should also be used to uniquely identify the mesh.

Individual physics modules - that are of course based on these kinds of state - can now be constructed. In general, this process looks something like the following:

1. The physics module constructor accepts a mesh pointer and forwards it to the `BasePhysics` constructor. This parameter is required only in multi-mesh configurations and defaults to the default mesh otherwise. Although meshes also have string-valued tags associated with them, a user of a physics module would find it more intuitive to pass the mesh pointer they wish to use. Specifically, the meaning of a mesh parameter is much easier to discern than a string parameter.
2. The physics module creates its fields (e.g., temperature for a thermal conduction module) via calls to `StateManager::newState()`. In addition to the `serac::FiniteElementState` constructor arguments, this method also accepts a string-valued tag for the mesh with which the field is associated. The appropriate tag is a member of `BasePhysics` and initialized in the previous step. FIXME: Should we provide a protected helper method in `BasePhysics` so derived modules don't need to reference the member explicitly? Or perhaps `StateManager::newState()` et al should just take a mesh pointer instead of a tag?
3. The `FiniteElementState` is then constructed and registered in the corresponding `MFEMSidreDataCollection`. The only tricky part about this process is the need for the underlying `GridFunction` to be allocated within Sidre. This required an additional option to the

`FiniteElementState` constructor that leaves the vector data uninitialized (aka `nullptr`). After we create the `FiniteElementState` we register its underlying `GridFunction` in the `MFEMSidreDataCollection` and zero it out.

4. When the user wishes to save simulation state to disk, they can call `outputState()` on their physics module. FIXME: This could be confusing because this will call the global `StateManager::save()` which will save all data associated with a particular mesh. In particular we wouldn't want users to save twice if they have two physics modules on the same mesh (by calling `outputState` on each).

The use of `serac::QuadratureData<T>` for material state data is discussed *here*.

Restart Workflow

The "metadata" `StateManager` uses for choosing a restart file is the cycle (aka step number). These are used in `StateManager::save()` and `StateManager::load()` and subsequently as part of the filename written to disk.

As with a nominal workflow, the user must call `initialize()`. Note that while in the nominal case the directory parameter refers to the directory to which data will be saved, in a restart case this is also the directory from which data will be loaded.

Because the mesh already exists in the save file from which we're restarting, there is no need to call `setMesh()`. Instead, the user calls `StateManager::load()`, passing it the cycle number from which they wish to restart and the tag identifying the mesh. As in the nominal case, this tag is not necessary for single-mesh simulations.

> **Warning:** Because the mesh tag is used in the filename, it must exactly match the tag used in the call to `setMesh()` in the previous simulation run.

`StateManager::load()` will reconstruct the `mfem::ParMesh`, `mfem::GridFunction`, and `mfem::QuadratureFunction` objects. The `StateManager` factory methods can be used in the exact same way as they would in a nominal run, though the internal logic is of course different. In particular, it will search through the restored data for a field with the requested name and use that instead of constructing a new field via the process described above.

QuadratureData

Serac's `QuadratureData` template is an abstraction over `mfem::QuadratureFunction`, the type used to store per-quadrature-point data. We implement this functionality in terms of `mfem::QuadratureFunction` so that we can store this data in `MFEMSidreDataCollection`, which implements `mfem::DataCollection::RegisterQField` (which accepts a `QuadratureFunction`).

Because `QuadratureFunction` only allows for floating-point data (as either scalars or vectors), `QuadratureData<T>` allows for the storage of arbitrary (user-defined) types via a double-buffer approach. That is, data is stored in a buffer of type `T[]` for easy access within the `serac::Functional` ecosystem (which natively supports `QuadratureData` instances) and then copied (via a bit_cast) to the `double[]` buffer encapsulated by an `mfem::QuadratureFunction` when we wish to save state to disk. In the case of a restart the process works in reverse - data is `bit_cast` 'ed from the `double[]` buffer to the `T[]` buffer.

To allow synchronization to occur only when necessary, the `StateManager` registers a reference to each `QuadratureData` in a type-erased (via virtual functions) callback list. This further layer of abstraction - called `SyncableData` - allows quadrature point data of varying types to be uniformly synchronized to the corresponding `mfem::QuadratureFunction` instances.

1.3.14 Developing a New Physics Module

Developers have two workflows for creating new physics modules:

1. Creating a new multi-physics module from existing physics modules.
2. Creating a new single physics PDE simulation module.

In the first case, construct the new physics module by including existing physics modules by composition. See the Thermal solid mechanics module for an example.

For the second case, starting with an existing physics module and re-writing it as necessary is a good practice. The following steps describe creation of a new physics module:

1. Create a new class derived from BasePhysics.
2. In the constructor, create new `std::shared_ptrs` to FiniteElementStates corresponding to each state variable in your PDE.
3. Link these states to the state pointer array in the `BasePhysics` class.
4. Create methods for defining problem parameters (e.g. material properties and sources).
5. Create methods for defining boundary conditions. These should be stored as BoundaryConditions and managed in the `BasePhysics`'s BoundaryConditionManager.
6. Override the virtual `completeSetup()` method. This should include construction of all of the data structures needed for advancing the timestep of the PDE.
7. Override the virtual `advanceTimestep()` method. This should solve the discretized PDE based on the chosen time integration method. This often requires defining `mfem::Operators` to use MFEM-based nonlinear and time integration methods.

1.3.15 Important Data Structures

- BasePhysics: Interface class for a generic PDE simulation module.
- BoundaryCondition: Class for storage of boundary condition-related data.
- BoundaryConditionManager: Storage class for related boundary conditions.
- EquationSolver: Class for solving nonlinear and linear systems of equations.
- FiniteElementState: Data structure describing a solution field and its underlying finite element discretization.

1.3.16 Source Code Documentation

Doxygen documentation for the Serac source code is located in the Doxygen directory.

1.4 Theory Reference

1.4.1 Heat Transfer

Strong Form

The heat transfer module solves the heat equation

$$c_p \rho \frac{\partial T}{\partial t} - \nabla \cdot (\kappa \nabla T) + s(x,t) f(T) = g(x,t)$$

subject to the boundary conditions

$$\begin{aligned} T(x,0) &= T_0(x) \\ T(x,t) &= T_D(x,t) && \text{on } \Gamma_D \\ \kappa \frac{\partial T}{\partial n} &= q(x,t) && \text{on } \Gamma_N \end{aligned}$$

where

$$\begin{aligned} T(x,t) &= \text{ temperature} \\ c_p &= \text{ specific heat capacity} \\ \rho &= \text{ density} \\ \kappa &= \text{ conductivity} \\ f(T) &= \text{ nonlinear reaction} \\ s(x,t) &= \text{ scaling function} \\ g(x,t) &= \text{ heat source} \\ T_0(x) &= \text{ initial temperature} \\ T_D(x,t) &= \text{ fixed boundary temperature} \\ q(x,t) &= \text{ fixed boundary heat flux.} \end{aligned}$$

Weak Form

We multiply this strong form of the PDE by an arbitrary function and integrate by parts to obtain the weak form

Find $T \in V$ such that

$$\int_\Omega \left(\left(c_p \rho \frac{\partial T}{\partial t} + s(x,t) r(u) - g(x,t) \right) v + \kappa \nabla T \cdot \nabla v \right) dV - \int_{\Gamma_N} q v \, dA = 0, \qquad \forall v \in \hat{V}$$

where

$$\begin{aligned} V &= \{ v \in H_1(\Omega) : v = T_D \text{ on } \Gamma_D \} \\ \hat{V} &= \{ v \in H_1(\Omega) : v = 0 \text{ on } \Gamma_D \} . \end{aligned}$$

Discretization

After discretizing by the standard continuous Galerkin finite element method, i.e.

$$T(x,t) = \sum_{i=1}^{N} \phi_i(x) u_i(t), \qquad v = \phi_j(x)$$

where ϕ_i are nodal shape functions and u_i are degrees of freedom, we obtain the discrete equations in residual form

$$\mathbf{M}\dot{\mathbf{u}} + \mathbf{K}\mathbf{u} + f(\mathbf{u}) - \mathbf{G} = \mathbf{0}$$

where

$$\begin{aligned} \mathbf{u} &= \text{degree of freedom vector (unknowns)} \\ \mathbf{M} &= \text{thermal mass (heat capacity or capacitance) matrix} \\ \mathbf{K} &= \text{thamal stiffness (thermal conductivity or conductance) matrix} \\ f(\mathbf{u}) &= \text{possibly nonlinear reaction vector} \\ \mathbf{G} &= \text{source vector.} \end{aligned}$$

This system can then be solved using the selected nonlinear and ordinary differential equation solution methods. For example, if we use the backward Euler method we obtain

$$\mathbf{M}\mathbf{u}_{n+1} + \Delta t(\mathbf{K}\mathbf{u}_{n+1} + f(\mathbf{u}_{n+1})) = \Delta t\mathbf{G} + \mathbf{M}\mathbf{u}_n.$$

Given a known $\mathbf{u}_n$, this is solved at each timestep Δt for $\mathbf{u}_{n+1}$ using a nonlinear solver, typically Newton's method. To accomplish this, the above equation is linearized which yields

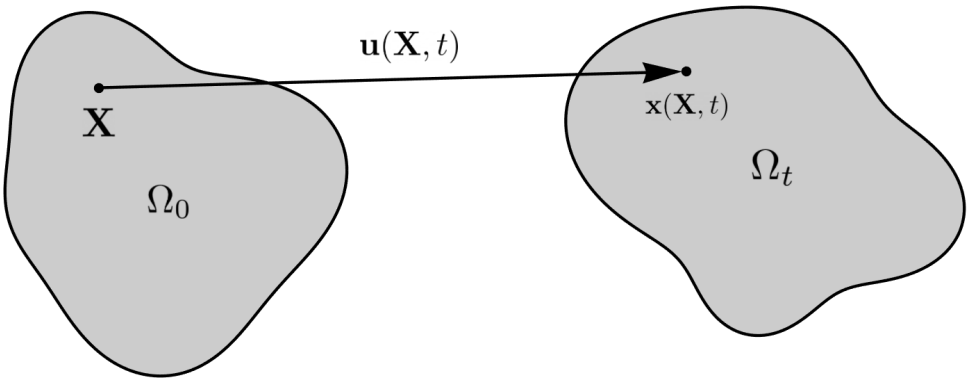
$$\left(\mathbf{M} + \Delta t\mathbf{K} + \Delta t\frac{\partial f}{\partial \mathbf{u}}\left(\mathbf{u}_{n+1}^{i}\right)\right)\Delta\mathbf{u}_{n+1}^{i+1} = -(\mathbf{M} + \Delta t\mathbf{K})\mathbf{u}_{n+1}^{i} - \Delta t f(\mathbf{u}_{n+1}^{i})) + \Delta t\mathbf{G} + \mathbf{M}\mathbf{u}_n.$$

Note the above equation assumes the thermal capacitance and conductance are independent of temperature. The computed Newton iterations $\mathbf{u}_{n+1}^{i+1} = \mathbf{u}_{n+1}^{i} + \Delta\mathbf{u}_{n+1}^{i+1}$ continue until the solution is converged.

1.4.2 Solid Mechanics

Strong Form

Consider the kinematics of finite deformation



where $\mathbf{x}(\mathbf{X}, t)$ is the current position of a point originally located at $\mathbf{X}$ in the undeformed (or reference) configuration. This motion is also commonly described in terms of the displacement

$$\mathbf{u}(\mathbf{X}, t) = \mathbf{x}(\mathbf{X}, t) - \mathbf{X}.$$

An important quantity characterizing this motion is the *deformation gradient*

$$\mathbf{F} = \frac{\partial \mathbf{x}}{\partial \mathbf{X}} = \frac{\partial \mathbf{u}}{\partial \mathbf{X}} + \mathbf{I}.$$

We also define the internal forces due to deformation in the solid in terms of the *Cauchy stress* σ. If the deformed body is cut by a surface with normal vector $\mathbf{n}$, the resulting traction vector $\mathbf{t}$ is defined as

$$\mathbf{t} = \sigma\mathbf{n}.$$

This stress is taken here as a function of the deformation gradient $\sigma = \sigma(\mathbf{F})$ by the appropriate hyperelastic constitutive (material) model. The conservation of angular momentum implies this stress tensor must be symmetric, i.e. $\sigma = \sigma^T$. We can then use the conservation of linear momentum to formulate the boundary value problem

$$\begin{aligned}
\nabla_{\mathbf{x}} \cdot \sigma(\mathbf{F}) + \rho\mathbf{b} &= \rho\ddot{\mathbf{u}} \\
\mathbf{u} &= \mathbf{u}_D && \text{on } \Gamma_D \\
\sigma\mathbf{n} &= \mathbf{t} && \text{on } \Gamma_N \\
\mathbf{u}(\mathbf{x}, 0) &= \mathbf{u}_0 \\
\dot{\mathbf{u}}(\mathbf{x}, 0) &= \dot{\mathbf{u}}_0
\end{aligned}$$

where

$$\begin{aligned}\sigma(\mathbf{F}) &= \text{Cauchy stress via constitutive response}\\ \rho &= \text{density}\\ \mathbf{b} &= \text{body force}\\ \mathbf{u}_D &= \text{fixed boundary}\\ \mathbf{t} &= \text{boundary traction}\\ \mathbf{u}_0 &= \text{initial displacement}\\ \dot{\mathbf{u}}_0 &= \text{initial velocity}\end{aligned}$$

and $\nabla_{\mathbf{x}}$ implies the gradient with respect to the current (deformed) configuration.

Weak Form

Multiplying the PDE by a vector-valued test function $\delta\mathbf{v}$ and integrating by parts yields the weak form

Find $\mathbf{u} \in \mathbf{U}$ such that

$$\int_{\Omega_t} \left(\sigma(\mathbf{u}) \cdot \nabla_{\mathbf{x}} \delta\mathbf{v} - \rho\mathbf{b} \cdot \delta\mathbf{v}\right) dV - \int_{\Gamma_{N_t}} \delta\mathbf{v} \cdot \mathbf{t}\, dA + \int_{\Omega_t} \rho\ddot{\mathbf{u}} \cdot \delta\mathbf{v}\, dV = 0, \qquad \forall \delta\mathbf{v} \in \hat{\mathbf{U}}$$

where

$$\begin{aligned}\mathbf{U} &= \left\{\mathbf{u} \in H_1^{\text{dim}}(\Omega) : \mathbf{u} = \mathbf{u}_D \text{ on } \Gamma_D\right\}\\ \hat{\mathbf{U}} &= \left\{\mathbf{u} \in H_1^{\text{dim}}(\Omega) : \mathbf{u} = \mathbf{0} \text{ on } \Gamma_D\right\}.\end{aligned}$$

and Ω is the current (deformed) configuration. In mechanics, the weak form is often referred to as the *principle of virtual power*. As Serac uses hyperelastic models, it is convenient to write this equation in the reference (undeformed) configuration

$$\begin{aligned}\int_{\Omega_0} \sigma(\mathbf{u}) \cdot \left(\nabla_{\mathbf{X}} \delta\mathbf{v}\mathbf{F}^{-1}\right) \det\mathbf{F}\, dV_0 - \int_{\Omega_0} \rho_0\mathbf{b} \cdot \delta\mathbf{v} dV_0\\ - \int_{\Gamma_{N_0}} \delta\mathbf{v} \cdot \mathbf{t}\, ||\mathbf{F}^{-T}\mathbf{n}_0|| \det\mathbf{F}\, dA_0 + \int_{\Omega_0} \rho_0\ddot{\mathbf{u}} \cdot \delta\mathbf{v}\, dV_0 = 0, \qquad \forall \delta\mathbf{v} \in \hat{\mathbf{U}}\end{aligned}$$

where ∇_X is the gradient with respect to the reference (material) coordinates.

Material Models

Serac currently is restricted to *hyperelastic* material formulations, i.e. materials that behave in a reversibly elastic fashion under large deformations. Mathematically, this implies they are derived from a *strain energy density* function $W = W(\mathbf{F})$. It can be shown that

$$\sigma(\mathbf{F}) = \frac{1}{\det\mathbf{F}} \frac{\partial W}{\partial \mathbf{F}} \mathbf{F}^T = \frac{1}{\det\mathbf{F}} \mathbf{P}\mathbf{F}^T$$

where

$$\mathbf{P} = \frac{\partial W}{\partial \mathbf{F}} = \det\mathbf{F}\sigma\mathbf{F}^{-T}$$

is the *first Piola-Kirchhoff stress*. Serac currently only has two material models. First, a neo-Hookean material where

$$\begin{aligned}W(\mathbf{F}) &= \frac{\mu}{2}(\bar{I}_1 - \text{dim}) + \frac{K}{2}(\det\mathbf{F} - 1)^2\\ \bar{I}_1 &= \frac{\text{trace}(\mathbf{F}\mathbf{F}^T)}{(\det\mathbf{F})^{2/\text{dim}}}\end{aligned}$$

and μ and K are the shear and bulk modulus, respectively. This definition also implies that the 2D simulations are using a plane strain assumption. The second model is a small strain isotropic linear elastic material where

$$\begin{aligned}\sigma(\epsilon) &= \lambda \text{trace}(\epsilon)\mathbf{I} + 2\mu\epsilon \\ \epsilon &= \frac{1}{2}\left(\mathbf{F} + \mathbf{F}^T\right) - \mathbf{I} \\ \lambda &= K - \frac{2}{\text{dim}}\mu\end{aligned}$$

and ϵ is the linearized strain tensor. Note that this model is only valid for small strains where the neo-Hookean model is nearly equivalent. It is included mostly for testing purposes.

Optionally, we can add a Kelvin-Voigt linear visoelastic term by adding

$$\sigma(\dot{\epsilon}) = \eta\dot{\epsilon}$$

to the stress calculations in dynamic simulations.

Discretization

We discretize the displacement field using nodal shape functions, i.e.

$$\mathbf{u}(\mathbf{X}) = \sum_{a=1}^{n} N^a(\mathbf{X})\mathbf{u}^a$$

where $\mathbf{u}^a$ are the degrees of freedom. We can then calculate the deformation gradient by

$$\mathbf{F} = \mathbf{I} + \sum_{a=1}^{n} \frac{\partial N^a}{\partial \mathbf{X}}\mathbf{u}^a$$

and substitute these quantities back into the weak form to obtain the vector-valued discrete residual equation

$$\int_{\Omega_0} \sigma \frac{\partial N^a}{\partial \mathbf{X}}\mathbf{F}^{-1}\text{det}\mathbf{F}\, dV_0 - \int_{\Omega_0} \rho_0 \mathbf{b} N^a dV_0 - \int_{\Gamma_{N_0}} \mathbf{t}^* N^a\, dA_0 + \int_{\Omega_0} \rho_0 \ddot{\mathbf{u}} N^a\, dV_0 = 0$$

where $\mathbf{t}^*$ is the traction applied in the reference configuration.

Optionally, we allow disabling the geometric nonlinearities by setting $\mathbf{F} = \mathbf{I}$ everywhere in this residual evaluation except for the material response (stress) function.

Performing these integrals yields the discrete equations

$$H(\mathbf{u}) - \mathbf{f} - \mathbf{g} + \mathbf{M}\ddot{\mathbf{u}} = 0$$

where

$$\begin{aligned}\mathbf{u} &= \text{displacement degree of freedom vector (unknowns)} \\ \mathbf{M} &= \text{mass matrix} \\ H(\mathbf{u}) &= \text{nonlinear internal force vector} \\ \mathbf{f} &= \text{body force} \\ \mathbf{g} &= \text{traction vector.}\end{aligned}$$

This discrete nonlinear second order ODE system can now be solved using the selected linear algebra methods.

Material Parameters:

Material models in serac may use different parameters for describing elastic properties. Specifying any two of these parameters lets you calculate the rest. The tool below can be used to perform these conversion calculations (assuming 3D):

J2 Linear Hardening Parameters

The hardening constants, H_i, H_k, in our J2 material model describe the extent to which the yield surface dilates and translates, respectively, when undergoing plastic deformation. The following animations illustrate the evolution of the yield surface and stress-strain relationship when subjected to cyclic strain, for different choices of H_i, H_k.

"Perfectly Plastic" response: zero isotropic and kinematic hardening

---

isotropic hardening only

---

kinematic hardening only

---

isotropic and kinematic hardening

1.4.3 Dirichlet Enforcement for Ordinary Differential Equations

Unconstrained Ordinary Differential Equations

Consider a system described by then following linear, second order differential equation:

In order to numerically integrate this differential equation, we can start by expressing the the future state as an extrapolation of the current state (e.g. using backward Euler):

$$\begin{cases} \mathbf{x}(t+\Delta t) = \mathbf{x}(t) + \dot{\mathbf{x}}(t)\Delta t + \ddot{\mathbf{x}}(t+\Delta t)\Delta t^2 \\ \dot{\mathbf{x}}(t+\Delta t) = \dot{\mathbf{x}}(t) + \ddot{\mathbf{x}}(t+\Delta t)\Delta t \end{cases}$$

Substituting this assumption back in to the differential equation evaluated at time $t + \Delta t$ produces an expression that relates a future state: $\{\mathbf{x}(t+\Delta t), \dot{\mathbf{x}}(t+\Delta t)\}$ to the current state $\{\mathbf{x}(t), \dot{\mathbf{x}}(t)\}$:

$$\mathbf{M}\,\ddot{\mathbf{x}}(t+\Delta t) + \mathbf{C}\left(\dot{\mathbf{x}}(t) + \ddot{\mathbf{x}}(t+\Delta t)\Delta t\right) + \mathbf{K}\left(\mathbf{x}(t) + \dot{\mathbf{x}}(t)\Delta t + \ddot{\mathbf{x}}(t+\Delta t)\Delta t^2\right) = \mathbf{0}$$

After rearranging terms to put the known quantities on the righthand side, we are left with

$$\left(\mathbf{M} + \mathbf{C}\Delta t + \mathbf{K}\Delta t^2\right)\,\ddot{\mathbf{x}}(t+\Delta t) = -\mathbf{C}\,\dot{\mathbf{x}}(t) - \mathbf{K}\big(\mathbf{x}(t) + \dot{\mathbf{x}}(t)\Delta t\big)$$

From here, provided that the coefficient matrix is invertible, a system of equations can be solved to determine $\ddot{\mathbf{x}}(t+\Delta t)$ and advance the solution state to time $t + \Delta t$.

Imposing Single Degree-of-Freedom Constraints

Now, let us consider how the solution process changes if some components of $\mathbf{x}(t)$ are controlled externally. In this case, our differential equation looks like

where $\mathbf{G}$ is a matrix with 1 on the diagonals corresponding to constrained components, and 0 elsewhere, and $\mathbf{g}(t)$ is the vector of prescribed values for the constrained degrees of freedom. There are several equivalent ways to interpret the additional constraint equations (we assume the constraints are consistent with the initial conditions):

$$\mathbf{G}\ \mathbf{x}(t) = \mathbf{g}(t) \iff \mathbf{G}\ \dot{\mathbf{x}}(t) = \dot{\mathbf{g}}(t) \iff \mathbf{G}\ \ddot{\mathbf{x}}(t) = \ddot{\mathbf{g}}(t)$$

At the end of the day, we are solving for a single acceleration vector, $\ddot{\mathbf{x}}(t + \Delta t)$, and as a result, we cannot hope to simultaneously satisfy more than one interpretation of these constraints. The different Dirichlet Enforcement Methods in serac relate to the following different interpretations of the constraint equations.

"Direct Control"

Direct control proceeds to solve for accelerations by effectively replacing the constrained rows of the differential equation by the constraint equations,

$$(\mathbf{1} - \mathbf{G})(\mathbf{M}\ \ddot{\mathbf{x}}(t + \Delta t) + \mathbf{C}\ \dot{\mathbf{x}}(t + \Delta t) + \mathbf{K}\ \mathbf{x}(t + \Delta t)) + \mathbf{G}\ \mathbf{x}(t + \Delta t) \ = \ \mathbf{g}(t + \Delta t)$$

Now, like before, we substitute in expressions for extrapolated state and solve for $\ddot{\mathbf{x}}(t + \Delta t)$. In this interpretation of the constraint, the resulting accelerations ensure that $\mathbf{G}\ \mathbf{x}(t + \Delta t) = \mathbf{g}(t + \Delta t)$ is satisfied.

"Rate Control"

Rate control uses a similar idea, but with the rate-form interpretation of the constraint:

$$(\mathbf{1} - \mathbf{G})(\mathbf{M}\ \ddot{\mathbf{x}}(t + \Delta t) + \mathbf{C}\ \dot{\mathbf{x}}(t + \Delta t) + \mathbf{K}\ \mathbf{x}(t + \Delta t)) + \mathbf{G}\ \dot{\mathbf{x}}(t + \Delta t) \ = \ \dot{\mathbf{g}}(t + \Delta t)$$

Here, the end-step accelerations are determined such that the derivative of the constraint equations is exactly satisfied, $\mathbf{G}\ \dot{\mathbf{x}}(t + \Delta t) = \dot{\mathbf{g}}(t + \Delta t)$. This tends to have better stability than the "Direct Control" option, but is prone to solution "drift" over long periods of time.

"Full Control"

This last option starts by additively decomposing the solution vector into two parts: constrained and unconstrained:

$$\begin{cases} \mathbf{x}(t) = \mathbf{x}_c(t) + \mathbf{x}_u(t) = \mathbf{G}\ \mathbf{x}(t) + (\mathbf{1} - \mathbf{G})\ \mathbf{x}(t) \\ \dot{\mathbf{x}}(t) = \dot{\mathbf{x}}_c(t) + \dot{\mathbf{x}}_u(t) = \mathbf{G}\ \dot{\mathbf{x}}(t) + (\mathbf{1} - \mathbf{G})\ \dot{\mathbf{x}}(t) \\ \ddot{\mathbf{x}}(t) = \ddot{\mathbf{x}}_c(t) + \ddot{\mathbf{x}}_u(t) = \mathbf{G}\ \ddot{\mathbf{x}}(t) + (\mathbf{1} - \mathbf{G})\ \ddot{\mathbf{x}}(t) \end{cases}$$

From here, the constrained terms $\{\mathbf{x}_c(t), \dot{\mathbf{x}}_c(t), \ddot{\mathbf{x}}_c(t)\}$ are replaced by their prescribed values, $\{\mathbf{g}(t), \dot{\mathbf{g}}(t), \ddot{\mathbf{g}}(t)\}$ and only the unconstrained acceleration components, $\ddot{\mathbf{x}}_u(t + \Delta t)$, are solved for. For situations where $\mathbf{g}(t)$ has a discontinuous derivative, this approach may be preferable to Rate Control.

Note: time derivatives of $\mathbf{g}(t)$ are currently computed by central finite difference stencils