Sandia National Laboratories

User/Reference Guide for MrHyDE - A framework for solving Multi-resolution Hybridized Differential Equations Version 1.0

Timothy M. Wildey
Computational Mathematics Department
Center for Computing Research
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1318
tmwilde@sandia.gov

May, 2024
SAND2024-06292

U.S. DEPARTMENT OF ENERGY
NNSA National Nuclear Security Administration

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

ABSTRACT

This document serves as both a user guide and a developers guide for Trilinos-based software framework named MrHyDE - **M**ulti-**r**esolution **Hy**bridized **D**ifferential **E**quations, which has been designed to enable the use of high-performance and heterogeneous computational architectures to enable multiscale and multiresolution modeling for complex multiphysics applications. This framework was developed at Sandia National Labs (NM) in support of a Department of Energy, Office of Science, Early Career research project. The goal of this document is to provide a comprehensive reference guide to the installation, use, modification, and enhancement of MrHyDE by new users. Theoretical details are included as necessary to describe the algorithms.

ACKNOWLEDGMENT

This document describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the document do not necessarily represent the views of the U.S. Department of Energy or the United States Government. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

This document was supported by the DOE Office of Science Early Career Research Program.

This page intentionally left blank.

CONTENTS

LIST OF FIGURES

This page intentionally left blank.

LIST OF TABLES

This page intentionally left blank.

NOMENCLATURE

Table 0-1.

Abbreviation	Definition
DoE	Department of Energy
MrHyDE	Multi-resolution Hybridized Differential Equations
ROL	Rapid Optimization Library
BFS	Beyond Forward Simulation

This page intentionally left blank.

1. INTRODUCTION

This document serves as a theory, reference and user manual for a Trilinos-based framework design for solving Multi-resolution Hybridized Differential Equations, called MrHyDE. This state-of-the-art software framework combines lightweight interfaces to select Trilinos second-generation packages and a set custom performance portable managers to enable the solution of transient nonlinear strongly coupled multiphysics/multiscale problems with an emphasis on beyond forward simulations (BFS) capabilities e.g., large-scale PDE-constrained optimization, and basic sampling-based capabilities for uncertainty quantification, and measure-theoretic stochastic inversion.

While this document is primarily aimed at new users of Trilinos/MrHyDE, there are several detailed explanations of certain components of MrHyDE that should be valuable to all users/developers.

We note that MrHyDE is not the right tools for every PDE-based analysis. This is no graphical user interface (GUI) and there never will be one. It is expected that the user/developer will be sufficiently fluent in certain high-performance computing (HPC) tools, such as CMake and C++ compilers, to compile both Trilinos and MrHyDE. A python interface is currently under development, but the functionality will be limited. MrHyDE can be packaged into a Docker or Singularity container for rapid deployment to novice or external users. The target audience for MrHyDE is computational scientists looking for a scalable simulation framework that is modular, easy to modify, well-documented, portable from laptops to exascale, and automatically enables BFS and multiscale capabilities.

MrHyDE has three different operating modes: classical, multiscale, and fully-explicit. Each of these have different characteristics and the design of MrHyDE enables each of these to be optimized for both memory and performance. These modes will be described in detail in later sections, but briefly:

Classical: This is mode of MrHyDE that is duplicative of other software packages. In particular, this mode enables the implicit or explicit solution of large-scale strongly coupled multiphysics problems using physics-compatible discretizations.

Multiscale/Multiresolution: This is the operating mode that gives MrHyDE its name, and has driven much of the research that has supported its development. Generally speaking, MrHyDE allows a "coarse" mesh to define subgrid models with potentially different physical phenomena, including different formulations and approximations, and transfer information between the "coarse" scale model and the local subgrid models in a fully-implicit manner, which involves a nested nonlinear Newton procedure. The "coarse" mesh does not necessarily need to be coarse - it may contain millions and perhaps even billions of elements.

Fully-Explicit: While the classical mode can perform time-integration using implicit, explicit, or operator split schemes, it is not optimized for explicit time integration. In the fully-explicit mode, MrHyDE does not require automatic differentiation, which significantly improves performance and marginally improves memory, and has the option the option to use matrix-free algorithms for the inversion of mass matrices in the time stepping, which greatly improves memory usage with little performance loss.

The most recent version of MrHyDE includes all of these modes in a single. Previous versions required a separate, derivative-free, build for the fully-explicit mode to operate at maximum efficiency. While derivative-free builds are still supported, they are no longer necessary.

2. GETTING STARTED

2.1. Downloading MrHyDE

MrHyDE is an open source software framework currently stored in a repository on Sandia's external github site.

The repository can be cloned using:

```
git clone git@github.com:sandialabs/MrHyDE.git
```

2.2. Downloading Trilinos

Trilinos is an open source collection of packages stored on GitHub.

The repository can be cloned using:

```
git clone https://github.com/trilinos/Trilinos.git
```

We strongly recommend using either the master branch of Trilinos, which is tested nightly. We cannot possibly support builds on branches that have experimental features/capabilities. MrHyDE currently requires Trilinos version 13.5 for a stable build from source. Trilinos version 14 introduced a backwards incompatible change to the naming convention in certain packages. MrHyDE automatically detects the Trilinos version and adjust the headers/typedefs accordingly. As of February 2024, MrHyDE can utilize either version 14.5 or version-of-the-day Trilinos on the master branch.

2.3. Trilinos packages utilized

In this chapter, we briefly describe the Trilinos packages that MrHyDE *explicitly* uses and the capabilities they provide. Some of these packages, particularly Panzer, depend on several other packages in Trilinos, but these dependencies are not described here.

Teuchos Used for parameter lists, MPI communicators, exceptions

Kokkos Performance portability library. Used for storing arrays of data, called Kokkos Views, that have optimal layouts for the compile-time determined architecture. Also used for parallel loops.

Intrepid2 Second generation discretization library. Used for HGRAD, HVOL, HDIV and HCURL basis functions, reference element quadrature/cubature rules, pushforward and pullback mappings between reference and physical elements, and computing other elemental quantities such as normals and tangents.

Shards Cell topology library.

Sacado Automatic differentiation library. MrHyDE primarily uses Static Forward-mode automatic differentiation (SFad) types which require the maximum number of derivatives to be known at compile time. Dynamic Forward-mode automatic differentiation (DFad) types are used in a few places, but are not as efficient as the SFad types.

ROL Large scale optimization library.

PanzerDOF Provides a global indexer for continuous or discontinuous approximations using any of the basis functions available through Intrepid2. Also support multi-block meshes.

PanzerSTK Provides an interface to the STK mesh database.

Tpetra Large scale linear algebra library. Includes vectors, multi-vectors, maps, graphs, sparse and dense matrices.

Amesos2 Second generation library for solving sparse and dense linear systems using direct methods. Provides a unified interface to multiple solvers: KLU2 (default), Basker, SuperLU, MUMPS, cuSOLVER, etc.

Belos Second generation library for iterative solvers for sparse linear systems. GMRES is the default option in MrHyDE, but many other linear solvers are available.

MueLu Second generation multilevel preconditioning library. Primarily used for algebraic multigrid preconditioners.

Compadre Relatively new library for performing point-cloud operations using intra-node parallelism. MrHyDE uses the kd-tree functionality to import microstructure and heterogeneous fields based on point clouds.

2.4. Third Party Libraries

Trilinos requires a few third-party libraries (TPLs) that need to be installed before attempting to create a build. Most of these are fairly standard and all are available either through HomeBrew on a Mac, or through the HPC modules. At this time, we do not recommend using the SEMS modules on the HPC machines. A list of these TPLs and the current (as of February 18, 2024) versions which enable an appropriate Trilinos build is provided in Table 2.4.

On a mac machine, Trilinos and MrHyDE can be built using either gcc or clang compilers. We recommend clang for slightly better performance.

On the Sandia HPC machines, Trilinos and MrHyDE can be built using gcc, Intel or clang. All of these exhibit similar performance on the HPC machines. We recommend using one of the

TPL	Mac OS Ventura (HomeBrew)
CMake	3.27.6
clang	15.0.0
openmpi	4.1.5
zlib	1.3
netcdf	4.9.2
pnetcdf	1.12.3_1
Boost	1.83.0
HDF5	1.14.2
libx11	1.8.6
ninja (optional)	1.11.1

Table 2-1. Third Party Libraries and the recommended versions for Mac OS Ventura using HomeBrew.

following development packs: cde/v3/devpack/clang-ompi, cde/v3/devpack/gcc-ompi, cde/v3/devpack/intel-impi, cde/v3/devpack/intel-ompi.

2.5. Building Trilinos

Creating an appropriate build of Trilinos is likely to be the biggest challenge for new users.

On a personal machine, Trilinos will need to be built from source. While Trilinos/MrHyDE can be configured and optimized for a hybrid CPU/GPU environment, we strongly recommend starting with a standard build on either Mac or Linux machine first. At this time, building on Microsoft Windows is not supported and there are no plans to add this.

On a Mac, the following configure script will set up an appropriate build

```
#!/bin/bash
rm -rf CMakeCache.txt
rm -rf CMakeFiles

EXTRA_ARGS=$@

TRILINOS_HOME='<Trilinos Location>'

INSTALL_DIR='<Install Location>'

cmake \
-G Ninja \
-D CMAKE_BUILD_TYPE:STRING=NONE \
-D TPL_ENABLE_MPI:BOOL=ON \
-D CMAKE_CXX_FLAGS:STRING="-O3 -ansi -pedantic -ftrapv -Wall -Wno-long-long -Wno-strict-aliasing
     -DBOOST_NO_HASH -DBOOST_STACKTRACE_GNU_SOURCE_NOT_REQUIRED" \
-D CMAKE_C_FLAGS:STRING="-O3" \
-D CMAKE_Fortran_FLAGS:STRING="-O3" \
-D Trilinos_ENABLE_CHECKED_STL:BOOL=OFF \
-D Trilinos_ENABLE_EXPLICIT_INSTANTIATION:BOOL=ON \
-D Trilinos_ENABLE_INSTALL_CMAKE_CONFIG_FILES:BOOL=ON \
-D Trilinos_SKIP_FORTRANCINTERFACE_VERIFY_TEST:BOOL=ON \
-D Trilinos_ENABLE_EXAMPLES:BOOL=OFF \
-D Trilinos_ENABLE_TESTS:BOOL=OFF \
-D Trilinos_ENABLE_ALL_PACKAGES:BOOL=OFF \
-D Trilinos_ENABLE_ALL_OPTIONAL_PACKAGES:BOOL=OFF \
-D Trilinos_ENABLE_Belos:BOOL=ON \
-D Trilinos_ENABLE_ROL:BOOL=ON \
-D Trilinos_ENABLE_PanzerCore=ON \
-D Trilinos_ENABLE_PanzerDofMgr=ON \
-D Trilinos_ENABLE_PanzerMiniEM=OFF \
-D Trilinos_ENABLE_PanzerAdaptersSTK=ON \
-D Trilinos_ENABLE_Intrepid2:BOOL=ON \
-D Trilinos_ENABLE_Shards:BOOL=ON \
-D Trilinos_ENABLE_Amesos2:BOOL=ON \
-D Trilinos_ENABLE_Belos:BOOL=ON \
-D Trilinos_ENABLE_MueLu:BOOL=ON \
-D Trilinos_ENABLE_Percept:BOOL=OFF \
-D Trilinos_ENABLE_Compadre:BOOL=ON \
-D Trilinos_ENABLE_SEACAS:BOOL=ON \
-D PanzerAdaptersSTK_ENABLE_EXAMPLES=OFF \
-D PanzerAdaptersSTK_ENABLE_TESTS=OFF \
-D TPL_ENABLE_Matio=OFF \
-D STK_ENABLE_TESTS:BOOL=OFF \
-D Panzer_ENABLE_TESTS:BOOL=OFF \
-D Panzer_ENABLE_EXAMPLES:BOOL=OFF \
-D Panzer_ENABLE_EXPLICIT_INSTANTIATION:BOOL=ON \
-D PANZER_HAVE_PERCEPT:BOOL=OFF \
-D CMAKE_INSTALL_PREFIX:PATH=${INSTALL_DIR} \
-D TPL_ENABLE_Boost:BOOL=ON \
-D TPL_ENABLE_BoostLib:BOOL=ON \
-D TPL_ENABLE_Netcdf:BOOL=ON \
-D TPL_ENABLE_HDF5=ON \
-D SEACASExodus_ENABLE_MPI:BOOL=OFF \
-D TPL_ENABLE_GLM=OFF \
-D Kokkos_ENABLE_SERIAL=ON \
${EXTRA-ARGS} \
${TRILINOS_HOME}
```

Note that this build uses ninja, which is not a required TPL but is highly recommended. Builds using ninja are often much faster than the standard cmake builds. Delete this line if ninja is not available. Unfortunately, the combination of ninja, nvcc and cmake 3.17 or later causes configuration errors on heterogeneous nodes. Removing the ninja option usually fixes this problem.

2.6. Building MrHyDE

Building MrHyDE is straightforward given an appropriate Trilinos build. Note that MrHyDE effectively treats Trilinos as a third-party library, so the build process is separate, but similar.

The following configure script should work on any system:

```
EXTRA_ARGS=$@

rm CMakeCache.txt
rm -rf CMakeFiles/

TRILINOS_HOME='<Trilinos Location>'
TRILINOS_INSTALL='<Trilinos Install Location>'
MRHYDE_HOME='<MrHyDE Location>'
MRHYDE_INSTALL='<MrHyDE Install Location>'

cmake \
-G Ninja \
-D Trilinos_SRC_DIR=${TRILINOS_HOME} \
-D Trilinos_INSTALL_DIR=${TRILINOS_INSTALL} \
-D CMAKE_INSTALL_PREFIX:PATH=${MRHYDE_INSTALL} \
-D CMAKE_CXX_FLAGS:STRING="-Wall" \
-D MrHyDE_MAX_DERIVS=64
${EXTRA-ARGS} \
${MRHYDE_HOME}
```

The above configure script uses several default settings for MrHyDE. Since MrHyDE using Sacado static forward automatic differentiation (SFad) types, the number of derivatives (per element) is required at compile time. The default is 64 since this is the minimum to run some of the regression tests. Note that this value is the maximum number of derivatives that may be required on an element at any given time. For example, a 3D scalar equation using a first-order HGRAD basis will have 8 degrees of freedom, and therefore the number of derivatives can be set to 8. If performing optimization, the number of scalar parameters and the local number of discretized parameter degrees of freedom also need to be accounted for. In such cases, the maximum derivatives is the maximum of the local degrees-of-freedom, the local degrees-of-freedom for the discretized parameters, and the number of scalar parameters. MrHyDE has a relatively new autotune feature that determines the number of derivatives required and selects the most appropriate one from a set of built in AD types. The built-in types have 2, 4, 8, 16, 18, 24, or 32 derivatives. If your problem has over 32 derivatives per element, then the max derivatives must be set to the appropriate number. MrHyDE will crash with a warning if this is not set high enough.

Many assembly routines use parallel for loops with a Kokkos::TeamPolicy, which may involve both a team size and a vector size for hierarchical parallelism. The team size is computed automatically, but the vector size, which defaults to the number of derivatives, can be adjusted by changing

```
-D MrHyDE_VECTOR_SIZE=64
```

The default is to perform both assembly and solves on the host device, which is almost always Kokkos::Serial. These can be changed by turning on a subset of

```
-D MrHyDE_ASSEMBLY_CUDA=OFF
-D MrHyDE_SOLVER_CUDA=OFF
-D MrHyDE_ASSEMBLY_OPENMP=OFF
-D MrHyDE_SOLVER_OPENMP=OFF
```

if Trilinos was built with Cuda or OpenMP enabled.

MrHyDE also has a set of executables used to test and optimize certain components in various environments. These are located under

```
MrHyDE/sandbox
```

and can be enabled by turning on

```
-D MrHyDE_ENABLE_SANDBOX=OFF
```

MrHyDE has a limited set of unit tests, which can be enabled by setting:

```
  -D MrHyDE_ENABLE_UNIT_TESTS=OFF \
```

MrHyDE primarily relies on the unit testing in most of the packages in Trilinos to validate the lower-level capabilities. Unit testing in MrHyDE is not updated or maintained frequently.

For those special users associated with the Mirage project, the Mirage-specific extensions can be enabled by setting:

```
-D MrHyDE_ENABLE_MIRAGE_EXTENSIONS=ON \
-D MrHyDE_MIRAGE_EXTENSIONS_DIR=${MIRAGE_EXTENSIONS} \
```

where MIRAGE_EXTENSIONS is the directory containing the Mirage extension libraries.

In the fully-explicit operating mode, it is much better, in terms of both memory and performance, to disable the automatic differentiation since it is not needed. This can be disabled by setting:

```
  -D MrHyDE_DISABLE_AD=ON \
```

In the current version of MrHyDE, setting this option is not necessary since a separate build of MrHyDE is not required to use the fully-explicit mode. We do maintain the ability to create a derivative-free build.

In theory, MrHyDE can be run using single-precision arithmetic by setting:

```
  -D MrHyDE_SINGLE_PRECISION=ON \
```

This capability has not been tested thoroughly and should not be used unless you know what you are doing.

Finally, MrHyDE also has DOxygen documentation which can be built by setting:

```
  -D MrHyDE_BUILD_DOXYGEN=ON \
```

2.7. Running MrHyDE

Assuming that both Trilinos and MrHyDE have been built and the MrHyDE build directory is

```
MrHyDE/build
```

the MrHyDE executable is in

```
MrHyDE/build/src/mrhyde
```

Running the code requires either installing the executable on the working path, or creating a soft link in a different directory. For example, running the regression tests requires the following

```
>> cd MrHyDE/regression
>> ln -s ../build/src/mrhyde
>> python runtests.py
```

In general, if a given directory has a soft link to the executable, the code can be run using MPI:

```
>> mpiexec -n 1 mrhyde input.yaml
```

where "n" is the number of MPI processes. The input file specification is optional if the default input file (input.yaml) is used. Each folder in the regression library has an example input file. See

```
MrHyDE/scripts/input-files/input-defaults.yaml
```

for an input file with many, but probably not all, of the default values. This file is just for reference and should not be used to run MrHyDE.

By design, MrHyDE is fairly quiet in terms of output to the screen. This is due to the emphasis on BFS capabilities, which typically require several forward model evaluations. Like most of the Trilinos packages, the amount of screen output is controlled via a verbosity flag.

```
%YAML 1.1
---
ANONYMOUS:
  verbosity: 0
  debug level: 0
...
```

Setting the verbosity to at least 10 will output linear/nonlinear solver progress, time integration status, and the Teuchos timers. Another flag, shown above, is the debug level. This is useful for developers who want to dig into the code progress without having to recompile. Setting this flag to 1 will print a statement when starting and finishing most constructors and interface/manager functions that only get called once, typicall during the set up phase. Setting this flag to 2 will also print when entering/leaving functions that get called many times, typically during the solve phase. Finally, setting this flag to 3 will also print vectors, matrices and certain Kokkos views.

2.8. Creating a Mesh

In this chapter, we describe how to define a mesh and the various options and limitations of various approaches.

MrHyDE is able to use the following element topologies:

1D: intervals

2D: triangles, quadrilaterals

3D: tetrahedral, hexahedral

There are two mechanisms for created meshes based on any of the element topologies listed above:

Cubit/Exodus The most flexible approach uses Cubit (or another mesh generation software, but we recommend Cubit) to create an Exodus mesh. These meshes can be structured or unstructured with complete user control over the number and placement of element blocks, side sets and node sets. Note that Cubit does not decompose the mesh for parallel processing, so the mesh must fit within the available memory on the system.

Panzer inline The Panzer inline mesh is a convenient tool for creating structured meshes for simple problems and verification/testing. It provides user control over the number of elements in each dimension and the parallel decomposition.

If using an Exodus mesh, only a few parameters need to be added to the mesh block in the input file:

```
Mesh:
    dimension: 2
    source: Exodus
    mesh file: mymesh.exo
```

If using the inline Panzer mesh, then a few additional parameters may be needed:

```
Mesh:
    dimension: 3
    element type: hex
    xmin: 0.0
    xmax: 1.0
    ymin: 0.0
    ymax: 1.0
    zmin: 0.0
    zmax: 1.0
    NX: 10
    NY: 10
    NZ: 10
    Xblocks: 1
    Yblocks: 1
    Zblocks: 1
    Xprocs: 1
    Yprocs: 1
    Zprocs: 1
```

Most of these have default values. For example, the number of element blocks in each dimension is 1 by default. If the number of processors in each dimension is provided, then the product of these numbers needs to match the total number of MPI ranks. If the number of processors in each dimension is not provided, the mesh will only be decomposed in the x-direction. While this insures that the number of partitions matches the total number of MPI ranks, it may lead to unexpected behavior if performing a strong scaling test.

If using a Cubit-generated Exodus mesh, then you will need to decompose the mesh before running a parallel job. Fortunately, this is straightforward using tools provided in Trilinos. On the HPC systems, one can simple load the tools and decompose the mesh as follow:

```
>> module load seacas
>> decomp -p 64 mymesh.exo
```

No changes are required in the MrHyDE input file. If running on a local machine, Trilinos can be configured to build seacas and create the decomposition tools.

2.9. Performance Tips

Most of the default settings in MrHyDE are meant to provide robustness across the suite of regression, and are not necessarily optimal for performance. Here are a few suggestion to improve the performance (run time), memory usage, or both:

- Do not use a debug build of Trilinos. While this is occasionally useful for development purposes, it significantly slows down certain libraries by up to 10x.
- Use profilers to identify where the code is spending most of its time. In the input file, turning the verbosity up to 10 or higher will tell MrHyDE to print all of the Teuchos timers that were defined. These timers are quite prevalent in the code and viewing this data often provides solid insights. Another option is to use the Kokkos profiling options to see if certain Kokkos kernels are slow or to monitor the amount of memory used in the various Kokkos views.
- If assembly time is large, make sure that you have not disabled autotune (see Remark 2.9.1). If autontune is not working for you, try adjusting the number of derivatives used for the Sacado SFad objects in the cofigure script for MrHyDE, e.g.,

  ```
  -D MrHyDE_MAX_DERIVS=16
  ```

  Changing this will require a full rebuild of MrHyDE, but not Trilinos. The assembly often scales almost linearly with this value, so significant performance gains can be obtaining by tightening this as much as possible. This number only needs to be the maximum of the number of state degree-of-freedom, the number of discretized parameter degrees-of-freedom and the number of active parameters. For example, linear elasticity in 3D without any discretized or active parameters requires only 24 derivatives in the SFad objects. The default value is 64 which enables all of the regression tests to run.
- If assembly is slow, adjust the workset size. This is the number of elements that get processed together in the assembly (or subgrid solves). The default is 100, and this is often quite reasonable on a CPU node with Kokkos::Serial for the intra-node parallelism. However, increasing the workset size can result in better performance on a CPU in certain circumstances. If performing assembly on the GPU, it is often necessary to use workset sizes of 1000+ to maximize performance. Figure 2.9 compares the affect of adjusting the number of derivatives versus adjusting the workset size for various steps in the assembly procedure.
- If memory is a concern and the mesh is sufficiently regular/structured, turn on the database compression. This will only store the unique information on a given processor which can reduce memory usage by up to 95%.

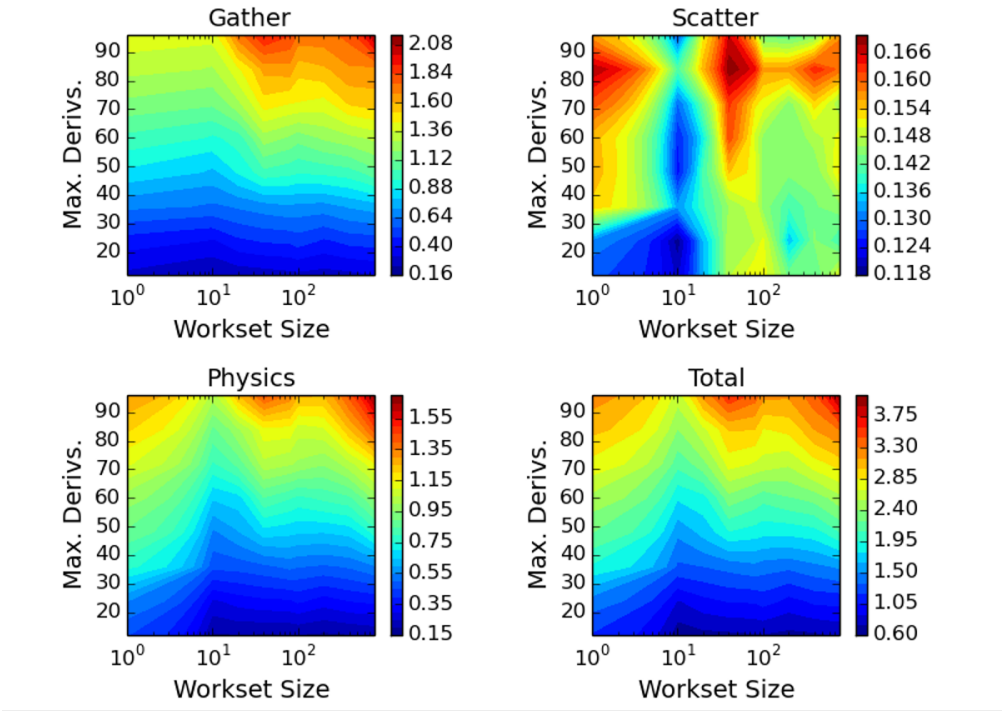

Figure 2-1. A comparison the affect of adjusting the number of derivatives versus adjusting the workset size for various steps in the assembly procedure. This was run on a CPU using Kokkos::Serial for the AssemblyDevice.

- To decrease memory a little bit further, decrease the workset size. The workset size determines the size of certain Kokkos views. In particular, the views associated with the workset and function manager that store AD objects. The storage of these array scales linearly with the workset size.
- If memory is still a concern, turn off the storage of the integration information and basis functions. By default, these are only computed once and stored in the cells and boundary cells. This might be prohibitive if a processor owns a very large number of elements. The drawback will be an increase in assembly time.
- Use static data for initial and boundary conditions whenever possible. This will avoid repeated L_2-projections.
- Adjusting certain linear solver parameters can improve both performance and memory.
- Disable postprocessing you don't really need. For some problems, this may take just as long as taking a time step.
- Check your mesh partitioning and load balancing. Run a problem with just a few time steps and investigate the load balancing in the timer output. Note that this data can be misleading if the imbalance is in a function that does not require an MPI synchronization afterwards.

Remark 2.9.1 *In the latest version of MrHyDE, the physics modules, worksets and function managers are all templated on the evaluation type (double, AD, etc). This allows MrHyDE to evaluate residuals at least an order-of-magnitude faster than previous versions, which is useful in both the traditional mode and the fully-explicit mode. The multiscale-hybridized mode does not currently take advantage of this. In addition, MrHyDE now comes with several built-in AD types in addition to the type defined by the user in the configure script. The built-in AD types have 2, 4, 8, 16, 18, 24 or 32 derivatives. With both of these advances, MrHyDE can now autotune performance by selecting the evaluation type that is optimal for a given problem. This significantly reduces the need for specialized build of MrHyDE for specific problems. However, if a simulation requires more than 32 derivatives per element, then the user will need to adjust the maximum derivatives in the configure script as described above. As in previous versions, MrHyDE will inform the user if the number of derivatives is not high enough.*

2.10. Regression Testing

Contributions to MrHyDE are welcome and encouraged. We only request that a set of regression tests pass before commits are made to the master branch.

These regression tests are contained within

`MrHyDE/regression`

and running these tests requires creating a soft link to the MrHyDE executable and running a Python script:

```
>> cd regression
>> ln -s ../build/src/mrhyde
>> python runtests.py
```

The current version of the Python script works with python3.

Typical output look like:

```
Sun Jul 11 08:29:05 2021
Test Results from Directory: /Users/tmwilde/Software/MrHyDE/regression
Total number of test(s): 96
-----------------------------------------------------------------------------------------------
    1/96       pass    0.77s  np=4                                        maxwell/PlaneWave
    2/96       pass    2.64s  np=4                                      phasefield/2d-3phi
    3/96       pass    0.44s  np=4                           porous/HGRAD_2D_preconditioner
    4/96       pass    0.40s  np=4                             porous/HGRAD_2D_verification
    5/96       pass    0.49s  np=4                         thermal/2D_verification_highorder
    6/96       pass    0.43s  np=4                               thermal/2D_verification_mpi
    7/96       pass    0.40s  np=4              thermal/2D_verification_transient_lumpedmass
    8/96       pass    0.83s  np=4                       thermal/2D_transient_source_control
    9/96       pass    0.56s  np=4                                thermal/3D_verification
   10/96       pass    0.44s  np=4                        thermal/2D_verification_nonzeroDBC
   11/96       pass    0.37s  np=4                  thermal/2D_verification_multiscale_HFACE
   12/96       pass    0.95s  np=4              thermal/3D_verification_multiscale_panzermesh
   13/96       pass    0.45s  np=4               thermal/2D_verification_multiscale_transient
   14/96       pass    2.10s  np=4       thermal/2D_verification_multiscale_dynamicmultimodel
   15/96       pass    0.94s  np=4                       thermal/3D_verification_multiscale
   16/96       pass    0.47s  np=4                 thermal/2D_verification_tri_highorder
   17/96       pass    1.48s  np=4                         thermal/2D_transient_fd_check
   18/96       pass    0.58s  np=4              thermal/2D_verification_multiscale_multimodel
```

```
 19/96       pass    0.37s  np=4                        thermal/2D_verification_multiscale
 20/96       pass    0.38s  np=4              thermal/2D_verification_multiscale_panzermesh
 21/96       pass    0.40s  np=4                                thermal/2D_verification
 22/96       pass    0.69s  np=4              thermal/3D_verification_multiscale_exodusmesh
 23/96       pass    0.75s  np=4                           thermal/2D_verification_transient
 24/96       pass    2.14s  np=4                                   maxwell_fp/3D_verfication
 25/96       pass    0.76s  np=4                                       shallowwater/droptest
 26/96       pass    1.83s  np=4                             helmholtz/manufactured_solution
 27/96       pass    0.73s  np=3                                    thermal/3D-Multiblock
 28/96       pass    0.80s  np=2                          thermal/2D_Data_Generating_Inversion
 29/96       pass    0.73s  np=1                                  thermoelastic/2D_transient
 30/96       pass    0.30s  np=1                                          ODE/CrankNicolson
 31/96       pass    0.31s  np=1                                                 ODE/custom
 32/96       pass    0.30s  np=1                                               ODE/DIRK-3,3
 33/96       pass    0.30s  np=1                                                 ODE/RK-4,4
 34/96       pass    0.34s  np=1                                 ODE/DIRK-1,2-Optimization
 35/96       pass    0.34s  np=1                                      ODE/BWE-Optimization
 36/96       pass    0.29s  np=1                                                   ODE/BDF2
 37/96       pass    0.30s  np=1                                             ODE/SSPRK-3,3
 38/96       pass    0.30s  np=1                                               ODE/DIRK-2,3
 39/96       pass    0.30s  np=1                                                   ODE/BDF3
 40/96       pass    0.31s  np=1                                                   ODE/BDF4
 41/96       pass    0.30s  np=1                                               ODE/DIRK-2,2
 42/96       pass    0.30s  np=1                                               ODE/DIRK-1,2
 43/96       pass    0.29s  np=1                                                    ODE/FWE
 44/96       pass    0.29s  np=1                                                    ODE/BWE
 45/96       pass    0.42s  np=1                                               vdns/channel
 46/96       pass    0.70s  np=1                                          cdr/2D_transient
 47/96       pass    0.62s  np=1                                        cdr/2D_ns_coupled
 48/96       pass    0.47s  np=1                                      cdr/2D_manufactured
 49/96       pass    0.61s  np=1                                             cdr/periodic
 50/96       pass    0.65s  np=1                                     UQ/Embedded-Sampling
 51/96       pass    0.66s  np=1                                   UQ/User-Defined-Sampling
 52/96       pass    0.32s  np=1                                         stokes/channel
 53/96       pass    0.31s  np=1                                stokes/2D_verification_pspg
 54/96       pass    0.34s  np=1                           porous/HDIV_hybrid_multiscale2
 55/96       pass    0.32s  np=1                              porous/HDIV_weakGalerkin_AC
 56/96       pass    0.30s  np=1                                       porous/HDIV_tri
 57/96       pass    0.40s  np=1                                    porous/HDIV_3D_tet
 58/96       pass    0.32s  np=1                                  porous/HDIV_PermData
 59/96       pass    0.97s  np=1                               porous/HDIV_weakGalerkin_3D
 60/96       pass    0.46s  np=1                             porous/HDIV_hybrid_multiscale
 61/96       pass    0.39s  np=1                          porous/HDIV_weakGalerkin_highorder
 62/96       pass    0.43s  np=1                                       porous/HDIV_3d
 63/96       pass    0.30s  np=1                                          porous/HDIV
 64/96       pass    0.33s  np=1                         porous/HDIV_hybrid_multiscale_1D
 65/96       pass    0.32s  np=1                                  porous/HDIV_hybrid
 66/96       pass    0.63s  np=1                 porous/HDIV_weakGalerkin_hybrid_multiscale
 67/96       pass    0.77s  np=1                                  porous/HDIV_3D_hybrid
 68/96       pass    0.30s  np=1                                       porous/HDIV_1D
 69/96       pass    0.31s  np=1                               porous/HDIV_weakGalerkin_2D
 70/96       pass    0.62s  np=1                          porous/HDIV_TET_hybrid_multiscale
 71/96       pass    0.32s  np=1                          porous/HDIV_weakGalerkin_PermData
 72/96       pass    0.36s  np=1                               porous/HDIV_hybrid_highorder
 73/96       pass    0.31s  np=1                                     porous/HDIV_hybrid_tri
 74/96       pass    0.49s  np=1                                thermal/2D_verification_tri
 75/96       pass    0.42s  np=1              thermal/2d_gradient_check_integrated_response
 76/96       pass    0.36s  np=1              thermal/2d_gradient_check_sensor_gradresponse
 77/96       pass    0.46s  np=1                                    thermal/2D_mixed_bcs
 78/96       pass    0.43s  np=1                          thermal/2d_gradient_check_non-ms
 79/96       pass    1.79s  np=1                               thermal/3D_verification_tet
 80/96       pass    0.36s  np=1                                     thermal/2D_multiblock
 81/96       pass    0.39s  np=1          thermal/2D_transient_mass_single_scale_inversion
 82/96       pass    0.63s  np=1                            thermal/2D_create_sensor_data
 83/96       pass    0.43s  np=1                          thermal/2D_integrated_quantities
 84/96       pass    0.35s  np=1                thermal/2d_gradient_check_sensor_response
 85/96       pass    0.95s  np=1                         le/crystal_elasticity_multiscale
 86/96       pass    0.76s  np=1                                      le/3D_manufactured
 87/96       pass    1.31s  np=1                               le/2d_sparse_simul_inversion
 88/96       pass    0.32s  np=1                     le/2d_uniaxial_tension_cubit_multiscale
 89/96       pass    0.32s  np=1                               le/2d_uniaxial_tension_cubit
 90/96       pass    0.49s  np=1                                  le/2d_two_disc_inversion
 91/96       pass    0.48s  np=1                                    le/2d_stress_inversion
 92/96       pass    0.41s  np=1                                     le/crystal_elasticity
 93/96       pass    0.47s  np=1                                        le/2D_manufactured
 94/96       pass    0.39s  np=1                               le/3d_uniaxial_tension_cubit
 95/96       pass    0.38s  np=1                                       navierstokes/channel
 96/96       pass    0.33s  np=1                                            burgers/1D_bump
------------------------------------------------------------------------------------------
 Pass: 96    Fail: 0    Skipped: 0    Total: 96

Total Runtime:      54.82s
Sun Jul 11 08:30:00 2021
```

Most of these regression tests run in under 1 second. Each of them is designed to cover a certain set of capabilities in the code.

The python script, runtests.py, was originally created for DGM, another software package from Sandia. It has a few options for testing a subset of the tests based on various flags. To see all of these flags, type

```
>> python runtests.py --help
```

For example, each test has a set of keywords that indicate the coverage of the test. To simply print these keywords, type

```
>> python runtests.py --list-keywords
```

to run the tests and print the keywords, type

```
>> python runtests.py --print-keywords
```

and to only run the tests associated with a specific keywords, e.g., HDIV, type

```
>> python runtests.py -k HDIV
```

It is also possible to run only the tests in a subdirectory, e.g., ODE, by typing

```
>> cd ODE
>> python ../runtests.py
```

If you add a feature that is not covered by one of the regression tests, then it is strongly recommended that you add a test for it. Each test folder contains at least three files

```
>> cd ODE/BWE
>> ls
input.yaml       mrhyde.gold       mrhyde.tst
```

which are the standard MrHyDE input file, the gold standard output file and the python script defining the test respectively. The runtests.py script looks for mrhyde.tst file in each subdirectory and automatically adds the test to the regression suite. Most of the tests perform a diff on mrhyde.gold and the output of the test, which is usually stored in mrhyde.log. Some tests only check relative or absolute tolerances of certain quantities, and a few only check for code completion.

Note that, despite what the documentation in runtests.py indicates, the number of MPI processes is determined by each mrhyde.tst script and it not controlled by a user input option.

Unfortunately, a new "feature" was introduced in the Mac Monterey OS where it requests permission for each executable to access the internet. For some applications, this can be set once and the response is recorded. For MrHyDE, this is not the case, and each invocation of the executable makes this user request per thread. This significantly slows down the regression testing and will hopefully be fixed in future versions of the OS.

3. SOFTWARE DESIGN AND INFRASTRUCTURE

MrHyDE grew out of the MILO software framework where the intention was to provide a relatively lightweight and simplistic C++ interface to various packages in Trilinos and to enable large-scale PDE constrained optimization through ROL for multiphysics applications. MrHyDE is the second generation version of this framework that maintains this simple and lightweight design, incorporates a robust and efficient concurrent multiscale framework and leverages Kokkos for performance portability to wide variety of modern heterogeneous computational architectures.

An illustration of the organization of MrHyDE is provided in Figure 3. At the highest level, MrHyDE is controlled by a driver script

```
MrHyDE/src/driver.cpp
```

that contains the C++ "main" function. The driver script simply sets up the other high-level objects, namely the managers and interfaces, and runs the desired analysis mode. The driver also contains the print statement:

```
>> ./mrhyde --version

MrHyDE - A framework for Multi-resolution Hybridized Differential Equations -- Version 1.0

Copyright 2018 National Technology & Engineering Solutions of Sandia, LLC (NTESS).
Under the terms of Contract DE-NA0003525 with NTESS, the U.S. Government retains certain rights in this software.

Questions? Contact Tim Wildey (tmwilde@sandia.gov) and/or Bart van Bloemen Waanders (bartv@sandia.gov)
```

and controls the printing of the Teuchos timers and MrHyDE profile data.

Each of the interfaces provides a connection between MrHyDE and a specific package (or set of packages) in Trilinos. The role of the managers is to use the interfaces and certain other classes to perform a certain task, such as assembly. The managers are unique to MrHyDE and tend to contain highly optimized and specialized functionality to perform these tasks. Helping the managers perform these tasks are the tools. These include the worksets, the volumetric and boundary element groups, a specialized data import class, various unique types of basis functions and the classes required for the specialized directed acyclic graph for assembly and function evaluation.

One unique feature in the MrHyDE is the total absence of the Model Evaluator paradigm which form a key component of many of Trilinos-based frameworks. This was by design with the goal of providing enhanced flexibility and transparency in the software framework. Of course, such decisions have consequences. In particular, MrHyDE does not use many of the packages in Trilinos that require Model Evaluators for full functionality. Some of this functionality, such as nonlinear solvers and time integrators, have specialized implementations in MrHyDE.

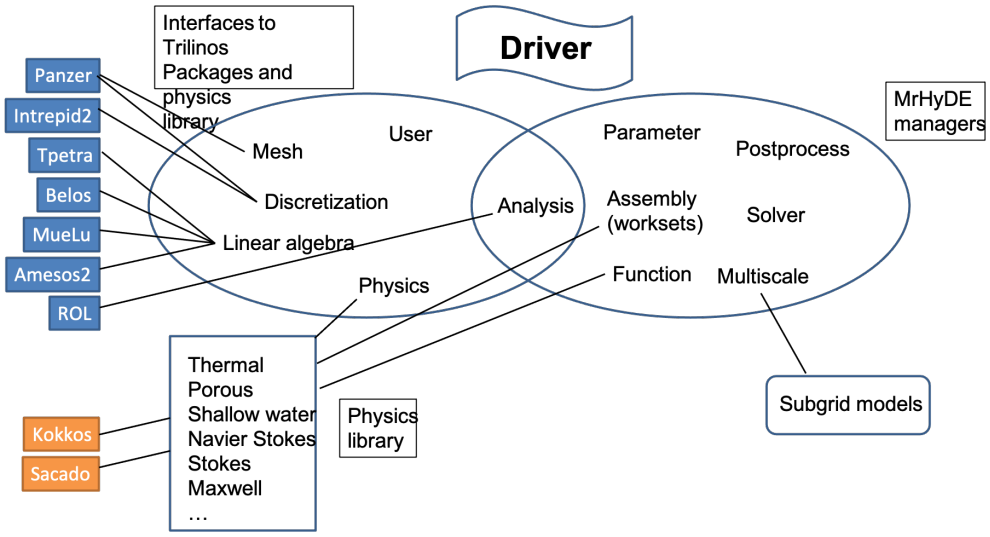

Figure 3-1. Illustration of the organization and dependencies in MrHyDE.

In the following (sub)sections, we describe the basic purpose of each of the interfaces, managers and tools in MrHyDE.

3.1. Interfaces

User The user interface connects the user to MrHyDE. It reads in the input yaml (default) or xml file and makes sure all of the required blocks are included.

Discretization Provides the interface to all of the Intrepid2 and Shards functionality. Also the interface to the PanzerDOF manager.

Linear Algebra Provides an interface to the Tpetra packages for (mult)-vectors and sparse matrices. Also provides the interface to the linear solver, e.g., Belos for iterative and Amesos2 for direct. Finally, it also provides the interface to various preconditioners available in MueLu and Ifpack2.

Mesh Provides the interface to the Panzer inline meshes and the STK interface for Exodus meshes. The mesh interface also adds all of the nodal and cell-centered fields to the mesh for visualization. It requires the physics interface for lists of these fields. The postprocess manager fills in the fields at each time step.

Physics Provides an interface the physics module library in MrHyDE. This is the only interface or manager that communicates with these physics modules.

Compadre As of right now, this is stored under the tools directory, but it is really an interface to the Compadre library which provides kd-tree point-cloud searches which are used to import user-defined data, e.g., permeabilities and microstructures, into the cells.

FFT Provides an interface to the FFTW library. This capability is currently very limited and tailored for a specific application.

3.1.1. User Interface

A clear and functional user interface is critical for complex software frameworks, like MrHyDE. While MrHyDE does not have a Graphical User Interface (GUI), it does have a fairly simple user interface. The user interacts with MrHyDE through input files, which may xml or yaml, but the latter is the default and preferable. All of the regression tests use yaml input files. For most simulations, there is only one input file with 8 main blocks: Mesh, Physics, Discretization, Solver, Analysis, Functions, Postprocess, Subgrid and Parameters. For some problems, these blocks can be rather large and settings can often be reused between different simulations, so each of these can be contained in separate yaml (or xml) files. A sample input file is:

```
%YAML 1.1
---
ANONYMOUS:
  verbosity: 0
  debug level: 0
  Mesh input file: input_mesh.yaml
  Functions:
    thermal source: 8*(pi*pi)*sin(2*pi*x)*sin(2*pi*y)
  Physics:
    modules: thermal
    build face terms: true
    Dirichlet conditions:
      e:
        all boundaries: '0.0'
    Initial conditions:
      e: '0.0'
  Discretization:
    order:
      e: 1
    quadrature: 2
  Solver:
    solver: steady-state
    workset size: 100
    nonlinear TOL: 1.0e-07
    max nonlinear iters: 2
  Analysis:
    analysis type: forward
  Postprocess:
    compute errors: true
    write solution: false
    True solutions:
      e: sin(2*pi*x)*sin(2*pi*y)
      e face: sin(2*pi*x)*sin(2*pi*y)
      'grad(e)[x]': 2*pi*cos(2*pi*x)*sin(2*pi*y)
      'grad(e)[y]': 2*pi*sin(2*pi*x)*cos(2*pi*y)
...
```

We can see that most of the blocks are defined explicitly (Parameters is optional and not used), but the Mesh input points to another file.

The convention in the input files is for sublist to be capitalized (first word only), and all parameters are lowercase except proper names. Typically, only the "verbosity" and "debug level" parameters, which are used globally throughout the code, appear in the main list. All other parameters are contained in one of the sublists.

The "verbosity" parameter controls how much output gets printed as the simulation is running. This is an integer and the default is 0, which does not print anything. Increasing this value will increase the amount of output. Typically, a value of 10 is sufficient. This will print the progress of the time integrator, the nonlinear solver and the linear solver.

The "debug level" parameter is used to find where errors may be occurring in the code. This is an integer and the default value is 0, which does not print anything. Setting this to 1 will print

statements when the code enters and exits most of the constructors and several other functions that are called only once during the set-up phase. Setting this to 2 will print everything from 1, as well as statements from functions that are called many times during the run phase. Setting this to 3 will print everything from 1 and 2, as well as a few matrices and vectors. This is useful if the linear solves are failing due to NaNs. However, setting the debug level 3 is only recommended on smaller problems since the amount of output is quite substantial.

The main purpose of the user interface class is to make sure each of the required blocks is provided and to combine these into a master Teuchos ParameterList that gets passed around to various constructors to set up the problem. Most of the parameters in MrHyDE can be specified through the input file with the appropriate keyword. A master list of all of the keywords, with their defaults, is contained in the Scripts directory. These master lists are just for reference and should not be used for setting up a simulation. Instead, one should start with the input files from the most relevant regression test.

3.1.2. Discretization Interface

The discretization interface is the only place in MrHyDE that interacts with the Trilinos discretization packages, namely, Shards, Intrepid2 and the PanzerDOF manager. Most of the functionality in the discretization interface is designed to perform basis finite element operations, such as computing basis functions at quadrature points and mapping this information between the reference and physical element. It also constructs the Panzer DOF manager once all of the variables have been defined by the physics manager.

3.1.3. Linear Algebra Interface

The linear algebra interface is (supposed to be) the only place in MrHyDE that interacts with the Trilinos linear algebra packages, namely, Tpetra, Belos, Muelu and Amesos. There are a few exceptions where linear algebra operations occur in other places, but these will eventually be moved into the linear algebra interface.

3.1.4. Mesh Interface

The mesh interface works with the functionality in the panzer STK interface to create an STK mesh database with the appropriate fields added to the mesh for visualization. Note that if visualization is not required, then most of the larger objects in the mesh interface are deallocated before MrHyDE starts running a simulation (time stepping))to maximize the available memory for the simulation.

3.1.5. Physics Interface

The physics interface is unique in that it does not connect MrHyDE with another package in Trilinos. Instead, it connects MrHyDE with the physics modules defined by the user. Most of this is fully automated, so the user only needs to modify the physics importer class and their own physics module.

3.1.6. Compadre Interface

The compadre interface is useful if one needs to use a kd-tree to locate arbitrary point clouds within a mesh.

3.1.7. FFTW Interface

The FFTW interface is only used by the Mirage applications.

3.2. Managers

Analysis Provides the high-level functionality to actually run the simulation(s).

Assembly Primary purpose is to turn a Tpetra multi-vector, representing the current state solution, into residuals and Jacobians. This includes gather operations, local residual/Jacobian calculations, fixing constraints (e.g., Dirichlet conditions) and scatter operations.

Function Manages all of the evaluation of the physics-defined or user-defined functions.

Multiscale Creates the subgrid models and determines which subgrid model each coarse scale element group uses.

Parameter Sets up and updates all of the active, inactive, discretized and stochastic parameters.

Postprocess Provides all of the functionality for visualization and computing responses, objective functions and integrated quantities.

Solver Provides the functionality for forward and adjoint solves. Includes transient and nonlinear solvers as well as the linear algebra interface, which contains the linear solvers and preconditioners.

UQ Embedded capability for generating samples for UQ studies. Also performs some basic statistical analysis.

3.2.1. Analysis Manager

The analysis manager is the second highest-level part of MrHyDE (behind only the driver) and controls the precise operating mode that the user has requested. Currently available operating modes are described in Section 4.4

3.2.2. Assembly Manager

This chapter describes the assembly manager and how it interacts with other pieces of the code. Before getting into the details, we note that Panzer has several tools for efficient assembly of residuals and Jacobians for coupled multi-physics systems. However, due to the fact that the hybridized and multiscale methods, described in Chapter 6, require different assembly strategies than standard finite element models, MrHyDE has a custom assembly procedure that is optimized for such problems.

In principle, finite element assembly is a simple procedure that takes the current state vector, $u \in \mathbb{R}^n$, and maps it to residual, $r \in \mathbb{R}^n$ and a Jacobian, $J \in \mathbb{R}^{n \times n}$. Getting from u to r and J typically involves three steps: gather, evaluation and scatter. The gather step breaks u into local vectors for each element. The evaluation step maps these values to quadrature points, computes any other functions at quadrature the quadrature points, computes the residual at these quadrature points, and integrates against test functions. This results in a local contribution to the residual and Jacobian. The scatter step then maps these local contributions to the global residual and Jacobian. In practice, the gather step is relatively cheap (fast) and easy to parallelize over the elements. The evaluation step is the most expensive due to the number of FLOPs involved. However, it is also easy to parallelize these computations over the elements. The scatter step is slightly more expensive than the gather step because multiple elements may contribute to a given row in the vector/matrix. This slightly increases the number of FLOPs and it demands specialized procedures if one want to parallelize over the elements due to contention[1]. Avoiding contention is quite easy. One can either color the elements so that no two elements being processed concurrently share any degrees of freedom, or one can use atomic operations. MrHyDE primarily uses atomics, but coloring is an option for future work.

As previously mentioned, the primary mechanism for exposing parallelism in the assembly procedure is processing multiple elements at the same time. In MrHyDE, this is the motivation for the cells, which contain the local information for a collection of elements, and the worksets, which store information for collections of elements to be used by the function manager or the physics modules. MrHyDE uses Kokkos::Views to store data that will be processed in parallel on a CPU (using OpenMP), on a GPU or an any other architecture.

In Algorithm 1, the last step is applying constraints to the degrees of freedom. This is necessary if one is using strongly enforced Dirichlet boundary conditions or point constraints to handle a null space in the operator. While these are virtually the same operations, and they are handled similarly in MrHyDE is the sense that these degrees of freedom are not removed from the linear system. However, the algorithms for handling each of these constraints are slightly different. For

[1] Contention refers to when more than one process is trying to manipulate data.

Algorithm 1: High-level overview of the assembly algorithm in MrHyDE

Input: Current state vector, u
Perform gather to each cell;
for $i = 1, \ldots, N_{cells}$ **do**
    if *assemble volumetric terms* **then**
        Update the workset with the current cell volumetric data;
        Compute the volumetric residual;
    end
    if *assemble face terms* **then**
        for $j = 1, \ldots, N_{faces}$ **do**
            Update the workset with the current cell face data;
            Compute the face residual;
        end
    end
    Perform scatter to global residual and Jacobian;
end
if *assemble boundary terms* **then**
    for $i = 1, \ldots, N_{Bcells}$ **do**
        Update the workset with the current boundary cell data;
        Compute the boundary residual;
    end
    Perform scatter to global residual and Jacobian;
end
Apply constraints, e.g., Dirichlet boundary conditions;
Output: Residual, r, and Jacobian, J

the Dirichlet constraints, MrHyDE keeps a list (stored as a 1-dimensional Kokkos::View on the AssemblyDevice) of all of the Dirichlet degrees of freedom. When performing the scatter operation, the row associated with each of these degrees of freedom is skipped, giving a row of all zeros. The corresponding row in the residual is also zero since the initial guess satisfies the Dirichlet condition. The final step in Algorithm 1 then simply places ones on the diagonal for these degrees of freedom. This avoids assembling into the row on the matrix and then having to zero it out. On the other hand, point constraints are far less common and a simulation typically only has a few point constraints at a time. Thus, these rows are assembled through the usual procedure and zeroed out in the last step.

Finally, we comment on how the cells (and boundary cells), worksets, function manager and physics modules interact to execute the assembly algorithm. The overall goal is to evaluate the residual (and simultaneously the Jacobian since we use automatic differentiation) at the quadrature points and the integrate this against the test functions for the discretization. To achieve this, we require the solution and perhaps many other complicated functions at these quadrature points. For a given collection of elements, the local basis and integration information is stored[2] in the cell[3]. The cells passes this information along to the workset, which then computes the current solutions, and any derivatives, at the quadrature points. At this point, the workset is fully updated to have all of the information for the given cell. The function manager and the physics modules can access anything in the workset. The physics modules then call the function manager to evaluate the functions required to compute the residual. These functions can depend on anything stored in the workset, e.g., solutions, derivatives of solutions, quadrature points, time, parameters, discretized parameters, etc. The physics modules get whatever information they require from the workset, e.g., basis functions, offsets, etc. Then, the residual (and Jacobian) can be computed easily in the physics module.

3.2.3. Function Manager

This chapter describes the function manager, the related objects supporting the Directed Acyclic Graphs, and how the function manager interacts with the worksets and physics modules.

At a high-level, the function manager is easy to describe. It takes functions, given as strings, defined by the user, the physics modules or elsewhere in MrHyDE, picks them apart looking for operations and data it knows about, and builds a tree structure to evaluate the function. Each function corresponds to a tree, and trees can belong to one of several forests. In MrHyDE, a forest is a collection of trees that all use the same evaluation points. There are only three forests in MrHyDE, although adding more would be straightforward). These forests are associated with the volumetric quadrature points, the side/face quadratures points, and individual points (for sensors and other pointwise evaluations).

A tree is further decomposed into branches and leafs. A branch corresponds to an operation on some data, e.g., plus, times, sin, exp, etc., and a leaf corresponds to a known quantity, e.g.,

[2] The basis and integration information is usually stored in the cells, but can be recomputed on-demand if memory usage is a concern.

[3] Or boundary cells if computing the contributions to the boundary residual.

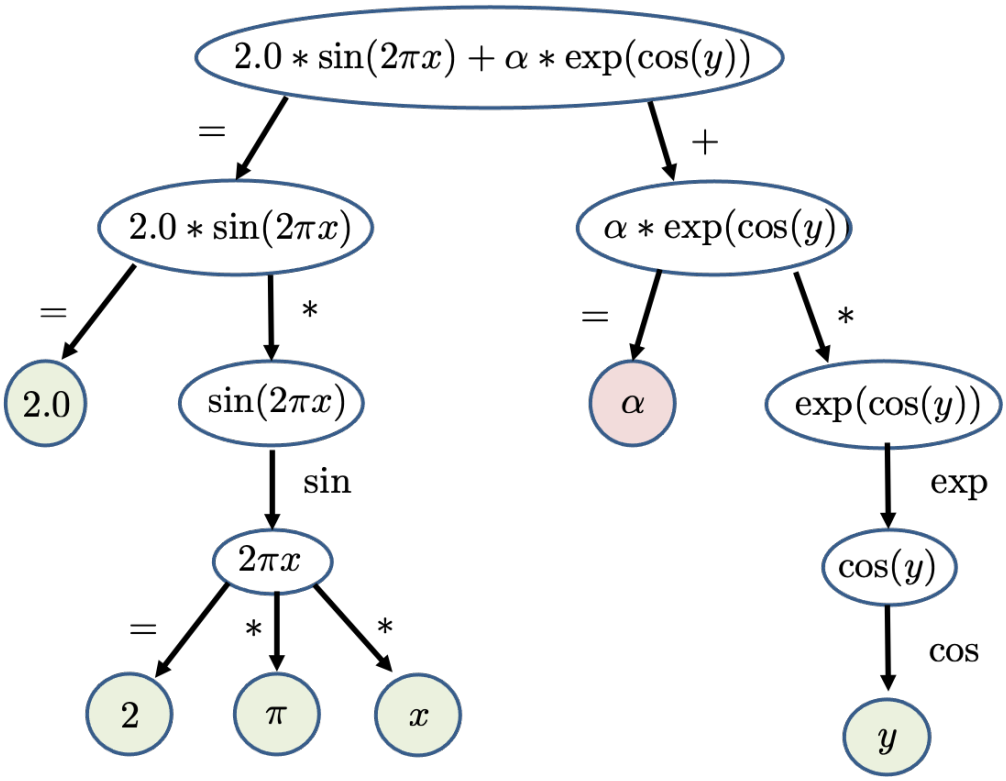

Figure 3-2. An illustration of how the function manager (and the interpreter) decompose functions into trees (graphs) with branches and leafs. The built-in known leafs are in green and a potentially unknown leaf is in red. If α is defined as a function or a parameter, then this will decompose properly.

something stored in the workset, π, etc. A function gets evaluated by working down branches until leafs are found, and then working back up. An example of a decomposed function is shown in Figure 3.2.3. The function gets broken down into simpler components until everything is expressed as a leaf or a function of a leaf. In terms of a graph, the nodes represent data and the edges represent operators. In Figure 3.2.3, the built-in leafs are shaded green, while a potentially unknown leaf is shaded in red. If the user has defined α as another function or as a parameter, then this function will decompose properly. Otherwise, an error will occur.

In the forest/tree/branch/leaf hierarchy, the leafs point to existing data, usually stored in the workset, while the branches require additional containers to store their data. One of the more complicated aspects of this framework is the fact that different types of data can be stored in a branch and data types can be promoted to more complicated ones. For example, in Figure 3.2.3, we see that both "2" and "π" are scalar quantities, so the product could be stored as a scalar. However, "x" is stored as a Kokkos::View of scalars, so the product of all three must be stored as a Kokkos::View of scalars. In fact, everything in the tree is either a scalar or a Kokkos::View of scalars, except perhaps α. If α is an AD object or a Kokkos::View of AD objects, then "$\alpha * \exp(\cos(y))$", and everything above it, needs to be stored as Kokkos::Views of AD objects. In general, a scalar can be promoted to a Kokkos::View of scalars or a Kokkos::View of ADs. A Kokkos::View of scalars can be promoted to a Kokkos::View of ADs.

This complicated procedure of storing different data types and promoting when necessary can greatly reduce both the number of FLOPs in evaluating functions as well as the memory required.

While Figure 3.2.3 is a useful illustration, the syntax is not quite correct to be a function string. Here are a few examples:

```
Functions:
    thermal source: 8*(pi*pi)*sin(2*pi*x)*sin(2*pi*y)
    density: 1.0+2.0*rho
    rho: gamma*exp(-2.0*t)
    gamma: 4.5
    absgradE: abs(grad(e)[x]) + abs(grad(e)[y])
```

Here, we can see that functions can depend on other functions as well as built-in variables, like x, π, t and e (assuming e is the state variable). Here is a summary of the various types of data that the function manager understands:

Constant scalar quantities For example, "2/34" or "pi".

Spatial coordinates Use only "x", "y" or "z". These cannot be used as names of any other variables.

Time Use only "t". This cannot be used as the name of any other variable.

Scalar state variables These can be used in a straightforward manner, e.g., "e".

Vector state variables These need to be specified component-wise using "[ ]", e.g., "B[x]".

Derivatives of state variables For HGRAD variables, the gradient is a vector, so we use "grad(e)[x]". For the divergence of an HDIV variable, we just use "div(u)" since the divergence gives a scalar. For the curl of an HCURL variable, we use "curl(E)[y]".

Scalar parameters Any parameter defined in the Parameters sublist can be used. For scalar parameters, we just use the name, e.g., "p".

Vector parameters For vector parameters, we use the name with the component, e.g., "p[1]".

Discretized parameters These are treated identical to state variables, including derivatives of the parameters.

Auxiliary variables These are also treated identical to state variables. These have not been tested much at this time.

Functions Any function can be used as a variable, e.g., "gamma" as above.

The function manager also understands a number of standard mathematical operations:

```
+, -, *, /, ^, sin, cos, tan, exp, log, abs, <, > , sqrt
```

The less than and greater than symbols are useful in defining logical within functions, e.g.,

```
Functions:
    circle source: 1.0*(r^2<1.0)
    r: sqrt(x^2+y^2)
```

defines a function that is nonzero only within a circle of radius 1.0 centered at the origin.

3.2.4. Multiscale Manager

The multiscale manager controls the usage of subgrid models and the transfer of information from the coarse scale problem to the fine scale problem and back. This process is formally referred to as the *macro-micro-macro* map. The multiscale framework is described in more detail in Chapter 5

3.2.5. Parameter Manager

The parameter manager's sole responsibility is to deal with the various types of parameters available in MrHyDE. These types are:

Active These are scalar or vector parameters that are eligible for sensitivities, i.e., adjoint-based gradients.

Inactive These are scalar or vector parameters that are not eligible for sensitivities. These are never updated from their initial values.

Stochastic These are scalar or vector parameters that are not eligible for sensitivities, but are randomly distributed according to a user-defined distribution.

Discretized These are parameters that represent discretized fields. Other codes often call these distributed parameters. In MrHyDE, these parameters can use any of the available discretization although only HVOL and HGRAD are regularly tested. The parameter manager sets up a Panzer DOF manager for these discretized parameters, the linear algebra interface is able to set up up linear algebra objects, i.e., vectors and matrices, associated with them, and sensitivities are available through the postprocess manager.

The user can define any of these parameters from within the Parameters sublist in the input file. To define an inactive parameter, we set:

```
Parameters:
    magma:
      type: scalar
      value: 1.0
      usage: inactive
```

To define an active parameter, we set:

```
Parameters:
    magma:
      type: scalar
      value: 1.0
      usage: active
```

To define two stochastic parameters, one uniform and one Gaussian, we set:

```
Parameters:
    a:
      type: scalar
      value: 1.0 # not actually used
      usage: stochastic
      distribution: uniform
      min: 1.0
      max: 2.0
    b:
      type: scalar
      value: 0.0 # not actually used
      usage: stochastic
      distribution: Gaussian
      mean: 0.0
      variance: 1.0
```

To define a discretized parameter, we set:

```
Parameters:
    magma:
      type: HGRAD
      order: 1
      usage: discretized
      initial value: 1.0
```

The user can define an arbitrary number of parameters and each of these can be used within functions or anything the function manager can interpret. See Section 3.2.3 for details.

3.2.6. Solver Manager

The solver manager controls the algorithms for time-stepping, solving nonlinear systems and solving linear systems. All of these capabilities are described in detail in Section 4.3.

3.2.7. UQ Manager

The UQ manager is an internal (to MrHyDE) capability that generates samples based on the distributions given by the user and computes some basic statistics on the quantities of interest computed at each of these sample points. The UQ manager also contains the basic functionality for DCI. In general, these capabilities are quite limited and should only be used if the setup time for MrHyDE is significant compared to the runtime. Otherwise, an external UQ package, such as Dakota, should be used instead.

3.3. Tools

Workset Stores the current state values, discretized parameter values, integration points/weights and the current basis functions at the current integration points. The groups store the current geometric data or basis functions and the current gathered state solution (as a Kokkos View) and the workset maps this to state solutions at the integration points. The physics modules and the function manager then utilize these values in computing residuals, responses, objectives, etc.

Group A group is simply a collection of elements that are to be processed together. On-node parallelism is enabled over these elements using Kokkos. Thus, the number of elements in a group dictates the first level of available parallelism. Many functions utilize hierarchical parallelism, which exposes additional dimensions to parallel, or team, calculations. Groups store all of the volumetric and face integration information and the basis functions evaluated at these points. If memory is a concern, there is an option to turn off a proportion of the storage of these basis functions and to recompute them as needed.

Boundary Group A boundary group is a collection of elements that all have an edge/face on a given boundary segment. These do not store any volumetric or face information, just boundary integration points/weights and basis functions evaluated at these points for the edge/face that lies on the given boundary. Otherwise, they serve a very similar purpose to the regular groups. However, while elements only belong to one group, an element may belong to multiple boundary groups if it has multiple edges/faces on different boundaries.

Group MetaData A lightweight class that stores some meta-data associated with the elements/groups. Each element block in the mesh has one group meta-data object and this information is shared by all groups and boundary groups.

DAG Contains the building blocks (forests, trees, branches and leafs) of the custom Directed Acyclic Graph which is used by the function manager to evaluate complicated user-defined functions.

Data A custom class for reading in data from files and extracting information as requested by the simulation.

Interpreter Takes strings representing expressions to evaluate and decomposed them into DAGs in terms of known data, e.g., state solutions, parameters, etc.

KokkosTools A basic debug tool for printing data in Kokkos Views or Tpetra objects.

Vista A lightweight wrapper for the output of the function evaluations that simply wraps the "operator()" for Kokkos Views, but also has the ability to return scalar values. This allows the trees (functions) in the function manager to only store the minimum amount of data necessary.

Compressed View A lightweight wrapper to Kokkos Views that allows for compression of the data stored in the view with automatic extraction. The only current use case is database compression where the data extraction is enabled by a pointer (view) to a database of basis functions and a vector containing the appropriate index into the database for each element.

Solution Storage A general class for storage Tpetra multivectors at various times. If the code attempts to store a vector at a time where data is already stored, it will automatically over-write the existing data. This capability is really only used when solving an adjoint problem and the forward solution needs to be available at each time step. All postprocessing extracts information as needed as the time integration progresses.

4. BASIC CAPABILITIES

4.1. Discretizations

MrHyDE has full support for many of the HGRAD, HVOL, HDIV and HCURL basis functions available in Intrepid2. It also has some support for custom basis functions, namely the Arbogast-Correa HDIV space and HFACE basis functions which are defined only on the skeleton of the mesh and are used for hybridized methods. A complete list of the supported basis functions can be found in the discretization interface

```
// HGRAD basis functions
#include "Intrepid2_HGRAD_QUAD_C1_FEM.hpp"
#include "Intrepid2_HGRAD_QUAD_C2_FEM.hpp"
#include "Intrepid2_HGRAD_QUAD_Cn_FEM.hpp"
#include "Intrepid2_HGRAD_HEX_C1_FEM.hpp"
#include "Intrepid2_HGRAD_HEX_C2_FEM.hpp"
#include "Intrepid2_HGRAD_HEX_Cn_FEM.hpp"
#include "Intrepid2_HGRAD_TRI_C1_FEM.hpp"
#include "Intrepid2_HGRAD_TRI_C2_FEM.hpp"
#include "Intrepid2_HGRAD_TRI_Cn_FEM.hpp"
#include "Intrepid2_HGRAD_TET_C1_FEM.hpp"
#include "Intrepid2_HGRAD_TET_C2_FEM.hpp"
#include "Intrepid2_HGRAD_TET_Cn_FEM.hpp"
#include "Intrepid2_HGRAD_LINE_C1_FEM.hpp"
#include "Intrepid2_HGRAD_LINE_Cn_FEM.hpp"
#include "Intrepid2_HVOL_C0_FEM.hpp"

// HDIV basis functions
#include "Intrepid2_HDIV_QUAD_I1_FEM.hpp"
#include "Intrepid2_HDIV_QUAD_In_FEM.hpp"
#include "Intrepid2_HDIV_HEX_I1_FEM.hpp"
#include "Intrepid2_HDIV_HEX_In_FEM.hpp"
#include "Intrepid2_HDIV_TRI_I1_FEM.hpp"
#include "Intrepid2_HDIV_TRI_In_FEM.hpp"
#include "Intrepid2_HDIV_TET_I1_FEM.hpp"
#include "Intrepid2_HDIV_TET_In_FEM.hpp"

// HDIV Arbogast-Correa basis functions
#include "Intrepid2_HDIV_AC_QUAD_I1_FEM.hpp"

// HCURL basis functions
#include "Intrepid2_HCURL_QUAD_I1_FEM.hpp"
#include "Intrepid2_HCURL_QUAD_In_FEM.hpp"
#include "Intrepid2_HCURL_HEX_I1_FEM.hpp"
#include "Intrepid2_HCURL_HEX_In_FEM.hpp"
#include "Intrepid2_HCURL_TRI_I1_FEM.hpp"
#include "Intrepid2_HCURL_TRI_In_FEM.hpp"
#include "Intrepid2_HCURL_TET_I1_FEM.hpp"
#include "Intrepid2_HCURL_TET_In_FEM.hpp"

// HFACE (experimental) basis functions
#include "Intrepid2_HFACE_QUAD_In_FEM.hpp"
#include "Intrepid2_HFACE_TRI_In_FEM.hpp"
#include "Intrepid2_HFACE_HEX_In_FEM.hpp"
#include "Intrepid2_HFACE_TET_In_FEM.hpp"
```

Some of these basis functions are specific to first or second order basis functions, but many are arbitrary order and the user can control this order from the input file:

```
Discretization:
    order:
      e: 1
    quadrature: 2
```

In MrHyDE, each variable can have a different basis order (not all combinations are stable for some problems), but the quadrature order is shared amongst all variables and must be chosen to meet the accuracy of the highest order basis. If the quadrature order is not provided in the input file, then it is automatically taken to be the twice the maximum basis order. The same conventions apply for side/face quadrature. If multiple variables use the same basis, then the basis information is only computed and stored once, and re-used for each variable.

4.2. Physics Modules

MrHyDE is designed to allow for rapid development of physics modules with automatic coupling to any existing physics module. This is achieved by having a lightweight and simple format for the physics modules that derives from a common base class. The physics modules are also isolated from the rest of code. Only the physics interface interacts with the physics modules, and the modules themselves only interact with the workset and the function manager.

In general, weak formulations of partial differential equations can have three different types of contributions: volumetric, boundary and face. In MrHyDE, we consider general equations of the form

$$V(u,v) + B(u,v) + F(u,v) = 0,$$

where u is the solution, v is the test function, and V, B and F represent the volumetric, boundary and face contributions respectively. As a concrete example, consider a Poisson equation on some domain $\Omega \subset \mathbb{R}^d$ with mixed (Robin) boundary conditions:

$$-\nabla \cdot (\mathbf{K}\nabla u) = f, \quad x \in \Omega \tag{4.1}$$

$$\mathbf{K}\nabla u \cdot \mathbf{n} + \alpha u = g, \quad x \in \partial\Omega, \tag{4.2}$$

The weak formulation seeks $u \in V = H^1(\Omega)$ such that,

$$\underbrace{\int_\Omega (\mathbf{K}\nabla u \cdot \nabla v - fv)\ dx}_{V(u,v)} + \underbrace{\int_{\partial\Omega} (\alpha u - g)\, v\ ds}_{B(u,v)} = 0, \tag{4.3}$$

for all $v \in V$. While this example is linear, any of the contributions can be nonlinear in u. Thanks to the utilization of automatic differentiation, the Jacobian is automatically computed from the residual, regardless of the complexity in residuals.

Several physics modules are included in the repository and are tested nightly. These include, but are not limited to:

Burgers An implementation of Burger's equation in 1D, 2D or 3D. The strong form equation is:

$$\frac{\partial u}{\partial t} + \nabla \cdot \left(\frac{1}{2} \mathbf{v} u^2 - \epsilon \nabla u \right) = s(x,t),$$

where ϵ is a diffusion parameter (may be zero), $\mathbf{v}$ is the user-defined vector dictating the direction of the flow and $s(x,t)$ is the source term. All of these parameters can be defined by the user from the input file. The solution field, called "u", is an HGRAD variable. This module does not have boundary contributions to the residual or flux functions at this time.

Convection-diffusion-reaction A standard scalar convection-diffusion equation with a reaction term that can be arbitrarily nonlinear. The strong form equation is

$$\frac{\partial c}{\partial t} + \mathbf{v} \cdot \nabla c + r(c) - \frac{1}{\rho c_p} \nabla \cdot (D \nabla c) = s(x,t),$$

where $\mathbf{v}$ is the user-defined velocity field (or velocity from Stokes or Navier Stokes), $r(c)$ is the reaction term, ρ is the density, c_p is the specific heat, D is the diffusion tensor and $s(x,t)$ is the source term. The solution field, called "c", is an HGRAD variable. This module once had boundary residuals, flux functions and SUPG stabilization, but none of these capabilities have been maintained in the most recent version.

Helmholtz This module solves a Helmholtz equation for both real and imaginary components, which are approximated using HGRAD basis functions. The module includes both volumetric and boundary contributions to the residual.

Linear elasticity Steady-state displacement-based formulation of linear elastic, or crystal elastic module, with a few options for the constitutive representation of the stress tensor as a function of the displacements. This module has volumetric and boundary contributions as well as a DG-type flux function for multiscale formulations. The solution fields are the displacements, "dx", "dy" and "dz", all of which are HGRAD variables. In general, the stress is computed as

$$\sigma(\mathbf{d}) = \mathbf{C} : \epsilon(\mathbf{d}),$$

where $\mathbf{C}$ is a fourth-order Cauchy stress tensor and $\epsilon(\mathbf{d}) = \frac{1}{2} \left(\nabla \mathbf{d} + (\nabla \mathbf{d})^T \right)$. The default formulation uses a linear isotropic formulation which reduces to

$$\sigma(\mathbf{d}) = \mu \epsilon(\mathbf{d}) + \lambda \left(\nabla \cdot \mathbf{d} \right) \mathbb{I},$$

where λ and μ are the standard Lame parameters. The crystal elastic module can incorporate more general formulations. The module can incorporate cubic symmetry in the Cauchy stress tensor, where up to three components of the Cauchy stress are independent: C_{11}, C_{44} and C_{12}. Six other components of the tensor are computed from these three: $C_{22} = C_{33} = C_{11}$, $C_{55} = C_{66} = C_{44}$ and $C_{13} = C_{23} = C_{12}$. In the case of isotropic media, these three components are related to the Lame parameters via

$$C_{11} = 2\mu + \lambda, \quad C_{44} = 2\mu, \quad C_{12} = \lambda.$$

Within the crystal elastic module, one can also apply random rotations associated with different grains in a microstructure. The rotated Cauchy stress is given by,

$$C^{\text{rot}}_{ijkl} = C_{mnop} R_{lp} R_{ko} R_{jn} R_{im},$$

where **R** is the rotation matrix. The linear elastic module has both volumetric and boundary contributions to the residual and has a DG flux for hybridized and multiscale simulations.

ODE This is a relatively simple module that is mostly used for testing the time integration methods. The solution variable is "q" and is discretized using an HVOL basis. The equation solved is

$$\frac{\partial q}{\partial t} = f(q,t),$$

where $f(q,t)$ is a user defined function. The regression tests use $f(q,t) = -q$ with a nonzero initial condition $q(0) = q_0$, which gives an analytical solution of $q(t) = q_o \exp(-t)$. The same ODE is solved on every element in the mesh. This module does not have any boundary contributions or flux functions.

Other modules include crystal elasticity, Stokes flow, Navier Stokes, Maxwells, several different formulation for single-phase imcompressible flow in porous media, and a nonlinear heat equation.

4.3. Solvers

This chapter provides a brief overview of the linear, nonlinear and transient solver capabilities in MrHyDE. Due to the fact that MrHyDE does not use the ModelEvaluator framework, some of these capabilities are implemented directly in MrHyDE rather than through an interface to the relevant Trilinos packages.

4.3.1. Linear Solvers and Preconditioners

MrHyDE only uses the second generation Trilinos linear algebra libraries, namely, Tpetra, Belos, MueLu, Ifpack2 and Amesos2. The Epetra stack, which includes Aztec00, ML, Ifpack and Amesos, has been deprecated and is no longer available. The linear algebra interface, which is stored within the solver manager, takes care of allocating Tpetra multi-vectors and compressed-row storage (CRS) matrices. Vectors and/or matrices can be created for either state or auxiliary variables, which typically have different Panzer DOF managers and different dimensions.

There are several different types of linear systems that may be required within in a given simulation. Beyond the usual state Jacobian, we may require L^2-projections for the initial conditions or mass matrices, boundary L^2-projections for strongly imposed Dirichlet boundary conditions, as well as all three of these options for auxiliary variables or discretized parameters. Consequently, the linear algebra interface stores solver options for all nine of these combinations. If the user only provides options in the solver sublist, then these settings are used for all linear

systems. However, different solvers can be used for different types of linear systems, e.g., Block CG for L^2-projections and Block Gmres for the state Jacobians, by providing the appropriate sublists in the input file.

The linear algebra interface can use any of the following Belos solver options for any of the aforementioned linear systems: Block Gmres (or Block GMRES), Block CG, BiCGStab (requires right-preconditioning), GCRODR, PCPG, Pseudo Clock CG, Pseudo Block Gmres, Pseudo Block Stochastic CG, Pseudo Block TFQMR, RCG and TFQMR. The default is Block Gmres. This can be modified by setting the appropriate flag in the input file, e.g.,

```
Solver:
    Belos solver: Pseudo Block CG
```

Any of these linear systems could also be solved using a direct solver through the Amesos2 interface. This solver is enabled by setting:

```
Solver:
    use direct solver: true
```

However, this solver should only be used for smaller, preferably single node, problems.

Preconditioning is typically required for all iterative solvers, even the L^2-projections. There are three different options for preconditioning in MrHyDE: algebraic multigrid (AMG), domain decomposition and various iterative methods available through Ifpack2. The default preconditioner is AMG through MueLu with smoothed/uncoupled aggregation and a Chebyshev smoother. Depending on the linear system, much better options may be available. Many of these options can be adjusted from the input file, e.g.,

```
Solver:
    Preconditioner Settings:
      verbosity: none
      'coarse: max size': 20
      max levels: 5
      cycle type: W
      multigrid algorithm: sa
      'sa: use filtered matrix': false
      'sa: damping factor': 1.2
```

4.3.2. Nonlinear Solvers

The default nonlinear solver is a Newton method where the Newton update, Δu is given by

$$J\Delta u = -R(u), \tag{4.4}$$

where J and $R(u)$ are the Jacobian and residual respectively for the current state u. The updated state is computed via: $u = u + \Delta u$.

For some problems, this update step is too aggressive and the iterations may fail to converge or diverge. To help reduce this problem, we can update via: $u = u + \alpha \Delta u$, for some $\alpha \in [0, 1]$. Choosing an optimal α is computationally expensive, so we gradually test decreasing values until we find an α such that the updated u has a smaller residual norm than the previous u. More sophisticated strategies are certainly possible, but are not yet implemented.

If the problem is transient and the nonlinear solve fails to converge within the maximum number of iterations, then the transient solver (described in the next section) will cut the time step size in half and try again. This often improves convergence of the nonlinear solver.

4.3.3. Transient Solvers

4.3.3.1. Overview of capabilities

MrHyDE has some specialized multirate/multiscale time integration requirements and therefore has its own built-in capabilities that do not utilize ModelEvaluators. For single scale multiphysics applications, a wide variety of time integration methods are available. For a transient simulation, MrHyDE requires a mechanism for defining the time derivative, typically a backward difference formula (BDF), and a Butcher tableau. MrHyDE supports up to sixth-order BDF methods and diagonally implicit or explicit Runge Kutta methods of arbitrary order.

In the input file, these are specified within the solver settings block:

```
Solver:
    solver: transient
    transient BDF order: 1
    transient Butcher tableau: BWE
```

These are the specifications for a backward Euler integrator.

Most of the classical time integrators based on Butcher tableau use the first-order BDF formula. For example, the four-stage fourth-order explicit Runge Kutta method uses

```
Solver:
    solver: transient
    transient BDF order: 1
    transient Butcher tableau: RK-4,4
```

Conversely, the classical BDF time integrators use the backward Euler Butcher tableau. For example, the second order BDF integrator uses:

```
Solver:
    solver: transient
    transient BDF order: 2
    transient Butcher tableau: BWE
    transient startup BDF order: 1
    transient startup Butcher tableau: DIRK-1,2
    transient startup steps: 2
```

We also notice that the higher-order BDF integrators require a "start-up" integrator to generate sufficient data for the backwards differences.

While several standard integrators are pre-implemented and can be selected based on their names, one can also define a custom Butcher tableau through the input file:

```
Solver:
    solver: transient
    transient BDF order: 1
    transient Butcher tableau: custom
    transient Butcher A: '0.0, 0.0, 0.0, 0.0; 0.5, 0.0, 0.0, 0.0; 0.0, 0.5, 0.0, 0.0; 0.0, 0.0, 1.0, 0.0'
    transient Butcher b: '0.1666666667, 0.3333333333, 0.3333333333, 0.1666666667'
    transient Butcher c: '0.0, 0.5, 0.5, 1.0'
```

Finally, we note that using the standard transient solver, the explicit methods still require Jacobians and therefore use AD, so the assembly procedure can be expensive. These Jacobians are really just mass matrices, but they are not necessarily diagonal unless mass lumping is used. A much faster, but more limited, approach for explicit time integration is described in Section 4.3.3.3.

4.3.3.2. How do these time integrators actually work?

As previously mentioned, MrHyDE does not use the ModelEvaluator framework and the precise details of the transient solver are relegated to a specific function within the workset. Consequently, most users are not aware of how these methods are implemented. In this section, we give an overview of the theory and the unified framework in MrHyDE that includes both the hierarchy of BDF methods and the family of methods that utilize Butcher tableau.

In this section, we are interested in solving the following system of ordinary differential equations:

$$u' = f(u,t), \quad u(t_0) = u_0. \tag{4.5}$$

To discretize, let Δt denote the step size and set $t_n = t_0 + n\Delta t$.

The BDF methods are linear multistep methods that approximate the time derivative using previous solution states. For the p^{th} order BDF method, we have

$$\sum_{k=0}^{p} \frac{\alpha_k}{\Delta t} u_{n-k} = f(t_n, u_n). \tag{4.6}$$

Using this convention, the BDF coefficients for the 1st through 6th order methods are given by

$$
\begin{aligned}
&\text{BDF1}: \quad [-1, 1] \\
&\text{BDF2}: \quad [1/2, -2, 3/2] \\
&\text{BDF3}: \quad [-1/3, 3/2, -3, 11/6] \\
&\text{BDF4}: \quad [1/4, -4/3, 3, 4, 25/12] \\
&\text{BDF5}: \quad [-1/5, 5/4, -10/3, 5, -5, 137/60] \\
&\text{BDF6}: \quad [1/6, -6/5, 225/60, -20/3, 15/2, -6, 147/60]
\end{aligned}
$$

Note that the first order method corresponds to the backward Euler method. Also note that the BDF methods do not impact the argument to $f(t, u)$. In contrast, the multi-stage methods defined by Butcher tableau, described next, will primarily affect the arguments to f and only impact the time derivative via a stage weight.

A wide variety of multi-stage methods can be concisely described in terms of a Butcher tableau, which are a mnemonic tool for arranging stage information as:

$$
\begin{array}{c|cccc}
c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\
c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\
\hline
 & b_1 & b_2 & \cdots & b_s
\end{array}
=
\begin{array}{c|c}
\mathbf{c} & \mathbf{A} \\
\hline
 & \mathbf{b}
\end{array}
\tag{4.7}
$$

wheres s is the number of stages.

The entries in **c** determine the time for each stage evaluation:

$$t_i = t_n + c_i \Delta t,$$

the entries in **b** determine the final update,

$$u_{n+1} = u_n + \Delta t \sum_{i=i}^{s} b_i k_i, \tag{4.8}$$

where

$$k_i = f(t_i, u_n + \Delta t \sum_{j=1}^{s} a_{ij} k_j). \tag{4.9}$$

MrHyDE can utilize any Butcher tableau up the diagonally implicit methods. Fully implicit schemes can achieve higher-order convergence for the fixed number of stages, but these are computationally challenging to implement due to the full coupling between stages. We note that the Butcher tableau associated with the backward Euler method is

$$
\begin{array}{c|c}
1 & 1 \\
\hline
 & 1
\end{array}
$$

Thus, all of the BDF methods use the backward Euler Butcher tableau.

There are various ways to use either a BDF formula or a Butcher tableau in a large-scale computational model. MrHyDE uses a specific approach that covers both use cases and never requires changes to the PDE or residual equation.

Suppose we are seeking to solve,

$$u' = f(t, u),$$

where we are ignoring any spatial discretization, although this will typically add a mass matrix to the left-hand side.

Now, let z_i denote the i$^{\text{th}}$ stage solution, i.e., at each stage we seek to solve for z_i. Thus, at each stage the nonlinear solver requires an equation in terms of z_i:

$$g(z_i) = f(t_i, m(z_i)),$$

where (4.9) implies

$$g(z_i) = k_i, \quad m(z_i) = u_n + \Delta t \sum_{j=1}^{s} a_{ij} k_j.$$

One choice for z_i is to set $z_i = k_i$, and solve for k_i, but this is inconsistent with standard implicit integrators that solve for u_{n+1}. In MrHyDE we define:

$$z_i = (b_i \Delta t) k_i + u_n,$$

which, combined with (4.8) immediately implies

$$u_{n+1} = u_n + \sum_{i=1}^{s} (z_i - u_n) .$$

Thus, for any single stage method we have $z_i = u_{n+1}$. Also, combined with (4.9) gives

$$m(z_i) = u_n + \sum_{j=1}^{s} \frac{a_{ij}}{b_j} \left(z_j - u_n\right) .$$

and

$$g(z_i) = \frac{z_i - u_n}{b_i \Delta t}.$$

Note that both g and m are linear with respect to z_i for every method implemented in MrHyDE.

For example, for a backward Euler method we have $z_i = u_{n+1}$ (since it is a single-stage method), and the expressions for g and m reduce to

$$g(z_i) = \frac{z_i - u_n}{\Delta t}, \quad m(z_i) = z_i.$$

Finally, we comment on the use of these methods in MrHyDE. Every time integration method requires both a BDF order and a Butcher tableau. The default for both is backward Euler, so to use a higher-order BDF method, one only needs to change the BDF order and the backward Euler Butcher tableau is implicitly used. Thus, the higher-order BDF methods only affect the definition of $g(z_i)$. Conversely, to use a multi-stage method based on a Butcher tableau, one only needs to specify or provide a Butcher tableau and the backward Euler BDF formula is automatically used. While it is tempting to try to use both higher-order BDF methods with high-order multi-stage methods, this can result in a non-convergent method if done naively.

4.3.3.3. Fully Explicit Time Integration

The approach describe in the previous section is sufficiently general to cover a class of implicit linear multi-step methods, and a variety of implicit or explicit multi-stage methods using a Butcher tableau. However, in the case of fully explicit methods, it is much slower than specialized methods. This is primarily due to two reasons:

- Automatic differentiation is used to construct the derivative of the residual expression with respect to the current stage solution. This results in a weighted mass matrix which gets recomputed in each stage.
- The mass matrix is not necessarily diagonal and therefore requires a linear solver, e.g., Block CG, to solve the corresponding linear system.

The explicit solver described in this section alleviates both of these issues and provides a computationally efficient, scalable and performance portable approach for explicit methods.

The explicit solver can be enabled by setting:

```
Solver:
    fully explicit: true
```

If the time integration method is not actually explicit, then errors will probably occur. The explicit solver uses the stored basis functions to create a weighted mass matrix where the constant scalar weights are defined in the physics block for each variable:

```
Physics:
    modules: maxwell
    Mass weights:
      E: 1.0e-11
      B: 1.0
```

The default weight is always 1.0.

If the mass matrix can be lumped, then the following flag should be set:

```
Solver:
    lump mass: true
```

This will instruct the explicit solver and the assembly manager to construct a Tpetra vector corresponding the diagonal of the lumped mass matrix.

In addition, if no other matrices are required, then the following flag should be set:

```
Solver:
    minimize memory: true
```

which will tell the linear algebra interface to skip constructing Tpetra graphs and matrices. This does not impact performance, but can greatly reduce the memory requirements.

Finally, since the explicit solver does not require automatic differentiation, this can be disabled by setting:

```
-D MrHyDE_DISABLE_AD=ON \
```

in the MrHyDE configure script. No changes are required in the Trilinos configure script. This replaces the Sacado SFad objects with ScalarT (double):

```
#ifdef MrHyDE_NO_AD
typedef ScalarT AD;
#else
typedef Sacado::Fad::SFad<ScalarT,maxDerivs> AD;
#endif
```

However, be aware that some functionality e.g., optimization, will not be available without automatic differentiation.

4.4. Analysis Modes

The following modes are available as different analysis types:

dry run A dry run is primarily designed to asses whether an input file is properly defined before running a large application. The verbosity will automatically be set to 10, which will output certain details about the simulation to the screen. No other output will be generated. A successful dry run indicates that the mesh, physics, discretizations, solvers, etc. were all set up properly and the code is ready to run.

forward This is the most common analysis mode and simply solves the given set of PDEs one time and generates any user-requested output.

forward+adjoint This allows the user to solve one forward problem and one adjoint to give the sensitivities of the user-defined responses with respect to the user-defined input parameters. This is useful when interfacing with external optimization libraries, such as Dakota.

ROL This mode performs PDE-constrained optimization using an interface with ROL.

ROL2 There is currently a separate interface to ROL2, but this will eventually become the default. When this occurs, both ROL and ROL2 analysis modes will use the same functionality.

UQ This performs a very simple UQ by sampling over the user-defined stochastic parameters, collecting the responses and optionally computing the mean, variance and other statistics.

DCI Coming soon. The mode performs data-consistent stochastic inversion by constructing a pull-back probability measure over the uncertain input space. The requires performing a forward UQ study, and postprocessing the results. While this second step can easily be done in Matlab or Python, MrHyDE provides some basic functionality to perform Gaussian kernel density estimation and rejection sampling. These are set in the input file under the analysis block:

restart Coming soon. The restart mode is designed to pick up where a previous simulation left off. The previous MrHyDE run needs to have had check-pointing turned on. In priniciple, any simulation can be restarted by providing the desired checkpoint file as a string under the analysis input parameter block. Note that restart files are processor-dependent, but the user should only provide the single processor name, e.g. mrhyde.rst, rather than mrhyde.1.rst. Also note that MrHyDE will not check to make sure the previous simulation used the same settings. It will simply read in the current time, the state vector, the scalar parameters and the discretized parameters.

```
Analysis:
    analysis type: forward
```

Some of the analysis types, namely ROL and UQ, have their own sublists of options. The ROL sublist is quite extensive and is often stored in a separate yaml file.

We note that the adjoint solves are handled implicitly through the MrHyDE infrastructure. No adjoint residuals or additional equations are required since it exclusively uses a discrete adjoint. MrHyDE uses automatic differentiation to compute the derivative of the objective function with respect to the state variables, which forms the right-hand side for the adjoint system, and the adjoint Jacobian is computed using the linearization around the state solution and assembling the transposed Jacobian directly. No transposes are required from the linear algebra objects.

4.5. Postprocessing Capabilities

Extracting meaningful data from a simulation to provide insights to a user is critical if the computational model is to be used for anything meaningful. MrHyDE provides a variety of mechanisms for extracting data from the simulation as it is running, i.e., without requiring the state solution to be stored throughout the simulation. These capabilities are described in the following sections.

4.5.1. Visualization

As previously mentioned, MrHyDE uses the Exodus data format for both mesh import and for data export. The Panzer STK interface is the primary mechanism for writing data to an Exodus file. By default, this output is stored in output.exo, but this file name can be changed by modifying the appropriate option in the "Postprocess" sublist in the input file:

```
Postprocess:
  write solution: true # default is false
  output file: output # .exo gets added automatically
```

The exodus file can contain arbitrary data on the nodes of the mesh, which get linearly interpolated in the volume of the element, or at the cell-centers, which are represented as piecewise constants over the mesh. While nodal interpolations are well-suited for the solutions that use the lowest order HGRAD basis functions, they are less appropriate for most of the other discretizations. For now, MrHyDE only plots the cell average for the solutions that use HDIV, HCURL, HVOL and HFACE basis functions. Exodus and STK have recently added the ability to visualize both edge and face data, but this is not enabled in MrHyDE yet.

If visualization is enabled, then several cell and nodal fields are added automatically, e.g., the state variables, the cell number, the processor number, etc. The one type of variable that is not automatically added is HFACE. These are more expensive to compute, so the user needs to explicitly request this feature using:

```
Postprocess:
  write solution: true # default is false
  write HFACE variables: true
```

While the data that can be visualized at the nodes is quite limited, there is virtually no limit on the data that can be visualized at the cell centers. This is because MrHyDE can evaluate arbitrary functions at the volumetric integration points, through the function manager, and taking the cell-average of these quantities is trivial. To visualize arbitrary functions, one simply needs to add these functions in the "Extra cell fields" sublist within the "Postprocess" sublist:

```
Postprocess:
  write solution: true # default is false
  Extra cell fields:
    ex: sin(pi*x)+2.0*grad(e)[x]
```

In principle, anything that is composed of leafs, or basic building blocks that the function manager is aware of, can be visualized at the cell centers.

The default behavior is to output to the Exodus file after every time step. For some problems, this is too much data, so there is an option to only output every P time steps:

```
Postprocess:
  write solution: true # default is false
  write frequency: 10
```

where here we have set $P = 10$.

4.5.2. Solution Verification

Solution verification is a critical component of V&V and MrHyDE provides the ability to perform solution verification using arbitrary analytical functions. Performing solution verification is enabled by setting:

```
Postprocess:
  compute errors: true # default is false
```

However, this won't actually compute anything. The user also needs to provide true solutions to compare against. MrHyDE automatically selects the solutions and norms based on the user-provided true solutions within the "Postprocess" sublist. For example, providing

```
Postprocess:
    compute errors: true
    True solutions:
      e: sin(2*pi*x)*sin(2*pi*y)
      e face: sin(2*pi*x)*sin(2*pi*y)
      'grad(e)[x]': 2*pi*cos(2*pi*x)*sin(2*pi*y)
      'grad(e)[y]': 2*pi*sin(2*pi*x)*cos(2*pi*y)
```

will tell MrHyDE to compute the L^2 norm of the error:

$$\int_\Omega \left(e - \sin(2\pi x)\sin(2\pi y)\right)^2,$$

the L^2 norm of the error in the gradient,

$$\int_\Omega \left(\frac{\partial e}{\partial x} - 2\pi\cos(2\pi x)\sin(2\pi y)\right)^2 + \int_\Omega \left(\frac{\partial e}{\partial y} - 2\pi\sin(2\pi x)\cos(2\pi y)\right)^2,$$

and the L^2-face norm of the error in the solution

$$\sum_{T\in\mathcal{T}_h}\sum_{\gamma\in\partial T}\int_\gamma \frac{1}{2|\gamma|}\left(e - \sin(2\pi x)\sin(2\pi y)\right)^2,$$

where $\mathcal{T}_h$ is a mesh and $|\gamma|$ is the measure of γ.

Computing the norm of the error in the gradient is enabled by defining true solutions for any of the components of the gradient. Any omitted components will use zero for the true solution.

For vector variables, we use rectangular brackets to denote specific components. For example, if B is a vector variable, one can compute the vector L^2 norm by defining true solutions for B[x], B[y], B[z]. Similarly, the error for the divergence of B or the curl of B can be enabled by defining true solutions for div(B), or any of curl(B)[x], curl(B)[y] or curl(B)[z].

4.5.3. Responses

4.5.3.1. Integrated Responses

In the current version of MrHyDE, integrated responses are handled through the objective functions that one would utilize for optimal design or control. This capability is described in detail in Chapter 8, but if the user is separately interested in the value of the integrated response over time, they can add a flag to save the response data as indicated below:

```
Postprocess:
    compute objective: true
    compute sensitivities: false
    Objective functions:
      obj0:
        type: integrated control
        function: '1.0*(e-targ)^2'
        save response data: true
        weight: 0.0625
```

4.5.3.2. Sensor Responses

Sensors are simply pointwise evaluations of user-defined functions, objective functions or solution fields. It is up to the user to use caution in the proper use of pointwise quantities, as the solution may not possess sufficient regularity for such quantities to be well-defined and converge properly as the mesh is refined.

There are three ways to generate the physical locations for sensors:

- **Exodus Import:** If the Exodus file contains sensors, then these can be imported automatically. This is rarely used and may not be tested very well.
- **Grid:** If the sensors can be defined on a uniform grid, then the grid bounds and spacings are sufficient to specify the sensor locations.
- **Text File:** For arbitrary point clouds, use the text input file format. The file should contain only the physical locations of each sensor. Format is $N \times d$, where N is the number of points and d is the spatial dimension.

5. CONCURRENT MULTISCALE METHODS

MrHyDE is designed to use heterogeneous computational architectures to enable concurrent multiscale modeling and simulation. The general form of a coupled multiscale system is

$$\begin{aligned} \frac{\partial u_c}{\partial t} + F_c(u_c, u_f) = 0, \\ \frac{\partial u_f}{\partial t} + F_f(u_c, u_f) = 0, \end{aligned} \tag{5.1}$$

where u_c denotes the coarse-scale variables and u_f denotes the fine scale variables. Either F_c or F_f may also depend on space and/or time, but we omit this explicit dependence for simplicity. If we assume that the fine scale variables can be eliminated (preferably locally), then we can solve an equation only involving the coarse scale variables

$$\frac{\partial u_c}{\partial t} + F_c(u_c, h(u_c)) = 0, \tag{5.2}$$

where the notation $h(u_c)$ represents the local nonlinear elimination of the fine scale information, which depends on the current coarse scale state. As we will soon see, this is related to the Schur complement in the case of steady-state linear operators.

At this point, there are two possible interpretations of the coupled system (5.1) and the reduced system (5.2). If the objective is to compute an approximation to u_c that is informed by the fine scale information, then one interpretation is that this can be obtained by solving (5.2). However, if the objective is to approximate the fine scale solution that is informed by global information, then we can interpret (5.2) as providing the time evolution of the coarse scale global coupling between fine scale models. The appropriate interpretation is problem dependent.

For simplicity, we will ignore any spatial discretizations throughout this section and proceed as if we are solving coupled systems of ODEs, but in general, all of the use cases involve spatial discretization and are typically based on variational formulations.

To simplify the notation, we will rewrite (5.2) as

$$g(u_c) + F_c(u_c, h(u_c)) = 0, \tag{5.3}$$

where clearly $g(u_c)$ represents the time derivative.

The precise form of $h(u_c)$ depends on the multiscale formulation, e.g., variational multiscale, homogenization, FE^2, etc. In some cases, it may simply be a numerical approximation of u_f given u_c, and in other cases it may be a model for the subgrid effects. However, in all cases we refer to h as the Macro-Micro-Macro map, since macro-scale information is sent to the local fine scale (subgrid) models, fine scale information is computed (or approximated analytically), and a

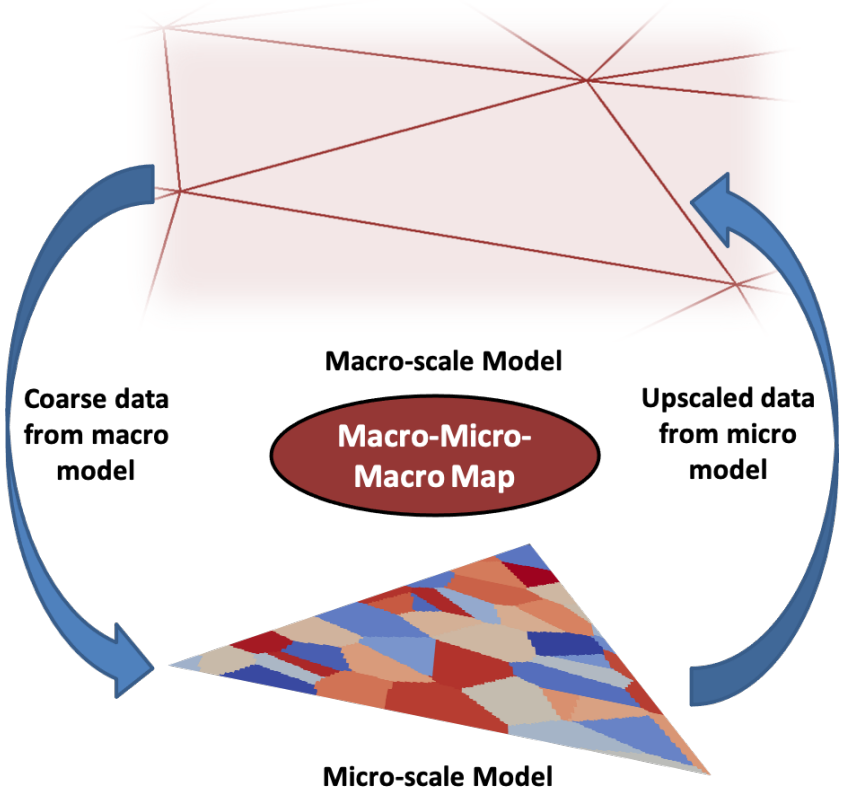

Figure 5-1. Illustration of the Macro-Micro-Macro map.

reduced (upscaled) version of this information gets returned to the coarse scale problem (see Figure 5-1).

MrHyDE is designed to utilize any type of subgrid model, but the primary demonstration is a multiscale Dirichlet-to-Neumann map which is described in Chapter 6. MrHyDE also requires the gradient of h with respect to u_c to enable accurate Jacobians for implicit time integration. This is often complicated since h will can depend on u_c either explicitly or implicitly through u_f, which depends on u_c. Thus, the subgrid models typically compute

$$\frac{\partial h}{\partial u_c} = \frac{\partial h}{\partial u_c} + \frac{\partial h}{\partial u_f}\frac{\partial u_f}{\partial u_c}.$$

Computing $\frac{\partial u_f}{\partial u_c}$ is the most challenging step. For the sake of simplicity, we first describe the steady-state case before moving on to the transient case.

5.1. The Steady-State Case

In this section, we focus on solving the following steady-state multiscale system,

$$F_c(u_c, u_f) = 0, \tag{5.4}$$
$$F_f(u_c, u_f) = 0, \tag{5.5}$$

which is a clear simplification of (5.1). Recall that we assume that the fine scale equation can be localized, so given the current approximation for u_c, we can solve

$$F_f(u_c, u_f) = 0$$

for u_f. Note that this does assume that the fine scale solution can be uniquely determined given any coarse scale data, but this is a fairly reasonable assumption. The goal is then to solve the nonlinear coarse scale equation,

$$F_c(u_c, h(u_c)) = 0,$$

for u_c. THe default procedure in MrHyDE is to use Newton's method, which, given an approximation $u_c^{(k)}$, solves

$$J_c^{(k)} \Delta_c = -F_c\left(u_c^{(k)}, h(u_c^{(k)})\right),$$

where

$$J_c^{(k)} = \frac{\partial F_c}{\partial u_c}\Big|_{u_c^{(k)}},$$

and we set $u_c^{(k+1)} = u_c^{(k)} + \Delta_c$. The iterations continue until a user-prescribed convergence tolerance, typically a relative tolerance on the norm of residual, is met. For the sake of simplicity, we will drop the iteration index on u_c in the remainder of this section.

In order to evaluate the residual, we need to be able to evaluate the Macro-Micro-Macro map, $S(u_c)$. This typically requires the following steps:

1. Either project the coarse scale data, u_c, onto the fine scale mesh or simply evaluate u_c at the fine scale quadrature points.
2. Use another Newton-based procedure to solve for u_f. Given an approximation $u_f^{(k)}$.

   $$J_f^{(i)} \Delta_f = -F_f\left(u_f^{(i)}, u_c\right),$$

   where

   $$J_f^{(i)} = \frac{\partial F_f}{\partial u_f}\Big|_{u_f^{(i)}},$$

   and we set $u_f^{(i+1)} = u_f^{(i)} + \Delta_f$. Continue until convergence criteria is met.
3. Project u_f back to the coarse scale or directly use the approximation.

At the discrete lvel, the projection steps above may involve a matrix-vector product or integrating the solutions against the appropriate test functions. MrHyDE precomputes the coarse scale basis functions at the fine scale quadrature points, so all of these integrals occur at the fine scale.

Example 5.1.1 *In the simple case of linear operators, the discretized coupled system can be written as,*

$$\begin{bmatrix} A_{ff} & A_{fc} \\ A_{cf} & A_{cc} \end{bmatrix} \begin{bmatrix} u_c \\ u_f \end{bmatrix} = \begin{bmatrix} b_c \\ b_f \end{bmatrix},$$

giving

$$h(u_c) = u_f = A_{ff}^{-1}(b_f - A_{cf}u_c),$$

and the coarse scale equation,

$$(A_{11} - A_{fc}A_{ff}^{-1}A_{cf})u_c = b_c - A_{fc}A_{ff}^{-1}b_f,$$

which includes the well-known Schur-complement operator.

The coarse scale Jacobian can be written as,

$$J_c = \frac{\partial}{\partial u_c} F_c(u_c, h(u_c)) = \frac{\partial F_c}{\partial u_c} + \frac{\partial F_c}{\partial h}\frac{\partial h}{\partial u_c}.$$

Computing $\partial F_c/\partial h$ is straightforward since h appears in the second argument, similar to u_f. However, computing $\partial h/\partial u_c$, i.e., the derivative of the upscaled information with respect to the coarse scale data, requires more work. MrHyDE assumes that the subgrid models will provide $\partial h/\partial u_c$, which can be written as

$$\frac{\partial h}{\partial u_c} = \frac{\partial h}{\partial u_f}\frac{\partial u_f}{\partial u_c},$$

where $\partial h/\partial u_f$ is often one, but not necessarily always true. The key is computing $\partial u_f/\partial u_c$. To obtain this information, we differentiate the final scale equation with respect to u_c,

$$\frac{\partial}{\partial u_c} F_f(u_c, u_f) = \frac{\partial F_f}{\partial u_c} + \frac{\partial F_f}{\partial u_f}\frac{\partial u_f}{\partial u_c} = 0. \tag{5.6}$$

We note that $\partial F_f/\partial u_f = J_f$, which we have already computed. In MrHyDE, we store the LU factorization (if using a direct method) or multilevel preconditioner (if using an iterative method) for the last Jacobian in the subgrid Newton solver, and solve

$$J_f \frac{\partial u_f}{\partial u_c} = -\frac{\partial F_f}{\partial u_c},$$

which typically involves one additional linear solve per degree-of-freedom in u_c. Since the matrix is already assembled and numerically factorized, this sensitivity propagation is typically faster than the subgrid nonlinear solve for u_f.

5.2. Transient Cases

For transient simulations, we consider separately the cases where the nonlinear elimination occurs before or after the discretization in time. If we first discretize (5.1) in time, e.g., with a diagonally implicit RK scheme, and then perform nonlinear elimination to compute each of the stage solutions, then we call this *synchronous time integration*. Conversely, if the coarse scale and fine scale equations evolve at different time scales, then we might want to pursue a multirate formulation where the fine scale equation uses a different time integrator than the coarse scale equation. We refer to this as *asynchronous time integration*.

5.2.1. Synchronous Time Integration

First, we consider the simpler case where we discretize (5.1) in time and then perform nonlinear elimination. Note that this does imply that both the coarse and fine scale equations evolve at the same rate. Recall from Chapter 4.3.3, that in MrHyDE, time integration schemes are implemented using a Butcher tableau and a BDF formula. Once discretized, we solve a sequence of nonlinear systems

$$\begin{aligned} g(z_{c,i}) + F_c(m(z_{c,i}), m(z_{f,i})) &= 0, \\ g(z_{f,i}) + F_f(m(z_{c,i}), m(z_{f,i})) &= 0, \end{aligned}$$

for the stage solutions $z_{f,i}$ and $z_{c,i}$, where g and m involve z_i and possibly previous stage solutions, and the step solutions are computed using

$$\begin{aligned} u_{f,n+1} &= u_{f,n} + \sum_{i=1}^{s} \left(z_{f,i} - u_{f,n} \right), \\ u_{c,n+1} &= u_{c,n} + \sum_{i=1}^{s} \left(z_{c,i} - u_{c,n} \right), \end{aligned}$$

The reduced version of this problem solves

$$g(z_{c,i}) + F_c(m(z_{c,i}), h(m(z_{c,i}))) = 0,$$

for $z_{c,i}$.

5.2.2. Asynchronous Time Integration

The asynchronous case is more complicated due to the fact that the coarse and fine scale integrators each have their own time step size, BDF formula and Butcher tableau. In addition, if the fine scale integrator uses a finer step size, then the coarse scale solution needs to be interpolated in time. For simplicity, we describe the solution process assuming we are using a Backward Euler integrator at both the coarse and fine scale, with M substeps at the fine scale. Other variations are implemented and the solution process is similar.

Given $u_{c,n}$, the reduced version of the problem solves

$$\frac{u_{c,n+1} - u_{c,n}}{\Delta t_c} + F_c(u_{c,n+1}, h(u_{c,n+1})) = 0,$$

where we have used the fact that the backward Euler scheme is fairly simple to omit g and m. Assuming that $u_{f,n}$ is given, we need to time step the fine scale model to obtain $u_{f,n+1}$ and $\partial u_{f,n+1} / \partial u_{c,n+1}$. We obtain this by taking M subgrid time steps to go from t_n to t_{n+1}. To simplify

the notation, we set $u_f^{(0)} = u_{f,n}$ and compute

$$\begin{aligned}
&\frac{u_f^{(1)} - u_f^{(0)}}{\Delta t_f} + F_f(p(t^{(1)}, u_{c,n+1}), u_f^{(1)}) = 0 \\
&\frac{u_f^{(2)} - u_f^{(1)}}{\Delta t_f} + F_f(p(t^{(2)}, u_{c,n+1}), u_f^{(2)}) = 0 \\
&\frac{u_f^{(3)} - u_f^{(2)}}{\Delta t_f} + F_f(p(t^{(3)}, u_{c,n+1}), u_f^{(3)}) = 0 \\
&\frac{u_f^{(4)} - u_f^{(3)}}{\Delta t_f} + F_f(p(t^{(4)}, u_{c,n+1}), u_f^{(4)}) = 0 \\
&\qquad\qquad\vdots \qquad\qquad\qquad\vdots
\end{aligned}$$

where $t^{(k)} = t_n + k\Delta t_f$ and $p(t^{(k)}, u_{c,n+1})$ denotes the interplant of $u_{c,n+1}$ in time at $t^{(k)}$ which may also use $u_{c,n+1}, u_{c,n}, \ldots$ if provided by the coarse scale model, e.g., if using a BDF method. At the end, we set $u_{f,n+1} = u_f^{(M)}$. To compute $\partial u_{f,n+1}/\partial u_{c,n+1}$, we need to propagate this through the solution sequence appropriately. For ease of notation, set $v^{(k)} = \partial u_f^{(k)}/\partial u_{c,n+1}$. Then we have

$$\begin{aligned}
&\frac{v^{(1)} - v^{(0)}}{\Delta t_f} + \frac{\partial F_f}{\partial p}\frac{\partial p}{\partial u_{c,n+1}} + \frac{\partial F_f}{\partial u_f}v^{(1)} = 0 \\
&\frac{v^{(2)} - v^{(1)}}{\Delta t_f} + \frac{\partial F_f}{\partial p}\frac{\partial p}{\partial u_{c,n+1}} + \frac{\partial F_f}{\partial u_f}v^{(2)} = 0 \\
&\frac{v^{(3)} - v^{(2)}}{\Delta t_f} + \frac{\partial F_f}{\partial p}\frac{\partial p}{\partial u_{c,n+1}} + \frac{\partial F_f}{\partial u_f}v^{(3)} = 0 \\
&\frac{v^{(4)} - v^{(3)}}{\Delta t_f} + \frac{\partial F_f}{\partial p}\frac{\partial p}{\partial u_{c,n+1}} + \frac{\partial F_f}{\partial u_f}v^{(4)} = 0 \\
&\qquad\qquad\vdots \qquad\qquad\qquad\vdots
\end{aligned}$$

where $v^{(0)} = 0$ and we set $\partial u_{f,n+1}/\partial u_{c,n+1} = v^{(M)}$. We also note that $\partial p/\partial u_{c,n+1}$ needs to be evaluated at $t^{(k)}$ if the interpolant is higher than first-order (constant).

Remark 5.2.1 *Numerically, it appears to be stable to use BWE (or any DIRK scheme) at the coarse scale and FWE (or any explicit RK scheme) at the fine scale, if a local CFL condition is satisfied for the local fine scale models. At present, it appears that this CFL is slightly larger than the CFL if one were to use the explicit scheme on the global system.*

6. HYBRIDIZED METHODS

Assume we have a system of transient nonlinear PDEs given by,

$$\frac{\partial u}{\partial t} + \nabla \cdot F(u) = G(u), \quad \text{in } \Omega \times (0,T], \tag{6.1}$$

with appropriate initial and boundary conditions which we omit for simplicity.

Let Ω be partitioned into N non-overlapping subdomains/elements, i.e., $\Omega = \cup_{i=1}^{N} \Omega_i$. Let $\mathcal{E} = \cup_{1 \leq i < j \leq N} \partial\Omega_i \cap \partial\Omega_j$ denotes the skeleton of the partition. We further assume that (6.1) is equivalent to the following decomposed system, where on each interface between elements of the partition, we have

$$\text{Flux}_i(\lambda, u_i) \cdot n_i = \text{Flux}_j(\lambda, u_j) \cdot n_j, \quad \text{on } \Omega_i \cap \Omega_j \times (0,T] \tag{6.2}$$

i.e., conservation of the flux over the interface, and on each element we have

$$\begin{cases} \frac{\partial u_i}{\partial t} + \nabla \cdot F(u_i) = G(u_i), & \text{in } \Omega_i \times (0,T], \\ u_i = \lambda, & \partial\Omega_i \times (0,T]. \end{cases} \tag{6.3}$$

Here, we introduced a field λ which exists only on $\mathcal{E}$, and $\lambda = u$ at the continuous level.

While this has been presented as a nonoverlapping domain decomposition problem, it is important to note that the coupled systems (6.2)-(6.3) also appears in hybridizable DG methods. In one former case, the number of subdomains is relatively small and matrix-free approaches are usually preferred with physically motivated multilevel preconditioning strategies, e.g., balancing. In the latter case, the number of subdomains depends on the number of elements in the mesh and the Schur complement associated with the interface condition (6.2), is constructed directly and solved using standard multilevel preconditioned Krylov methods, e.g., algebraic multigrid. For the most part, these two perspectives: nonoverlapping domain decomposition and hybridized methods, are interchangeable. The notable exception will be in designing high-order time integration methods, e.g., multistage Runge Kutta methods.

MrHyDE exploits the fact that the coupled system in (6.2)-(6.3) is in the same form as the coupled multiscale system (5.1), with the exception that there is no time derivative in (6.2), so (6.2)-(6.3) is actually a DAE.

7. PARALLELISM

MrHyDE is designed to take advantage of the inter-node and intra-node parallelism embedded in or enabled through various Trilinos packages for large-scale simulations.

7.1. Inter-Node Parallelism

At this time, MrHyDE only uses MPI for inter-node, or inter-core communication. In order to use inter-node parallelism, the mesh must be decomposed onto equal number of partitions as the number of requested MPI processes. For the most part, the inter-node parallelism happens behind the scenes in several of the Trilinos packages, e.g., Tpetra, Belos, MueLu, etc. In a few places in MrHyDE, MPI allReduce commands are used to determine a total or a max number of a quantity over all of the MPI processes.

7.2. Intra-Node Parallelism

Intra-node, or on-node, parallelism refers to the utilization of multiple threads or concurrently executed tasks within a given node, The ability to exploit intra-node parallelism depends on the computational architecture. Many modern CPU cores can allow for multiple threads, typically two or four, and modern heterogeneous systems, such the hybrid CPU/GPU systems, provide several thousand SIMD threads. Unfortunately, designing algorithms and data structures to exploit the full computational power of any of these architectures is not trivial, and may vary significantly between architectures. MrHyDE utilizes Kokkos to enable efficient implementations that are performance portable across a wide range of architectures.

A full description of Kokkos is beyond the scope of this document. Briefly, MrHyDE utilizes Kokkos Devices, which are a MemorySpace plus an ExecutionSpace, to define where and how computations occur on the node. Two different devices are used prevalently throughout the code: HostDevice and AssemblyDevice. At this time, the HostDevice must be

- <Kokkos::Serial, Kokkos::HostSpace>.

The AssemblyDevice can be:

- <Kokkos::Serial, Kokkos::HostSpace>,
- <Kokkos::OpenMP, Kokkos::HostSpace>,
- <Kokkos::Cuda, Kokkos::CudaSpace>.

Additional options are available through Kokkos, but have not been tested at this time. The last consideration is where the linear algebra objects will be allocated and utilized. At this time, the linear algebra packages, and therefore the linear algebra interface, use a concept of a Node rather than a device although the differences between the two are minimal. MrHyDE can use the following solver nodes:

- Kokkos::Compat::KokkosSerialWrapperNode,
- Kokkos::Compat::KokkosOpenMPWrapperNode
- Kokkos::Compat::KokkosCudaWrapperNode.

The choice of the appropriate AssemblyDevice and SolverNode can have a tremendous impact on performance, but depends on the problem being solved. Not all preconditioners are available on the GPU and solver performance may actually be worse on the GPU versus the CPU. Moreover, many linear solvers and preconditioners require substantial amounts of memory which may exceed that available on the GPU.

Finally, we comment on one notable exception from the MemorySpaces:

- Kokkos::CudaUVMSpace.

UVM, or Unified Virtual Memory, is a memory space that is accessible to either the CPU or the GPU on a heterogeneous device. While this space was quite popular several years ago due to the relative ease of implementations and the ability to overlap some communication with computation, it is not a concept that applies to all GPU architectures and can cause unintended memory transfers, and therefore is not recommended for usage in MrHyDE. At this time, MrHyDE does not utilize UVM space in any of the custom routines. However, Trilinos has not completely purged UVM from all of the packages and this is still the default MemorySpace for some packages where Cuda is enabled. To mitigate this discrepancy, MrHyDE employs a compatibility layer with temporary Kokkos Views that use UVM memory. The data in these Views gets copied into MrHyDE UVM-free Views once the interaction with the specific Trilinos packages is complete.

8. OPTIMIZATION AND INVERSION

As previously mentioned, MrHyDE provided the ability to solve an adjoint equation for coupled multiphysics and multiscale systems. This adjoint solution can then be used to compute a gradient of an objective function with respect to the active or discretized parameters. A full derivation of the adjoint-based gradient is beyond the intended scope of this report, but we will mention a few key aspects that may impact a user:

- For transient problems, the adjoint problems runs backwards in time and needs to be linearized around the state approximation. Thus, the state solutions needs to be available as the adjoint solver progresses. If an adjoint solve is required, MrHyDE automatically stores the full forward solution at each time node. Checkpointing is a common approach to reduce the memory requirements in storing the forward solution, but this is not implemented in MrHyDE.
- The adjoint solve requires the derivative of the forward residual with respect to the state (the Jacobian) and the derivative of the objective function with respect to the state. The gradient calculation also requires the derivative of the forward residual with respect to the active and/or discretized parameters.
- For these discrete adjoints, the Jacobian for the adjoint system is the transpose of the forward Jacobian. MrHyDE assembles the adjoint Jacobian directly by transposing the row and column indices in the scatter step. No transposes are required at the linear algebra level.
- MrHyDE does not support Hessians or Hessian-vector products at this time (future work).
- The adjoint gradients are supported for multi-step time integrators, but not for multi-stage integrators. This is work in progress.

MrHyDE support several different types of objective functions for optimization and inversion. The general form of the objective function is given by:

$$F(u,\lambda) = \sum_{i=1}^{N_{\text{obj}}} \left(\omega_i f_i(u,\lambda) + \sum_{i=j}^{N_{j,r}} \eta_{i,j} R_{i,j}(\lambda) \right), \tag{8.1}$$

where u denotes the state variable, λ denotes the parameters (discretized or active), $f_i(u,\lambda)$ is a misfit function, ω_i is the weight on this misfit function, $R_{i,j}(\lambda)$ is a regularization term, and $\eta_{i,j}$ is the weight on this regularization term. Multiple misfit functions are allowed, but currently these can only the weighted average, given by (8.1), can be optimized. Multi-objective optimization is planned for future work.

The misfit terms in the objective can have multiple forms:

Integrated control The misfit for an integrated control function is given by:

$$f_i(u,\lambda) = \int_\Omega \left(r(u,\lambda,x) - q(x)\right)^2 \, dx,$$

where $r(u,\lambda,x)$ is the response function, $q(x)$ is a given target function. An example of an integrated response objective is

```
Postprocess:
    compute objective: true
    compute sensitivities: false
    Objective functions:
      obj0:
        type: integrated control
        function: '1.0*(e-targ)^2'
        weight: 0.0625
```

Integrated response The misfit for an integrated response function is given by:

$$f_i(u,\lambda) = (\overline{r} - \overline{q})^2, \qquad \overline{r} = \int_\Omega r(u,\lambda,x) \, dx,$$

where $r(u,\lambda,x)$ is the response function and $\overline{q}$ is given target data. An example of an integrated response objective is

```
Postprocess:
    compute objective: true
    compute sensitivities: true
    Objective functions:
      obj0:
        type: integrated response
        response: 'e'
        target: 1.0
        weight: 0.5
```

Discrete control The misfit for a discrete control function is given by:

$$f_i(u,\lambda) = \|\mathbf{u} - \mathbf{q}\|_2^2,$$

where $\mathbf{u}$ is the discrete solution vector and $\mathbf{q}$ is a given discrete target vector. An example of a discrete control objective is

```
Postprocess:
    compute objective: true
    compute sensitivities: true
    Objective functions:
      obj1:
        type: discrete control
        weight: 0.5
```

Sensor response The misfit for a sensor response function is given by:

$$f_i(u,\lambda) = \sum_{j=1}^{N_{\text{sens}}} \left(r_j - q_j\right)^2, \qquad r_j = r(u,\lambda,x_j),$$

where r_j is a response function, x_j is a sensor location and q_j is given sensor data. An example of a sensor response objective is

```
Postprocess:
    compute objective: true
    Objective functions:
      obj0:
        type: sensors
        sensor points file: sensor_points.dat
        sensor data file: sensor_data.dat
        save sensor data: false
        response: 'e'
        weight: 1.0
```

The regularization functions can be any function of the parameter, but should not depend on the state. L^2, H^1 and total variation regularization can be implemented in this way from the input file. Note that the regularization functions are tied to an objective function and an objective function can be associated with multiple regularization functions. For example, the following code adds two regularization terms: one a volumetric L^2 term and one a boundary H^1 term.

```
Postprocess:
    compute objective: true
    Objective functions:
      obj_dx:
        type: sensors
        sensor points file: sensor_points.dat
        sensor data file: sensor_dx_data.dat
        response: 'dx'
        weight: 0.5
        Regularization functions:
          reg0:
            type: integrated
            location: volume
            function: mufield^2
            weight: 1.0e-5
          breg0:
            type: integrated
            location: boundary
            boundary name: top
            function: '1.0*(grad(disc_trac)[x])^2'
            weight: 0.5e-5
```

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Technical Library	1911	sanddocs@sandia.gov

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop

Hardcopy—External

Number of Copies	Name(s)	Company Name and Company Mailing Address