Nalu Documentation

Release 1.2.0

Nalu Development Team

Oct 31, 2017

Contents

CHAPTER 1

User Manual

Building Nalu

Building Nalu Semi-Automatically Using Spack

Mac OS X or Linux

The following describes how to build Nalu and its dependencies mostly automatically on your Mac using Spack. This can also be used as a template to build Nalu on any Linux system with Spack.

Step 1

This assumes you have a (Homebrew) installation of GCC installed already (we are using GCC 7.2.0). These instructions have been tested on OSX 10.11 and MacOS 10.12. MacOS 10.12 will not build CMake or Pkg-Config with GCC anymore because they will pick up system header files that have objective C code in them. We build Nalu using Spack on MacOS Sierra by using Homebrew to install `cmake` and `pkg-config` and defining these as external packages in Spack (see packages.yaml).

Step 2

Checkout the official Spack repo from github (we will checkout into `${HOME}`):

```
cd ${HOME} && git clone https://github.com/LLNL/spack.git
```

Step 3

Add Spack shell support to your `.profile` or `.bash_profile` etc, by adding the lines:

```
export SPACK_ROOT=${HOME}/spack
source ${SPACK_ROOT}/share/spack/setup-env.sh
```

Step 4

Run the setup_spack.sh script from the repo which tries to find out what machine your on and then copies the corresponding `*.yaml` configuration files to your Spack installation:

```
cd ${HOME} && git clone https://github.com/NaluCFD/NaluSpack.git
cd ${HOME}/NaluSpack/spack_config && ./setup_spack.sh
```

Step 5

Try `spack info nalu` to see if Spack works. If it does, check the compilers you have available by:

```
machine:~ user$ spack compilers
==> Available compilers
-- clang sierra-x86_64 ------------------------------------------
clang@9.0.0-apple

-- gcc sierra-x86_64 --------------------------------------------
gcc@7.2.0  gcc@6.4.0  gcc@5.4.0
```

Step 6

Install Nalu with whatever version of GCC (7.2.0 for us) you prefer by editing and running the `install_nalu_gcc_mac.sh` script in the NaluSpack repo:

```
cd ${HOME}/NaluSpack/install_scripts && ./install_nalu_gcc_mac.sh
```

That should be it! Spack will install using the constraints we've specified in `shared_constraints.sh` as can be seen in the install script.

NREL's Peregrine Machine

The following describes how to build Nalu and its dependencies mostly automatically on NREL's Peregrine machine using Spack. This can also be used as a template to help build Nalu on any Linux system with Spack.

Step 1

Login to Peregrine, and checkout the `https://github.com/NaluCFD/NaluSpack.git` repo (we will be cloning into the ${HOME} directory):

```
cd ${HOME} && git clone https://github.com/NaluCFD/NaluSpack.git
```

One first thing to note is that the login nodes and the compute nodes on Peregrine run different OSes. So programs will be organized in Spack according to the OS they were built on, i.e. a login node (rhel6) typically called the front-end or compute node (centos6) typically called the back-end. You can see this in the directory structure where the programs will be built which will be located in `${SPACK_ROOT}/opt`. You should build on a compute node.

Step 2

Checkout the official Spack repo from github:

```
cd ${HOME} && git clone https://github.com/LLNL/spack.git
```

Step 3

Configure your environment in the recommended way. You should purge all modules and only load GCC 5.2.0 in your login script. In the example .bash_profile in the repo we also load Python. If you have problems building with Spack on Peregrine, it is most likely your environment has deviated from this recommended one. Even when building with the Intel compiler in Spack, this is the recommended environment.

```
{
module purge
module load gcc/5.2.0
module load python/2.7.8
unload mkl
} &> /dev/null
```

Also add Spack shell support to your `.bash_profile` as shown in the example .bash_profile in the repo or the following lines:

```
export SPACK_ROOT=${HOME}/spack
source ${SPACK_ROOT}/share/spack/setup-env.sh
```

Log out and log back in or source your `.bash_profile` to get the Spack shell support loaded. Try `spack info nalu` to see if Spack works.

Step 4

Configure Spack for Peregrine. This is done by running the setup_spack.sh script provided which tries finding what machine you're on and copying the corresponding `*.yaml` file to your Spack directory:

```
cd ${HOME}/NaluSpack/spack_config && ./setup_spack.sh
```

Step 5

Try `spack info nalu` to see if Spack works.

Step 6

Note the build scripts provided here adhere to the official versions of the third party libraries we test with, and that you may want to adhere to using them as well. Also note that when you checkout the latest Spack, it also means you will be using the latest packages available if you do not set constraints at install time and the newest packages may not have been tested to build correctly on NREL machines yet. So specifying versions of the TPL dependencies in this step is recommended.

Install Nalu using a compute node either interactively (`qsub -V -I -l nodes=1:ppn=24, walltime=4:00:00 -A <allocation> -q short`) with the example script install_nalu_gcc_peregrine.sh or edit the script to use the correct allocation and `qsub install_nalu_gcc_peregrine.sh`.

That's it! Hopefully the `install_nalu_gcc_peregrine.sh` script installs the entire set of dependencies and you get a working build of Nalu on Peregrine...after about 2 hours of waiting for it to build.

To build with the Intel compiler, note the necessary commands in install_nalu_intel_peregrine.sh batch script (note you will need to point `${TMPDIR}` to disk as it defaults to RAM and will cause problems when building Trilinos).

Then to load Nalu (and you will need Spack's openmpi for Nalu now) into your path you will need to `spack load openmpi %compiler` and `spack load nalu %compiler`, using `%gcc` or `%intel` to specify which to load.

NREL's Merlin Machine

The following describes how to build Nalu and its dependencies mostly automatically on NREL's Merlin machine using Spack.

Step 1

Login to Merlin, and checkout the `https://github.com/NaluCFD/NaluSpack.git` repo (we will be cloning into the ${HOME} directory):

```
cd ${HOME} && git clone https://github.com/NaluCFD/NaluSpack.git
```

On Merlin, thankfully the login nodes and compute nodes use the same OS (centos7), so building on the login node will still allow the package to be loaded on the compute node. Spack will default to using all cores, so be mindful using it on a compute node. You should probably build on a compute node, or set Spack to use a small number of processes when building.

Step 2

Checkout the official Spack repo from github:

`cd ${HOME} && git clone https://github.com/LLNL/spack.git`

Step 3

Configure your environment in the recommended way. You should purge all modules and load `GCCcore/4.9.2` in your login script. See the example .bash_profile . If you have problems building with Spack on Merlin, it is most likely your environment has deviated from this recommended one. Even when building with the Intel compiler in Spack, this is the recommended environment.

```
module purge
module load GCCcore/4.9.2
```

Also add Spack shell support to your `.bash_profile` as shown in the example .bash_profile in the repo or the following lines:

```
export SPACK_ROOT=${HOME}/spack
source ${SPACK_ROOT}/share/spack/setup-env.sh
```

Log out and log back in or source your `.bash_profile` to get the Spack shell support loaded.

Step 4

Configure Spack for Merlin. This is done by running the setup_spack.sh script provided which tries finding what machine you're on and copying the corresponding `*.yaml` file to your Spack directory:

```
cd ${HOME}/NaluSpack/spack_config && ./setup_spack.sh
```

Step 5

Try `spack info nalu` to see if Spack works.

Step 6

Note the build scripts provided here adhere to the official versions of the third party libraries we test with, and that you may want to adhere to using them as well. Also note that when you checkout the latest Spack, it also means you will be using the latest packages available if you do not specify a package version at install time and the newest packages may not have been tested to build correctly on NREL machines yet. So specifying versions of the TPL dependencies in this step is recommended.

Install Nalu using a compute node either interactively (`qsub -V -I -l nodes=1:ppn=24, walltime=4:00:00 -A <allocation> -q batch`) or with the example batch script install_nalu_gcc_merlin.sh by editing to use the correct allocation and then `qsub install_nalu_gcc_merlin.sh`.

That's it! Hopefully that command installs the entire set of dependencies and you get a working build of Nalu on Merlin.

To build with the Intel compiler, note the necessary commands in install_nalu_intel_merlin.sh batch script.

Then to load Nalu (and you will need Spack's openmpi for Nalu now) into your path you will need to `spack load openmpi %compiler` and `spack load nalu %compiler`, using `%gcc` or `%intel` to specify which to load.

Development Build of Nalu

When building Nalu with Spack, Spack will cache downloaded archive files such as `*.tar.gz` files. However, by default Spack will also erase extracted or checked out ('staged') source files after it has built a package successfully. Therefore if your build succeeds, Spack will have erased the Nalu source code it checked out from Github.

The recommended way to get a version of Nalu you can develop in is to checkout Nalu yourself outside of Spack and build this version using the dependencies Spack has built for you. To do so, checkout Nalu:

```
git clone https://github.com/NaluCFD/Nalu.git
```

Next, create your own directory to build in, or use the existing `build` directory in Nalu to run the CMake configuration. When running the CMake configuration, point Nalu to the dependencies by using `spack location -i <package>`. For example in the `build` directory run:

```
cmake -DTrilinos_DIR:PATH=$(spack location -i nalu-trilinos) \
      -DYAML_DIR:PATH=$(spack location -i yaml-cpp) \
      -DCMAKE_BUILD_TYPE=RELEASE \
      ..
make
```

There are also scripts available for this according to machine here. This should allow you to have a build of Nalu in which you are able to continuosly modify the source code and rebuild.

Development Build of Trilinos

If you want to go even further into having a development build of Trilinos while using TPLs Spack has built for you, checkout Trilinos somewhere and see the example configure script for Trilinos according to machine here.

Building Nalu Manually

If you prefer not to build using Spack, below are instructions which describe the process of building Nalu by hand.

Linux and OSX

The instructions for Linux and OSX are mostly the same, except on each OS you may be able to use a package manager to install some dependencies for you. Using Homebrew on OSX is one option listed below. Compilers and MPI are expected to be already installed. If they are not, please follow the open-mpi build instructions. Below, we are using OpenMPI v1.10.4 and GCC v4.9.2. Start by create a `$nalu_build_dir` to work in.

Homebrew

If using OSX, you can install many dependencies using Homebrew. Install Homebrew on your local machine and reference the list below for some packages Homebrew can install for you which allows you to skip the steps describing the build process for each application, but not that you will need to find the location of the applications in which Homebrew has installed them, to use when building Trilinos and Nalu.

```
brew install openmpi
brew install cmake
brew install libxml2
brew install boost
brew tap homebrew/science
brew install superlu43
```

CMake v3.6.1

CMake is provided here.

Prepare:

```
cd $nalu_build_dir/packages
curl -o cmake-3.6.1.tar.gz http://www.cmake.org/files/v3.6/cmake-3.6.1.tar.gz
tar xf cmake-3.6.1.tar.gz
```

Build:

```
cd $nalu_build_dir/packages/cmake-3.6.1
./configure --prefix=$nalu_build_dir/install
make
make install
```

SuperLU v4.3

SuperLU is provided here.

Prepare:

```
cd $nalu_build_dir/packages
curl -o superlu_4.3.tar.gz http://crd-legacy.lbl.gov/~xiaoye/SuperLU/superlu_4.3.tar.
↪gz
tar xf superlu_4.3.tar.gz
```

Build:

```
cd $nalu_build_dir/packages/SuperLU_4.3
cp MAKE_INC/make.linux make.inc
```

To find out what the correct platform extension PLAT is:

```
uname -m
```

Edit `make.inc` as shown below (diffs shown from baseline).

```
PLAT = _x86_64
SuperLUroot    = /your_path/install/SuperLU_4.3 i.e., $nalu_build_dir/install/SuperLU_
↪4.3
BLASLIB        = -L/usr/lib64 -lblas
CC             = mpicc
FORTRAN        = mpif77
```

On some platforms, the `$nalu_build_dir` may be mangled. In such cases, you may need to use the entire path to `install/SuperLU_4.3`.

Next, make some new directories:

```
mkdir $nalu_build_dir/install/SuperLU_4.3
mkdir $nalu_build_dir/install/SuperLU_4.3/lib
mkdir $nalu_build_dir/install/SuperLU_4.3/include
cd $nalu_build_dir/packages/SuperLU_4.3
make
cp SRC/*.h $nalu_build_dir/install/SuperLU_4.3/include
```

Libxml2 v2.9.2

Libxml2 is found here.

Prepare:

```
cd $nalu_build_dir/packages
curl -o libxml2-2.9.2.tar.gz http://www.xmlsoft.org/sources/libxml2-2.9.2.tar.gz
tar -xvf libxml2-2.9.2.tar.gz
```

Build:

```
cd $nalu_build_dir/packages/libxml2-2.9.2
CC=mpicc CXX=mpicxx ./configure -without-python --prefix=$nalu_build_dir/install
make
make install
```

Boost v1.60.0

Boost is found here.

Prepare:

```
cd $nalu_build_dir/packages
curl -o boost_1_60_0.tar.gz http://iweb.dl.sourceforge.net/project/boost/boost/1.60.0/
↪boost_1_60_0.tar.gz
tar -zxvf boost_1_60_0.tar.gz
```

Build:

```
cd $nalu_build_dir/packages/boost_1_60_0
./bootstrap.sh --prefix=$nalu_build_dir/install --with-libraries=signals,regex,
↪filesystem,system,mpi,serialization,thread,program_options,exception
```

Next, edit `project-config.jam` and add a 'using mpi', e.g,

using mpi: /path/to/mpi/openmpi/bin/mpicc

```
./b2 -j 4 2>&1 | tee boost_build_one
./b2 -j 4 install 2>&1 | tee boost_build_intall
```

YAML-CPP v0.5.3

YAML is provided here. Versions of Nalu before v1.1.0 used earlier versions of YAML-CPP. For brevity only the latest build instructions are discussed and the history of the Nalu git repo can be used to find older installation instructions if required.

Prepare:

```
cd $nalu_build_dir/packages
git clone https://github.com/jbeder/yaml-cpp
```

Build:

```
cd $nalu_build_dir/packages/yaml-cpp
mkdir build
cd build
cmake -DCMAKE_CXX_COMPILER=mpicxx -DCMAKE_CXX_FLAGS=-std=c++11 -DCMAKE_CC_
↪COMPILER=mpicc -DCMAKE_INSTALL_PREFIX=$nalu_build_dir/install ..
make
make install
```

Zlib v1.2.8

Zlib is provided here.

Prepare:

```
cd $nalu_build_dir/packages
curl -o zlib-1.2.8.tar.gz http://zlib.net/zlib-1.2.8.tar.gz
tar -zxvf zlib-1.2.8.tar.gz
```

Build:

```
cd $nalu_build_dir/packages/zlib-1.2.8
CC=gcc CXX=g++ CFLAGS=-O3 CXXFLAGS=-O3 ./configure --prefix=$nalu_build_dir/install/
make
make install
```

HDF5 v1.8.16

HDF5 1.8.16 is provided here.

Prepare:

```
cd $nalu_build_dir/packages/
curl -o hdf5-1.8.16.tar.gz http://www.hdfgroup.org/ftp/HDF5/releases/hdf5-1.8.16/src/
↪hdf5-1.8.16.tar.gz
tar -zxvf hdf5-1.8.16.tar.gz
```

Build:

```
cd $nalu_build_dir/packages/hdf5-1.8.16
./configure CC=mpicc FC=mpif90 CXX=mpicxx CXXFLAGS="-fPIC -O3" CFLAGS="-fPIC -O3"␣
↪FCFLAGS="-fPIC -O3" --enable-parallel --with-zlib=$nalu_build_dir/install --prefix=
↪$nalu_build_dir/install
make
make install
make check
```

NetCDF v4.3.3.1 and Parallel NetCDF v1.6.1

In order to support all aspects of Nalu's parallel models, this combination of products is required.

Parallel NetCDF v1.6.1

Parallel NetCDF is provided on the Argon Trac Page.

Prepare:

```
cd $nalu_build_dir/packages/
tar -zxvf parallel-netcdf-1.6.1.tar.gz
```

Build:

```
cd parallel-netcdf-1.6.1
./configure --prefix=$nalu_install_dir CC=mpicc FC=mpif90 CXX=mpicxx CFLAGS="-I$nalu_
↪install_dir/include -O3" LDFLAGS=-L$nalu_install_dir/lib --disable-fortran
make
make install
```

Note that we have created an install directory that might look like `$nalu_build_dir/install`.

NetCDF v4.3.3.1

NetCDF is provided here.

Prepare:

```
cd $nalu_build_dir/packages/
curl -o netcdf-c-4.3.3.1.tar.gz https://codeload.github.com/Unidata/netcdf-c/tar.gz/
↪v4.3.3.1
tar -zxvf netcdf-c-4.3.3.1.tar.gz
```

Build:

```
cd netcdf-c-4.3.3.1
./configure --prefix=$nalu_install_dir CC=mpicc FC=mpif90 CXX=mpicxx CFLAGS="-I$nalu_
↪install_dir/include -O3" LDFLAGS=-L$nalu_install_dir/lib --enable-pnetcdf --enable-
↪parallel-tests --enable-netcdf-4 --disable-shared --disable-fsync --disable-
↪cdmremote --disable-dap --disable-doxygen --disable-v2
make -j 4
make check
make install
```

Trilinos

Trilinos is managed by the Trilinos project and can be found on Github.

Prepare:

```
cd $nalu_build_dir/packages/
git clone https://github.com/trilinos/Trilinos.git
cd $nalu_build_dir/packages/Trilinos
mkdir build
```

Build

Place into the build directory, one of the `do-configTrilinos_*` files, that can be obtained from the Nalu repo.

`do-configTrilinos_*` will be used to run cmake to build trilinos correctly for Nalu. Note that there are two files: one for 'release' and the other 'debug'. The files can be found on the Nalu GitHub site or copied from `$nalu_build_dir/packages/Nalu/build`, which is created in the Nalu build step documented below. For example:

Pull latest version of `do-configTrilinos_*` from Nalu's GitHub site:

```
curl -o $nalu_build_dir/packages/Trilinos/build/do-configTrilinos_release https://raw.
↪githubusercontent.com/NaluCFD/Nalu/master/build/do-configTrilinos_release
```

Or if you create the Nalu directory as directed below, simply copy one of the `do-configTrilinos_*` files from local copy of Nalu's git repository:

```
cp $nalu_build_dir/packages/Nalu/build/do-configTrilinos_release $nalu_build_dir/
↪packages/Trilinos/build
```

Now edit `do-configTrilinos_release` to modify the paths so they point to `$nalu_build_dir/install`.

```
cd $nalu_build_dir/packages/Trilinos/build
chmod +x do-configTrilinos_release
```

Make sure all other paths to netcdf, hdf5, etc., are correct.

```
./do-configTrilinos_release
make
make install
```

ParaView Catalyst

Optionally enable ParaView Catalyst for in-situ visualization with Nalu. These instructions can be skipped if you do not require in-situ visualization with Nalu.

Build ParaView SuperBuild v5.3.0

The ParaView SuperBuild builds ParaView along with all dependencies necessary to enable Catalyst with Nalu. Clone the ParaView SuperBuild within `$nalu_build_dir/packages`:

```
cd $nalu_build_dir/packages/
git clone --recursive https://gitlab.kitware.com/paraview/paraview-superbuild.git
cd paraview-superbuild
git fetch origin
git checkout v5.3.0
git submodule update
```

Create a new build folder in `$nalu_build_dir/`:

```
cd $nalu_build_dir
mkdir paraview-superbuild-build
cd paraview-superbuild-build
```

Copy `do-configParaViewSuperBuild` to `paraview-superbuild-build`. Edit `do-configParaViewSuperBuild` to modify the defined paths as follows:

```
mpi_base_dir=<same MPI base directory used to build Trilinos>
nalu_build_dir=<path to root nalu build dir>
```

Make sure the MPI library names are correct.

```
./do-configParaViewSuperBuild
make -j 8
```

Build Nalu ParaView Catalyst Adapter

Create a new build folder in `$nalu_build_dir/`:

```
cd $nalu_build_dir
mkdir nalu-catalyst-adapter-build
cd nalu-catalyst-adapter-build
```

Copy `do-configNaluCatalystAdapter` to `nalu-catalyst-adapter-build`. Edit `do-configNaluCatalystAdapter` and modify `nalu_build_dir` at the top of the file to the root build directory path.

```
./do-configNaluCatalystAdapter
make
make install
```

Nalu

Nalu is provided here. One may either build the released Nalu version 1.2.0 which matches with Trilinos version 12.12.1, or the master branch of Nalu which matches with the master branch or develop branch of Trilinos. If it is necessary to build an older version of Nalu, refer to the history of the Nalu git repo for instructions on doing so.

Prepare:

```
git clone https://github.com/NaluCFD/Nalu.git
```

Build

In `Nalu/build`, you will find the do-configNalu script. Copy the `do-configNalu_release` or `debug` file to a new, non-tracked file:

```
cp do-configNalu_release do-configNaluNonTracked
```

Edit the paths at the top of the files by defining the `nalu_build_dir variable`. Within `Nalu/build`, execute the following commands:

```
./do-configNaluNonTracked
make
```

This process will create `naluX` within the `Nalu/build` location. You may also build a debug executable by modifying the Nalu config file to use "Debug". In this case, a `naluXd` executable is created.

Build Nalu with ParaView Catalyst Enabled

If you have built ParaView Catalyst and the Nalu ParaView Catalyst Adapter, you can build Nalu with Catalyst enabled.

In `Nalu/build`, find `do-configNaluCatalyst`. Copy `do-configNaluCatalyst` to a new, non-tracked file:

```
cp do-configNaluCatalyst do-configNaluCatalystNonTracked
./do-configNaluCatalystNonTracked
make
```

The build will create the same executables as a regular Nalu build, and will also create a bash shell script named `naluXCatalyst`. Use `naluXCatalyst` to run Nalu with Catalyst enabled. It is also possible to run `naluX` with Catalyst enabled by first setting the environment variable:

```
export CATALYST_ADAPTER_INSTALL_DIR=$nalu_build_dir/install
```

Nalu will render images to Catalyst in-situ if it encounters the keyword `catalyst_file_name` in the `output` section of the Nalu input deck. The `catalyst_file_name` command specifies the path to a text file containing

ParaView Catalyst input deck commands. Consult the `catalyst.txt` files in the following Nalu regression test directories for examples of the Catalyst input deck command syntax:

```
ablForcingEdge/
mixedTetPipe/
steadyTaylorVortex/
```

```
output:
  output_data_base_name: mixedTetPipe.e
  catalyst_file_name: catalyst.txt
```

When the above regression tests are run, Catalyst is run as part of the regression test. The regression test checks that the correct number of image output files have been created by the test.

The Nalu Catalyst integration also supports running Catalyst Python script files exported from the ParaView GUI. The procedure for exporting Catalyst Python scripts from ParaView is documented in the Catalyst user guide. To use an exported Catalyst script, insert the `paraview_script_name` keyword in the `output` section of the Nalu input deck. The argument for the `paraview_script_name` command contains a file path to the exported script.

```
output:
  output_data_base_name: mixedTetPipe.e
  paraview_script_name: paraview_exported_catalyst_script.py
```

Running Nalu

This section describes the general process of setting up and executing Nalu, understanding the various input file options available to the user, and how to extract results and analyze them. For the simplest case, Nalu requires the user to provide a YAML input file with the options that control the run along with a computational mesh in Exodus-II format. More complex setups might require additional files:

- Trilinos MueLu preconditioner configuration in XML format
- ParaView Cataylst input file for in-situ visualizations
- Additional Exodus-II mesh files for solving different physics equation sets on different meshes, or for solution transfer to an input/output mesh.

Exodus-II File Format

Nalu requires the user to provide the computational mesh in Exodus-II format. The output and restart files generated by Nalu are also in Exodus-II format where the requested fields are output along side the mesh. The restart files from one Nalu simulation can serve as the input file for a subsequent simulation.

Several commercial mesh generation software support output to Exodus-II format. Two such software used by Nalu developers are:

- CUBIT
- Pointwise

Furthermore, NaluWindUtils provides an **`abl_mesh`** utility that can be used to generate simple structured meshes (output into Exodus-II format) for use with atmospheric boundary layer simulations.

Examining Exodus-II Files

Exodus-II uses the NetCDF format to store data, therefore, the several NetCDF utilities can be used to examine the file metadata. For example, the following code snippet shows the use of **ncdump** to examine the names of the mesh blocks and side sets, as well as the nodal fields available in a given mesh file.

```
ncdump -v eb_names,ss_names,name_nod_var channel_coarse_ic.g
# <output truncated to show only relevant parts>
data:

 eb_names =
  "interior" ;

 ss_names =
  "inlet",
  "outlet",
  "bottomwall",
  "topwall",
  "back",
  "front" ;

 name_nod_var =
  "turbulent_ke",
  "velocity_x",
  "velocity_y",
  "velocity_z" ;
```

For brevity, the example above has removed the NetCDF `dimensions` and `variables` sections to show just the contents of the variable names of interest. The output shows that the mesh in question contains one element block (`interior`) with six boundary planes (side-sets) and has two nodal fields: the velocity vector, and the turbulent kinetic energy scalar. **ncdump** can be invoked with the `-h` flag to print just the headers. Of particular interest is the NetCDF `dimensions` section that contains information about the total number of nodes, element, boundary faces, etc. in the mesh file.

Most visualization programs support loading Exodus-II mesh/solution files and can be used to visualize the flow fields generated by Nalu. Two open-source visualization programs available are:

- ParaView
- VisIt

Preliminary support for in-situ visualization using ParaView Catalyst is available within the Nalu code base and can be enabled by linking to Catalyst libraries during compile time. See input file specifications more details on setting up Cataylst for in-situ visualization of Nalu solution files.

Other Exodus-II Utilities

A brief description of some useful Exodus-II utilities are provided here. Please consult the documentation of these programs to understand the full range of options available.

decomp

> `decomp` is a SEACAS utility (available from a Trilinos install) that can be used to decompose a mesh file acros several MPI ranks for use in a subsequent paralell simulation.

epu

> `epu` performs the reverse action of `decomp`, i.e., it combines parallel decomposed files from a simulation into a single Exodus-II database. The simplest invocation is

```
epu -auto nalu_output.e.8.0
```

The `-auto` flag determines the database structured based on the file provided on the command line and combines the files (in the above example into `nalu_output.e`).

mapvar-kd

Map solution fields from one mesh to another mesh.

percept

The Percept project provides various tools to perform mesh refinement, higher-order promotion, etc. See documentation for `mesh_adapt` to determine various options available.

Invoking Nalu - Command-line options

Nalu's runtime behavior can be controlled by using several command line input options during invocation. Users can invoke `-h` to determine the various options available.

`-h, --help`
Print the help message describing all Nalu options and exit

`-i, --input-deck`
Use the filename provided as the input file. If this option is not provided, **naluX** will attempt to load a file called `nalu.i` in the current working directory as the input file.

`-o, --log-file`
The log file where the output generated by Nalu is directed to. If no file is provided, then **naluX** will use the base name of the Nalu input file with the extension `.log` as the output file. For example, if **naluX** was invoked as `naluX -i ABL.neutral.i` then the output will be redirected to a file named `ABL.neutral.log`. Note that the file is overwritten if it already exists.

`-v, --version`
Print the Nalu version string.

`-p, --pprint`
Enable parallel printing from all MPI ranks.

`-D, --debug`
Enable verbose debug printing to log file.

Nalu Input File

Nalu requires the user to provide an input file, in YAML format, during invocation at the command line using the *naluX -i* flag. By default, **naluX** will look for `nalu.i` in the current working directory to determine the mesh file as well as the run setup for execution. A sample `nalu.i` is shown below:

Listing 1.1: Sample Nalu input file for the Heat Conduction problem

```
# -*- mode: yaml -*-
#
# Example Nalu input file for a heat conduction problem
#

Simulations:
  - name: sim1
    time_integrator: ti_1
    optimizer: opt1
```

```
linear_solvers:
  - name: solve_scalar
    type: tpetra
    method: gmres
    preconditioner: sgs
    tolerance: 1e-3
    max_iterations: 75
    kspace: 75
    output_level: 0

realms:

  - name: realm_1
    mesh: periodic3d.g
    use_edges: no
    automatic_decomposition_type: rcb

    equation_systems:
      name: theEqSys
      max_iterations: 2

      solver_system_specification:
        temperature: solve_scalar

      systems:
        - HeatConduction:
            name: myHC
            max_iterations: 1
            convergence_tolerance: 1e-5

    initial_conditions:

      - constant: ic_1
        target_name: block_1
        value:
         temperature: 10.0

    material_properties:
      target_name: block_1
      specifications:
        - name: density
          type: constant
          value: 1.0
        - name: thermal_conductivity
          type: constant
          value: 1.0
        - name: specific_heat
          type: constant
          value: 1.0

    boundary_conditions:

    - wall_boundary_condition: bc_left
      target_name: surface_1
      wall_user_data:
        temperature: 20.0
```

```
    - wall_boundary_condition: bc_right
      target_name: surface_2
      wall_user_data:
        temperature: 40.0

    solution_options:
      name: myOptions

      use_consolidated_solver_algorithm: yes

      options:
      - element_source_terms:
          temperature: FEM_DIFF

    output:
      output_data_base_name: femHC.e
      output_frequency: 10
      output_node_set: no
      output_variables:
       - dual_nodal_volume
       - temperature

Time_Integrators:
  - StandardTimeIntegrator:
      name: ti_1
      start_time: 0
      termination_step_count: 25
      time_step: 10.0
      time_stepping_type: fixed
      time_step_count: 0
      second_order_accuracy: no

      realms:
        - realm_1
```

Nalu input file contains the following top-level sections that describe the simulation to be executed.

Realms

Realms describe the computational domain (via mesh input files) and the set of physics equations (Low-Mach Navier-Stokes, Heat Conduction, etc.) that are solved over this particular domain. The list can contain multiple computational domains (*realms*) that use different meshes as well as solve different sets of physics equations and interact via *solution transfer*. This section also contains information regarding the initial and boundary conditions, solution output and restart options, the linear solvers used to solve the linear system of equations, and solution options that govern the discretization of the equation set.

A special case of a realm instance is the input-output realm; this realm type does not solve any physics equations, but instead serves one of the following purposes:

- provide time-varying boundary conditions to one or more boundaries within one or more of the participating realms in the simulations. In this context, it acts as an *input* realm.
- extract a subset of data for output at a different frequency from the other realms. In this context, it acts as an *output* realm.

Inclusion of an input/output realm will require the user to provide the additional `transfers` section in the Nalu input file that defines the solution fields that are transferred between the realms. See *Physics Realm Options* for detailed documentation on all Realm options.

Linear Solvers

This section configures the solvers and preconditioners used to solve the resulting linear system of equations within Nalu. The linear system convergence tolerance and other controls are set here and can be used with multiple systems across different realms. See *Linear Solvers* for more details.

Time Integrators

This section configures the time integration scheme used (first/second order in time), the duration of simulation, fixed or adaptive timestepping based on Courant number constraints, etc. Each time integration section in this list can accept one or more `realms` that are integrated in time using that specific time integration scheme. See *Time Integration Options* for complete documentation of all time integration options available in Nalu.

Transfers

An optional section that defines one or more solution transfer definitions between the participating `realms` during the simulation. Each transfer definition provides a mapping of the to and from realm, part, and the solution field that must be transferred at every timestep during the simulation. See *Transfers* section for complete documentation of all transfer options available in Nalu.

Simulations

Simulations provides the top-level architecture that orchestrates the time-stepping across all the realms and the required equation sets.

Linear Solvers

The `linear_solvers` section contains a list of one or more linear solver settings that specify the solver, preconditioner, convergence tolerance for solving a linear system. Every entry in the YAML list will contain the following entries:

Note: The variable in the `linear_solvers` subsection are prefixed with `linear_solvers.name` but only the variable name after the period should appear in the input file.

`linear_solvers.name`
The key used to refer to the linear solver configuration in *`equation_systems.solver_system_specification`* section.

`linear_solvers.type`
The type of solver library used. Currently only one option (`tpetra`) is supported.

`linear_solvers.method`
The solver used for solving the linear system. Valid options are: `gmres`, `biCgStab`, `cg`.

`linear_solvers.preconditioner`
The type of preconditioner used. Valid options are `sgs`, `mt_sgs`, `muelu`.

`linear_solvers.tolerance`
The relative tolerance used to determine convergence of the linear system.

`linear_solvers.max_iterations`
Maximum number of linear solver iterations performed.

`linear_solvers.kspace`
The Krylov vector space.

`linear_solvers.output_level`
Verbosity of output from the linear solver during execution.

linear_solvers.muelu_xml_file_name
Only used when the *linear_solvers.preconditioner* is set to `muelu` and specifies the path to the XML filename that contains various configuration parameters for Trilinos MueLu package.

linear_solvers.write_matrix_files
A boolean flag indicating whether the matrix, the right hand side, and the solution vector are written to files during execution. The matrix files are written in MatrixMarket format. The default value is `no`.

linear_solvers.recompute_preconditioner
A boolean flag indicating whether preconditioner is recomputed during runs. The default value is `yes`.

linear_solvers.reuse_preconditioner
Boolean flag. Default value is `no`.

linear_solvers.summarize_muelu_timer
Boolean flag indicating whether MueLu timer summary is printed. Default value is `no`.

Time Integration Options

Time_Integrators
A list of time-integration options used to advance the `realms` in time. Each list entry must contain a YAML mapping with the key indicating the type of time integrator. Currently only one option, `StandardTimeIntegrator` is available.

```
Time_Integrators:
  - StandardTimeIntegrator:
      name: ti_1
      start_time: 0.0
      termination_step_count: 10
      time_step: 0.5
      time_stepping_type: fixed
      time_step_count: 0
      second_order_accuracy: yes

      realms:
        - fluids_realm
```

time_int.name
The lookup key for this time integration entry. This name must match the one provided in `Simulations` section.

time_int.termination_time
Nalu will stop the simulation once the `termination_time` has reached.

time_int.termination_step_count
Nalu will stop the simulation once the specified `termination_step_count` timesteps have been completed. If both *time_int.termination_time* and this parameter are provided then this parameter will prevail.

time_int.time_step
The time step (Δt) used for the simulation. If *time_int.time_stepping_type* is `fixed` this value does not change during the simulation.

time_int.start_time
The starting time step (default: `0.0`) when starting a new simulation. Note that this has no effect on restart which is controlled by *restart.restart_time* in the *restart* section.

time_int.time_step_count
The starting timestep counter for a new simulation. See *restart* for restarting from a previous simulation.

time_int.second_order_accuracy
A boolean flag indicating whether second-order time integration scheme is activated. Default: no.

time_int.time_stepping_type
One of fixed or adaptive indicating whether a fixed time-stepping scheme or an adaptive timestepping scheme is used for simulations. See *time_step_control* for more information on max Courant number based adaptive time stepping.

time_int.realms
A list of realms names. The names entered here must match *name* used in the realms section. Names listed here not found in realms list will trigger an error, while realms not included in this list but present in realms will not be initialized and silently ignored. This can cause the code to abort if the user attempts to access the specific realm in the *transfers* section.

Physics Realm Options

As mentioned previously, realms is a YAML list data structure containing at least one *Physics Realm Options* entry that defines the computational domain (provided as an Exodus-II mesh), the set of physics equations that must be solved over this domain, along with the necessary initial and boundary conditions. Each list entry is a YAML dictionary mapping that is described in this section of the manual. The key subsections of a Realm entry in the input file are

Realm subsection	Purpose
equation_systems	Set of physics equations to be solved
initial_conditions	Initial conditions for the various fields
boundary_conditions	Boundary condition for the different fields
material_properties	Material properties (e.g., fluid density, viscosity etc.)
solution_options	Discretization options
output	Solution output options (file, frequency, etc.)
restart	Optional: Restart options (restart time, checkpoint frequency etc.)
time_step_control	Optional: Parameters determining variable timestepping

In addition to the sections mentioned in the table, there are several additional sections that could be present depending on the specific simulation type and post-processing options requested by the user. A brief description of these optional sections are provided below:

Realm subsection	Purpose
turbulence_averaging	Generate statistics for the flow field
post_processing	Extract integrated data from the simulation
solution_norm	Compare the solution error to a reference solution
data_probes	Extract data using probes
actuator	Model turbine blades/tower using actuator lines
abl_forcing	Pressure source term to drive ABL flows to a desired velocity profile

Common options

name
The name of the realm. The name provided here is used in the *Time_Integrators* section to determine the time-integration scheme used for this computational domain.

mesh
The name of the Exodus-II mesh file that defines the computational domain for this realm. Note that only the base name (i.e., without the .NPROCS.IPROC suffix) is provided even for simulations using previously decomposed mesh/restart files.

automatic_decomposition_type
Used only for parallel runs, this indicates how the a single mesh database must be decomposed amongst the MPI processes during initialization. This option should not be used if the mesh has already been decomposed by an external utility. Possible values are:

Value	Description
rcb	recursive coordinate bisection
rib	recursive inertial bisection
linear	elements in order first n/p to proc 0, next to proc 1.
cyclic	elements handed out to id % proc_count

activate_aura
A boolean flag indicating whether an extra element is *ghosted* across the processor boundaries. The default value is `no`.

use_edges
A boolean flag indicating whether edge based discretization scheme is used instead of element based schemes. The default value is `no`.

polynomial_order
An integer value indicating the polynomial order used for higher-order mesh simulations. The default value is `1`. When *`polynomial_order`* is greater than 1, the Realm has the capability to promote the mesh to higher-order during initialization.

solve_frequency
An integer value indicating how often this realm is solved during time integration. The default value is `1`.

support_inconsistent_multi_state_restart
A boolean flag indicating whether restarts are allowed from files where the necessary field states are missing. A typical situation is when the simulation is restarted using second-order time integration but the restart file was created using first-order time integration scheme.

activate_memory_diagnostic
A boolean flag indicating whether memory diagnostics are activated during simulation. Default value is `no`.

balance_nodes
A boolean flag indicating whether node balancing is performed during simulations. See also *`balance_node_iterations`* and `balance_nodes_target`.

balance_node_iterations
The frequency at which node rebalancing is performed. Default value is `5`.

balance_node_target
The target balance ratio. Default value is `1.0`.

Equation Systems

equation_systems
`equation_systems` subsection defines the physics equation sets that are solved for this realm and the linear solvers used to solve the different linear systems.

Note: The variable in the *`equation_systems`* subsection are prefixed with `equation_systems.name` but only the variable name after the period should appear in the input file.

---

equation_systems.name
A string indicating the name used in log messages etc.

equation_systems.max_iterations
The maximum number of non-linear iterations performed during a timestep that couples the different equation systems.

equation_systems.solver_system_specification
A mapping containing `field_name: linear_solver_name` that determines the linear solver used for solving the linear system. Example:

```
solver_system_specification:
  pressure: solve_continuity
  enthalpy: solve_scalar
  velocity: solve_scalar
```

The above example indicates that the linear systems for the enthalpy and momentum (velocity) equations are solved by the linear solver corresponding to the tag `solve_scalar` in the `linear_systems` entry, whereas the continuity equation system (pressure Poisson solve) should be solved using the linear solver definition corresponding to the tag `solve_continuity`.

equation_systems.systems
A list of equation systems to be solved within this realm. Each entry is a YAML mapping with the key corresponding to a pre-defined equation system name that contains additional parameters governing the solution of this equation set. The predefined equation types are

Equation system	Description
LowMachEOM	Low-Mach Momentum and Continuity equations
Enthalpy	Energy equations
ShearStressTransport	$k-\omega$ SST equation set
TurbKineticEnergy	TKE equation system
MassFraction	Mass Fraction
MixtureFraction	Mixture Fraction
MeshDisplacement	Arbitrary Mesh Displacement

An example of the equation system definition for ABL precursor simulations is shown below:

```
# Equation systems example for ABL precursor simulations
systems:
  - LowMachEOM:
      name: myLowMach
      max_iterations: 1
      convergence_tolerance: 1.0e-5
  - TurbKineticEnergy:
      name: myTke
      max_iterations: 1
      convergence_tolerance: 1.0e-2
  - Enthalpy:
      name: myEnth
      max_iterations: 1
      convergence_tolerance: 1.0e-2
```

Initial conditions

initial_conditions
The `initial_conditions` sub-sections defines the conditions used to initialize the computational fields if they are not provided via the mesh file. Two types of field initializations are currently possible:

- `constant` - Initialize the field with a constant value throughout the domain;

- `user_function` - Initialize the field with a pre-defined user function.

The snippet below shows an example of both options available to initialize the various computational fields used for ABL simulations. In this example, the pressure and turbulent kinetic energy fields are initialized using a constant value, whereas the velocity field is initialized by the user function `boundary_layer_perturbation` that imposes sinusoidal fluctations over a velocity field to trip the flow.

```
initial_conditions:
  - constant: ic_1
    target_name: [fluid_part]
    value:
      pressure: 0.0
      turbulent_ke: 0.1

  - user_function: ic_2
    target_name: [fluid_part]
    user_function_name:
      velocity: boundary_layer_perturbation
    user_function_parameters:
      velocity: [1.0,0.0075398,0.0075398,50.0,8.0]
```

`initial_conditions.constant`
This input parameter serves two purposes: 1. it indicates the *type* (`constant`), and 2. provides the custom *name* for this condition. In addition to the *`initial_conditions.target_name`* this section requires another entry `value` that contains the mapping of `(field_name, value)` as shown in the above example.

`initial_conditions.user_function`
Indicates that this block of YAML input must be parsed as input for a user defined function.

`initial_conditions.target_name`
A list of element blocks (*parts*) where this initial condition must be applied.

Boundary Conditions

`boundary_conditions`
This subsection of the physics Realm contains a list of boundary conditions that must be used during the simulation. Each entry of this list is a YAML mapping entry with the key of the form `<type>_boundary_condition` where the available types are:

- `inflow`
- `open` – Outflow BC
- `wall`
- `symmetry`
- `periodic`
- `non_conformal` – e.g., BC across sliding mesh interfaces
- `overset` – overset mesh assembly description

All BC types require *`bc.target_name`* that contains a list of side sets where the specified BC is to be applied. Additional information necessary for certain BC types are provided by a sub-dictionary with the key `<type>_user_data:` that contains the parameters necessary to initialize a specific BC type.

`bc.target_name`
A list of side set part names where the given BC type must be applied. If a single string value is provided, it is converted to a list internally during input file processing phase.

Inflow Boundary Condition

```
- inflow_boundary_condition: bc_inflow
  target_name: inlet
  inflow_user_data:
    velocity: [0.0,0.0,1.0]
```

Open Boundary Condition

```
- open_boundary_condition: bc_open
  target_name: outlet
  open_user_data:
    velocity: [0,0,0]
    pressure: 0.0
```

Wall Boundary Condition

bc.wall_user_data
: This subsection contains specifications as to whether wall models are used, or how to treat the velocity at the wall when there is mesh motion.

The following code snippet shows an example of using an ABL wall function at the terrain during ABL simulations. See *ABL Wall Function* for more details on the actual implementation.

```
# Wall boundary condition example for ABL terrain modeling
- wall_boundary_condition: bc_terrain
  target_name: terrain
  wall_user_data:
    velocity: [0,0,0]
    use_abl_wall_function: yes
    heat_flux: 0.0
    roughness_height: 0.2
    gravity_vector_component: 3
    reference_temperature: 300.0
```

When there is mesh motion involved the wall boundary must specify a user function to determine relative velocity at the surface.

```
# Wall boundary specification with mesh motion
- wall_boundary_condition: bc_cylinder
  target_name: cylinder_wall
  wall_user_data:
    user_function_name:
      velocity: wind_energy
    user_function_string_parameters:
      velocity: [cylinder]
```

The misnomer `wind_energy` is a pre-defined *user function* that provides the correct velocity at the wall accounting for relative mesh motion with respect to fluid and doesn't specifically deal with any wind energy simulation. The `user_function_string_parameters` contains a YAML mapping of fields, e.g. velocity, to the list of *names* provided in the `soln_opts.mesh_motion` entry in the `solution_options` section.

Example of wall boundary with a custom user function for temperature at the wall

```
- wall_boundary_condition: bc_6
  target_name: surface_6
  wall_user_data:
    user_function_name:
     temperature: steady_2d_thermal
```

Symmetry Boundary Condition

Requires no additional input other than *bc.target_name*.

```
- symmetry_boundary_condition: bc_top
   target_name: top
   symmetry_user_data:
```

Periodic Boundary Condition

Unlike the other BCs described so far, the parameter *bc.target_name* behaves differently for the periodic BC. This parameter must be a list containing exactly two entries: the boundary face pair where periodicity is enforced. The nodes on these planes must coincide after translation in the direction of periodicity. This BC also requires a *periodic_user_data* section that specifies the search tolerance for locating node pairs.

periodic_user_data

```
- periodic_boundary_condition: bc_east_west
    target_name: [east, west]
    periodic_user_data:
      search_tolerance: 0.0001
```

Non-Conformal Boundary

Like the periodic BC, the parameter *bc.target_name* must be a list with exactly two entries that specify the boundary plane pair forming the non-conformal boundary.

```
- non_conformal_boundary_condition: bc_left
  target_name: [surface_77, surface_7]
  non_conformal_user_data:
    expand_box_percentage: 10.0
```

Material Properties

material_properties

The section provides the properties required for various physical quantities during the simulation. A sample section used for simulating ABL flows is shown below

```
material_properties:
  target_name: [fluid_part]

  constant_specification:
   universal_gas_constant: 8314.4621
```

```
 reference_pressure: 101325.0

reference_quantities:
  - species_name: Air
    mw: 29.0
    mass_fraction: 1.0

specifications:
  - name: density
    type: constant
    value: 1.178037722969475
  - name: viscosity
    type: constant
    value: 1.6e-5
  - name: specific_heat
    type: constant
    value: 1000.0
```

material_properties.target_name
A list of element blocks (*parts*) where the material properties are applied. This list should ideally include all the parts that are referenced by *initial_conditions.target_name*.

material_properties.constant_specification
Values for several constants used during the simulation. Currently the following properties are defined:

Name	Description
universal_gas_constant	Ideal gas constant R
reference_temperature	Reference temperature for simulations
reference_pressure	Reference pressure for simulations

material_properties.reference_quantities
Provides material properties for the different species involved in the simulation.

Name	Description
species_name	Name used to lookup properties
mw	Molecular weight
mass_fraction	Mass fraction
primary_mass_fraction	
secondary_mass_fraction	
stoichiometry	

material_properties.specifications
A list of material properties with the following parameters

material_properties.specifications.name
The name used for lookup, e.g., density, viscosity, etc.

material_properties.specifications.type
The type can be one of the following

Type	Description
`constant`	Constant value property
`polynomial`	Property determined by a polynomial function
`ideal_gas_t`	Function of T_{ref}, P_{ref}, molecular weight
`ideal_gas_t_p`	Function of T_{ref}, pressure, molecular weight
`ideal_gas_yk`	
`hdf5table`	Lookup from an HDF5 table
`mixture_fraction`	Property determined by the mixture fraction
`geometric`	
`generic`	

Examples

1.Specification for density as a function of temperature

```
specifications:
   - name: density
     type: ideal_gas_t
```

2.Specification of viscosity as a function of temperature

```
- name: viscosity
  type: polynomial
  coefficient_declaration:
   - species_name: Air
     coefficients: [1.7894e-5, 273.11, 110.56]
```

The `species_name` must correspond to the entry in *`reference quantitites`* to lookup molecular weight information.

3.Specification via `hdf5table`

```
material_properties:
  table_file_name: SLFM_CGauss_C2H4_ZMean_ZScaledVarianceMean_logChiMean.h5

  specifications:
    - name: density
      type: hdf5table
      independent_variable_set: [mixture_fraction, scalar_variance, scalar_
↪dissipation]
      table_name_for_property: density
      table_name_for_independent_variable_set: [ZMean, ZScaledVarianceMean,␣
↪ChiMean]
      aux_variables: temperature
      table_name_for_aux_variables: temperature

    - name: viscosity
      type: hdf5table
      independent_variable_set: [mixture_fraction, scalar_variance, scalar_
↪dissipation]
      table_name_for_property: mu
      table_name_for_independent_variable_set: [ZMean, ZScaledVarianceMean,␣
↪ChiMean]
```

4.Specification via `mixture_fraction`

```
material_properties:
  target_name: block_1
```

```
specifications:
  - name: density
    type: mixture_fraction
    primary_value: 0.163e-3
    secondary_value: 1.18e-3
  - name: viscosity
    type: mixture_fraction
    primary_value: 1.967e-4
    secondary_value: 1.85e-4
```

Output Options

output
Specifies the frequency of output, the output database name, etc.

Example:

```
output:
  output_data_base_name: out/ABL.neutral.e
  output_frequency: 100
  output_node_set: no
  output_variables:
   - velocity
   - pressure
   - temperature
```

output.output_data_base_name
The name of the output Exodus-II database. Can specify a directory relative to the run directory, e.g., `out/nalu_results.e`. The directory will be created automatically if one doesn't exist. Default: `output.e`

output.output_frequency
Nalu will write the output file every `output_frequency` timesteps. Note that currently there is no option to output results at a specified simulation time. Default: `1`.

output.output_start
Nalu will start writing output past the `output_start` timestep. Default: `0`.

output.output_forced_wall_time
Force output at a specified *wall-clock time* in seconds.

output.output_node_set
Boolean flag indicating whether nodesets, if present, should be output to the output file along with element blocks.

output.compression_level
Integer value indicating the compression level used. Default: `0`.

output.output_variables
A list of field names to be output to the database. The field variables can be node or element based quantities.

Restart Options

restart
This section manages the restart for this realm object.

restart.restart_data_base_name
The filename for restart. Like *output*, the filename can contain a directory and it will be created if not already present.

restart.restart_time
If this variable is present, it indicates that the current run will restart from a previous simulation. This requires that the *mesh* be a restart file with all the fields necessary for the equation sets defined in the *equation_systems.systems*. Nalu will restart from the closest time available in the *mesh* to `restart_time`. The timesteps available in a restart file can be examined by looking at the `time_whole` variable using the `ncdump` utility.

Note: The restart database used for restarting a simulation is the *mesh* parameter. The *restart_data_base_name* parameter is used exclusively for outputs.

---

restart.restart_frequency
The frequency at which restart files are written to the disk. Default: `500` timesteps.

restart.restart_start
Nalu will write a restart file after `restart_start` timesteps have elapsed.

restart.restart_forced_wall_time
Force writing of restart file after specified *wall-clock time* in seconds.

restart.restart_node_set
A boolean flag indicating whether nodesets are output to the restart database.

restart.max_data_base_step_size
Default: `100,000`.

restart.compression_level
Compression level. Default: `0`.

Time-step Control Options

time_step_control
This optional section specifies the adpative time stepping parameters used if *time_int.time_stepping_type* is set to `adaptive`.

```
time_step_control:
  target_courant: 2.0
  time_step_change_factor: 1.2
```

dtctrl.target_courant
Maximum Courant number allowed during the simulation. Default: `1.0`

dtctrl.time_step_change_factor
Maximum allowable increase in `dt` over a given timestep.

Turbulence averaging

turbulence_averaging
`turbulence_averaging` subsection defines the turbulence post-processing quantities and averaging procedures. A sample section is shown below

```
turbulence_averaging:
  time_filter_interval: 100000.0

  specifications:

    - name: turbulence_postprocessing
      target_name: interior
      reynolds_averaged_variables:
        - velocity

      favre_averaged_variables:
        - velocity
        - resolved_turbulent_ke

      compute_tke: yes
      compute_reynolds_stress: yes
      compute_q_criterion: yes
      compute_vorticity: yes
      compute_lambda_ci: yes
```

Note: The variable in the *turbulence_averaging* subsection are prefixed with `turbulence_averaging.name` but only the variable name after the period should appear in the input file.

turbulence_averaging.time_filter_interval
Number indicating the time filter size over which calculate the running average. The current implementation of the running average in Nalu uses a "sawtooth" average. The running average is set to zero each time the time filter width is reached and a new average is calculated for the next time interval.

turbulence_averaging.specifications
A list of turbulence postprocessing properties with the following parameters

turbulence_averaging.specifications.name
The name used for lookup and logging.

turbulence_averaging.specifications.target_name
A list of element blocks (parts) where the turbulence averaging is applied.

turbulence_averaging.specifications.reynolds_average_variables
A list of field names to be averaged.

turbulence_averaging.specifications.favre_average_variables
A list of field names to be Favre averaged.

turbulence_averaging.specifications.compute_tke
A boolean flag indicating whether the turbulent kinetic energy is computed. The default value is `no`.

turbulence_averaging.specifications.compute_reynolds_stress
A boolean flag indicating whether the reynolds stress is computed. The default value is `no`.

turbulence_averaging.specifications.compute_favre_stress
A boolean flag indicating whether the Favre stress is computed. The default value is `no`.

turbulence_averaging.specifications.compute_favre_tke
A boolean flag indicating whether the Favre stress is computed. The default value is `no`.

turbulence_averaging.specifications.compute_q_criterion
A boolean flag indicating whether the q-criterion is computed. The default value is `no`.

turbulence_averaging.specifications.compute_vorticity
A boolean flag indicating whether the vorticity is computed. The default value is `no`.

turbulence_averaging.specifications.compute_lambda_ci
A boolean flag indicating whether the Lambda2 vorticity criterion is computed. The default value is `no`.

Data probes

data_probes
`data_probes` subsection defines the data probes. A sample section is shown below

```
data_probes:

  output_frequency: 100

  search_method: stk_octree
  search_tolerance: 1.0e-3
  search_expansion_factor: 2.0

  specifications:
    - name: probe_bottomwall
      from_target_part: bottomwall

      line_of_site_specifications:
        - name: probe_bottomwall
          number_of_points: 100
          tip_coordinates: [-6.39, 0.0, 0.0]
          tail_coordinates: [4.0, 0.0, 0.0]

      output_variables:
        - field_name: tau_wall
          field_size: 1
        - field_name: pressure

  specifications:
    - name: probe_profile
      from_target_part: interior

      line_of_site_specifications:
        - name: probe_profile
          number_of_points: 100
          tip_coordinates: [0, 0.0, 0.0]
          tail_coordinates: [0.0, 0.0, 1.0]

      output_variables:
        - field_name: velocity
          field_size: 3
        - field_name: reynolds_stress
          field_size: 6
```

Note: The variable in the *data_probes* subsection are prefixed with `data_probes.name` but only the variable name after the period should appear in the input file.

data_probes.output_frequency
Integer specifying the frequency of output.

data_probes.search_method
String specifying the search method for finding nodes to transfer field quantities to the data probe lineout.

data_probes.search_tolerance
Number specifying the search tolerance for locating nodes.

data_probes.search_expansion_factor
Number specifying the factor to use when expanding the node search.

data_probes.specifications
A list of data probe properties with the following parameters

data_probes.specifications.name
The name used for lookup and logging.

data_probes.specifications.from_target_part
A list of element blocks (parts) where to do the data probing.

data_probes.specifications.line_of_site_specifications
A list specifications defining the lineout

Parameter	Description
name	File name (without extension) for the data probe
number_of_points	Number of points along the lineout
tip_coordinates	List containing the coordinates for the start of the lineout
tail_coordinates	List containing the coordinates for the end of the lineout

data_probes.specifications.output_variables
A list of field names (and field size) to be probed.

Post-processing

post_processing
`post_processing` subsection defines the different post-processign options. A sample section is shown below

```
post_processing:

- type: surface
  physics: surface_force_and_moment
  output_file_name: results/wallHump.dat
  frequency: 100
  parameters: [0,0]
  target_name: bottomwall
```

Note: The variable in the *`post_processing`* subsection are prefixed with `post_processing.name` but only the variable name after the period should appear in the input file.

post_processing.type
Type of post-processing. Possible values are:

Value	Description
surface	Post-processing of surface quantities

post_processing.physics
Physics to be post-processing. Possible values are:

Value	Description
surface_force_and_moment	Calculate surface forces and moments
surface_force_and_moment_wall_function	Calculate surface forces and moments when using a wall function

post_processing.output_file_name
String specifying the output file name.

post_processing.frequency
Integer specifying the frequency of output.

post_processing.parameters
Parameters for the physics function. For the `surface_force_and_moment` type functions, this is a list specifying the centroid coordinates used in the moment calculation.

post_processing.target_name
A list of element blocks (parts) where to do the post-processing

Transfers

transfers
Transfers section describes the search and mapping operations to be performed between participating `realms` within a simulation.

Simulations

simulations
This is the top-level section that orchestrates the entire execution of Nalu.

Examples

Here we describe any examples we have for users to try running Nalu.

Tutorials

Here we describe any tutorials that may be further in-depth than examples.

CHAPTER 2

Developer Manual

Testing Nalu

Nalu's regression tests and unit tests are run nightly using the GCC and Intel compilers against the Trilinos master and development branches on a machine at NREL. The results can be seen at the CDash Nalu website.

Running Tests Locally

The nightly tests are implemented using CTest and these same tests are available to developers to run locally as well. Due to the nature of error propagation of calculations in computers, results of regression testing can be difficult to keep consistent. Output from Nalu can vary from established reference data for the regression tests based on the compiler you are using, the types of optimizations set, and the versions of the third-party libraries Nalu utilizes, along with the blas/lapack implementation in use. Therefore it may make sense when you checkout Nalu to create your own reference data for the tests for the machine and configuration you are using, which is described later in this document. Or you can use a lower tolerance when running the tests. At the moment, a single tolerance is chosen in which to use for all the tests. The following instructions will describe how to run Nalu's tests.

Since Nalu's tests require a large amount of data (meshes), this data is hosted in a separate repository from Nalu. This mesh repo is set as a submodule in the `reg_tests/mesh` directory in the main Nalu repository. Submodule repos are not checked out by default, so either use `git submodule init` and then `git submodule update` to clone it in your checkout of Nalu, or when you first clone Nalu you can also use `git clone --recursive <repo_url>` to checkout all submodules as well.

Once this submodule is intialized and cloned, you will need to configure Nalu with testing on. To configure Nalu with testing enabled, in Nalu's existing `build` directory, we will run this command:

```
cmake -DTrilinos_DIR:PATH=`spack location -i nalu-trilinos` \
      -DYAML_DIR:PATH=`spack location -i yaml-cpp` \
      -DENABLE_TESTS:BOOL=ON \
      ..
```

Note we have chosen to originally build Nalu with Spack in this case, hence the use of `spack location -i <package>` to locate our Yaml and Trilinos installations. Then we use `-DENABLE_TESTS:BOOL=ON` to enable

CTest. Once Nalu is configured, you should be able to run the tests by building Nalu in the `build` directory, and running `make test` or `ctest`. Looking at `ctest -h` will show you many ways you can run tests and choose which tests to run.

There are advantages to using CTest, such as being able to run subsets of the tests, or tests matching a particular regular expression for example. To do so, in the `build` directory, you can run `ctest -R femHC` to run the test matching the `femHC` regular expression. Other useful capabilities are `ctest --output-on-failure` to see test outputs when they fail, `ctest --rerun-failed` to only run the tests that previously failed, `ctest --print-labels` to see the test labels, and `ctest -L unit` to run the tests with label 'unit' for example. All testing related log files and output can be seen in `Nalu/build/Testing/Temporary` and `Nalu/build/reg_tests` after the test have been run.

To define your own tolerance for tests, at configure time, add `-DTEST_TOLERANCE=0.0001` for example to the Nalu CMake configure line.

Updating Reference Data for Your Machine

When running the tests, the norms for each test are the output and they are 'diffed' against the 'gold' norms that we have established for each test. To dictate whether or not a test passes, we use a chosen tolerance in which we allow the results to deviate from the 'gold' norm. As stated earlier, these 'gold' norms are not able to reflect every configuration of Nalu, per compiler, optimization, TPL versions, blas/lapack version, etc. This tolerance is currently defined in the `CMakeLists.txt` in Nalu's `reg_tests` directory. This tolerance can also be passed into Nalu at configure time using `-DTEST_TOLERANCE=0.0000001` for example. To update the 'gold' norms locally to your configuration, merely run the tests once, and copy the `*.norm` files in the `build/reg_tests/test_files` directory to the corresponding test location in `reg_tests/test_files` while overwriting the current 'gold' norms.

In regards to 'official' gold norms, Linux with GCC 4.9.2, netlib-blas/lapack, and the following TPL versions are officially tested:

```
openmpi@1.10.4
boost@1.60.0
cmake@3.6.1
parallel-netcdf@1.6.1
yaml-cpp@0.5.3
hdf5@1.8.16
netcdf@4.3.3.1
zlib@1.2.11
superlu@4.3
```

Adding Tests to Nalu

The testing infrastructure is almost completely confined to the `reg_tests` directory. To add a test to Nalu, we need to add the test name, and create a test directory to place the input files and gold norms for the test. First, the test itself can be added to the list of CTest tests by adding a line to the `CTestList.cmake` file. For a single regression test, provided it is similar to the categories shown at the top of the `CTestList.cmake` file, it can likely be added with a single line using the test name and amount of processes you would like to run the test with and choosing the correct function to use. For example:

```
add_test_r(mytest 6)
```

After this has been done, in the `reg_tests/test_files` directory, you should add a directory corresponding to your test name and include the input file, `mytest.i`, and reference output file `mytest.norm.gold`. If you are using an xml file that doesn't exist in the `xml` directory, you will need to commit that as well.

To see commands used when running the tests, see the functions at the top of the `CTestList.cmake` file. These functions ultimately create `CTestTestFile.cmake` files in the CMake build directory at configure time. You can see the exact commands used for each test after configure in the `build/reg_tests/CTestTestFile.cmake` file.

Note if your test doesn't conform to an existing archetype, a new function in `CTestList.cmake` may need to be created. Also, if you are using a mesh file that doesn't exist in the mesh repo, you will need to add it, and update the submodule in the Nalu main repo to use the latest commit of the mesh submodule repo.

Adding Testing Machines to CDash

To add a testing machine that will post results to CDash first means that you should have all software dependencies satisified for Nalu. Next the script located at CTestNightlyScript.cmake can be run for example as:

```
ctest \
  -DNIGHTLY_DIR=${NALU_TESTING_DIR} \
  -DYAML_DIR=${YAML_INSTALL_DIR} \
  -DTRILINOS_DIR=${TRILINOS_INSTALL_DIR} \
  -DHOST_NAME=machine.domain.com \
  -DEXTRA_BUILD_NAME=Linux-gcc-whatever \
  -VV -S ${NALU_DIR}/reg_tests/CTestNightlyScript.cmake
```

In this case `${NALU_TESTING_DIR}` is one directory above where the Nalu repo has been checked out. This runs CTest in scripting mode with verbosity on and it will update the Nalu repo with the latest revisions, configure, build, test, and finally submit results to the CDash site. Since CTest does the building, it needs to know the locations of Yaml and Trilinos. For examples of nightly testing, refer to the testing scripts currently being run here.

Source Code Documentation

The source documentation is extracted from the C++ files using Doxygen.

Simulation – Nalu Top-level Interface

class `sierra::nalu::`**`Simulation`**

Realms

Realm is a Nalu abstraction of a set of equations that are solved on a computational domain, reresented by an Exodus-II mesh. A simulation can contain multiple Realms and that can interact via *`sierra::nalu::Transfer`* instance. *`InputOutputRealm`* is a special type of Realm that exists solely to provide data (input) or extract a subset of data from another *`Realm`*.

class `sierra::nalu::`**`Realm`**

Representation of a computational domain and physics equations solved on this domain.

Subclassed by *sierra::nalu::InputOutputRealm*

Public Functions

void **`check_job`**(bool *get_node_count*)

check job for fitting in memory

class sierra::nalu::**InputOutputRealm**
Inherits from *sierra::nalu::Realm*

class sierra::nalu::**Realms**

Time Integration

class sierra::nalu::**TimeIntegrator**

Linear Solver Interface

class sierra::nalu::**LinearSystem**
Subclassed by *sierra::nalu::TpetraLinearSystem*

Public Functions

virtual void **resetRows**(std::vector<stk::mesh::Entity> *nodeList*, **const** unsigned *beginPos*, **const** unsigned *endPos*) = 0
Reset LHS and RHS for the given set of nodes to 0.

Parameters

- nodeList: A list of STK node entities whose rows are zeroed out
- beginPos: Starting index (usually 0)
- endPos: Terminating index (1 for scalar quantities; nDim for vectors)

class sierra::nalu::**LinearSolver**
Subclassed by sierra::nalu::TpetraLinearSolver

class sierra::nalu::**TpetraLinearSystem**
Inherits from *sierra::nalu::LinearSystem*

Public Functions

virtual void **resetRows**(**const** std::vector<stk::mesh::Entity> *nodeList*, **const** unsigned *beginPos*, **const** unsigned *endPos*)
Reset LHS and RHS for the given set of nodes to 0.

Parameters

- nodeList: A list of STK node entities whose rows are zeroed out
- beginPos: Starting index (usually 0)
- endPos: Terminating index (1 for scalar quantities; nDim for vectors)

Transfers

class sierra::nalu::**Transfer**

class sierra::nalu::**Transfers**

Equation Systems

class `sierra::nalu::`**`EquationSystem`**
Base class representation of a PDE.

EquationSystem defines the API supported by all concrete implementations of PDEs for performing the following actions:

- Register computational fields
- Register computational algorithms for interior domain and boundary conditions
- Manage solve and update of the PDE for a given timestep

Subclassed by *sierra::nalu::ContinuityEquationSystem*, *sierra::nalu::EnthalpyEquationSystem*, *sierra::nalu::HeatCondEquationSystem*, *sierra::nalu::LowMachEquationSystem*, *sierra::nalu::MassFractionEquationSystem*, sierra::nalu::MeshDisplacementEquationSystem, *sierra::nalu::MixtureFractionEquationSystem*, *sierra::nalu::MomentumEquationSystem*, *sierra::nalu::ProjectedNodalGradientEquationSystem*, sierra::nalu::RadiativeTransportEquationSystem, *sierra::nalu::ShearStressTransportEquationSystem*, *sierra::nalu::SpecificDissipationRateEquationSystem*, *sierra::nalu::TurbKineticEnergyEquationSystem*

Public Functions

virtual void **`solve_and_update`**`()`
Assemble the LHS and RHS and perform linear solve for prescribed number of iterations.

This method is invoked in *EquationSystems::solve_and_update* method as shown below

```
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();
  eqsys->solve_and_update();              //<<<< Assemble and solve system
  eqsys->post_iter_work();
}
post_iter_work();
```

See *EquationSystems::solve_and_update*

virtual void **`pre_iter_work`**`()`
Perform setup tasks before entering the solve and update step.

This method is invoked in *EquationSystems::solve_and_update* method as shown below

```
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();                 //<<<< Pre-iteration setup
  eqsys->solve_and_update();
  eqsys->post_iter_work();
}
post_iter_work();
```

See *EquationSystems::solve_and_update*

virtual void **post_iter_work**()
Perform setup tasks after he solve and update step.

This method is invoked in *EquationSystems::solve_and_update* method as shown below

```
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();
  eqsys->solve_and_update();
  eqsys->post_iter_work();                //<<<< Post-iteration actions
}
post_iter_work();
```

See *EquationSystems::solve_and_update*

virtual void **post_iter_work_dep**()
Deprecated post iteration work logic.

Public Members

std::vector<AlgorithmDriver *> **preIterAlgDriver_**
List of tasks to be performed before each *EquationSystem::solve_and_update*.

std::vector<AlgorithmDriver *> **postIterAlgDriver_**
List of tasks to be performed after each *EquationSystem::solve_and_update*.

class sierra::nalu::**LowMachEquationSystem**
Low-Mach formulation of the Navier-Stokes Equations.

This class is a thin-wrapper around *sierra::nalu::ContinuityEquationSystem* and *sierra::nalu::MomentumEquationSystem* that orchestrates the interactions between the velocity and the pressure Possion solves in the *LowMachEquationSystem::solve_and_update* method.

Inherits from *sierra::nalu::EquationSystem*

Public Functions

virtual void **pre_iter_work**()
Perform setup tasks before entering the solve and update step.

This method is invoked in *EquationSystems::solve_and_update* method as shown below

```
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();                 //<<<< Pre-iteration setup
  eqsys->solve_and_update();
  eqsys->post_iter_work();
}
post_iter_work();
```

See *EquationSystems::solve_and_update*

virtual void **solve_and_update**()
Assemble the LHS and RHS and perform linear solve for prescribed number of iterations.

This method is invoked in *EquationSystems::solve_and_update* method as shown below

```
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();
  eqsys->solve_and_update();             //<<<< Assemble and solve system
  eqsys->post_iter_work();
}
post_iter_work();
```

See *EquationSystems::solve_and_update*

class sierra::nalu::**EnthalpyEquationSystem**
Inherits from *sierra::nalu::EquationSystem*

Public Functions

void **solve_and_update**()
Assemble the LHS and RHS and perform linear solve for prescribed number of iterations.

This method is invoked in *EquationSystems::solve_and_update* method as shown below

```
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();
  eqsys->solve_and_update();             //<<<< Assemble and solve system
  eqsys->post_iter_work();
}
post_iter_work();
```

See *EquationSystems::solve_and_update*

void **post_iter_work_dep**()
Deprecated post iteration work logic.

class sierra::nalu::**TurbKineticEnergyEquationSystem**
Inherits from *sierra::nalu::EquationSystem*

Public Functions

void **solve_and_update**()
Assemble the LHS and RHS and perform linear solve for prescribed number of iterations.

This method is invoked in *EquationSystems::solve_and_update* method as shown below

```
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();
```

```
  eqsys->solve_and_update();              //<<<< Assemble and solve system
  eqsys->post_iter_work();
}
post_iter_work();
```

See *EquationSystems::solve_and_update*

class sierra::nalu::**ShearStressTransportEquationSystem**
Inherits from *sierra::nalu::EquationSystem*

Public Functions

virtual void **solve_and_update**()
Assemble the LHS and RHS and perform linear solve for prescribed number of iterations.

This method is invoked in *EquationSystems::solve_and_update* method as shown below

```
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();
  eqsys->solve_and_update();              //<<<< Assemble and solve system
  eqsys->post_iter_work();
}
post_iter_work();
```

See *EquationSystems::solve_and_update*

class sierra::nalu::**HeatCondEquationSystem**
Inherits from *sierra::nalu::EquationSystem*

Public Functions

void **solve_and_update**()
Assemble the LHS and RHS and perform linear solve for prescribed number of iterations.

This method is invoked in *EquationSystems::solve_and_update* method as shown below

```
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();
  eqsys->solve_and_update();              //<<<< Assemble and solve system
  eqsys->post_iter_work();
}
post_iter_work();
```

See *EquationSystems::solve_and_update*

class sierra::nalu::**MassFractionEquationSystem**
Inherits from *sierra::nalu::EquationSystem*

Public Functions

void **solve_and_update**()
Assemble the LHS and RHS and perform linear solve for prescribed number of iterations.

This method is invoked in *EquationSystems::solve_and_update* method as shown below

```
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();
  eqsys->solve_and_update();              //<<<< Assemble and solve system
  eqsys->post_iter_work();
}
post_iter_work();
```

See *EquationSystems::solve_and_update*

class sierra::nalu::**MixtureFractionEquationSystem**
Inherits from *sierra::nalu::EquationSystem*

Public Functions

void **solve_and_update**()
Assemble the LHS and RHS and perform linear solve for prescribed number of iterations.

This method is invoked in *EquationSystems::solve_and_update* method as shown below

```
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();
  eqsys->solve_and_update();              //<<<< Assemble and solve system
  eqsys->post_iter_work();
}
post_iter_work();
```

See *EquationSystems::solve_and_update*

class sierra::nalu::**MomentumEquationSystem**
Representation of the Momentum conservation equations in 2-D and 3-D.

Inherits from *sierra::nalu::EquationSystem*

class sierra::nalu::**ContinuityEquationSystem**
Inherits from *sierra::nalu::EquationSystem*

class sierra::nalu::**SpecificDissipationRateEquationSystem**
Inherits from *sierra::nalu::EquationSystem*

class sierra::nalu::**ProjectedNodalGradientEquationSystem**
Inherits from *sierra::nalu::EquationSystem*

Public Functions

void **solve_and_update**()

Assemble the LHS and RHS and perform linear solve for prescribed number of iterations.

This method is invoked in *EquationSystems::solve_and_update* method as shown below

```
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();
  eqsys->solve_and_update();              //<<<< Assemble and solve system
  eqsys->post_iter_work();
}
post_iter_work();
```

See *EquationSystems::solve_and_update*

class sierra::nalu::**EquationSystems**

A collection of Equations to be solved on a *Realm*.

EquationSystems holds a vector of *EquationSystem* instances representing the equations that are being solved in a given *Realm* and is responsible for the management of the solve and update of the various field quantities in a given timestep.

See *EquationSystems::solve_and_update*

Public Functions

bool **solve_and_update**()

Solve and update the state of all variables for a given timestep.

This method is responsible for executing setup actions before calling solve, performing the actual solve, updating the solution, and performing post-solve actions after the solution has been updated. To provide sufficient granularity and control of this pre- and post- solve actions, the solve method uses the following series of steps:

```
// Perform tasks for this timestep before any Equation system is called
pre_iter_work();
// Iterate over all equation systems
for (auto eqsys: equationSystems_) {
  eqsys->pre_iter_work();
  eqsys->solve_and_update();
  eqsys->post_iter_work();
}
// Perform tasks after all equation systems have updated
post_iter_work();
```

Tasks that require to be performed before any equation system is solved for needs to be registered to preIterAlgDriver_ on *EquationSystems*, similiary for post-solve tasks. And actions to be performed immediately before individual equation system solves need to be registered in *EquationSystem::preIterAlgDriver_*.

See *pre_iter_work()*, *post_iter_work()*, *EquationSystem::pre_iter_work()*,

See *EquationSystem::post_iter_work()*

void **pre_iter_work**()
Perform necessary setup tasks that affect all *EquationSystem* instances at a given timestep.

See *EquationSystems::solve_and_update()*

void **post_iter_work**()
Perform necessary actions once all *EquationSystem* instances have been updated for the prescribed number of *outer iterations* at a given timestep.

See *EquationSystems::solve_and_update()*

Public Members

std::vector<AlgorithmDriver *> **preIterAlgDriver_**
A list of tasks to be performed before all *EquationSystem::solve_and_update*.

std::vector<AlgorithmDriver *> **postIterAlgDriver_**
A list of tasks to be performed after all *EquationSystem::solve_and_update*.

CVFEM and FEM Interface

class sierra::nalu::**MasterElement**
Subclassed by sierra::nalu::Edge2DSCS, *sierra::nalu::Hex8FEM*, sierra::nalu::HexahedralP2Element, *sierra::nalu::HexSCS*, *sierra::nalu::HexSCV*, sierra::nalu::HigherOrderEdge2DSCS, *sierra::nalu::HigherOrderHexSCS*, *sierra::nalu::HigherOrderHexSCV*, *sierra::nalu::HigherOrderQuad2DSCS*, *sierra::nalu::HigherOrderQuad2DSCV*, sierra::nalu::HigherOrderQuad3DSCS, *sierra::nalu::PyrSCS*, *sierra::nalu::PyrSCV*, *sierra::nalu::Quad3DSCS*, *sierra::nalu::Quad42DSCS*, *sierra::nalu::Quad42DSCV*, sierra::nalu::QuadrilateralP2Element, *sierra::nalu::TetSCS*, *sierra::nalu::TetSCV*, sierra::nalu::Tri2DSCV, *sierra::nalu::Tri32DSCS*, *sierra::nalu::Tri32DSCV*, *sierra::nalu::Tri3DSCS*, *sierra::nalu::WedSCS*, *sierra::nalu::WedSCV*

3-D Topologies

class sierra::nalu::**HexSCV**
Inherits from *sierra::nalu::MasterElement*

class sierra::nalu::**HexSCS**
Inherits from *sierra::nalu::MasterElement*

class sierra::nalu::**TetSCV**
Inherits from *sierra::nalu::MasterElement*

class sierra::nalu::**TetSCS**
Inherits from *sierra::nalu::MasterElement*

class sierra::nalu::**PyrSCV**
Inherits from *sierra::nalu::MasterElement*

class sierra::nalu::**PyrSCS**
Inherits from *sierra::nalu::MasterElement*

class sierra::nalu::**WedSCV**
Inherits from *sierra::nalu::MasterElement*

class `sierra::nalu::`**WedSCS**
Inherits from *sierra::nalu::MasterElement*

class `sierra::nalu::`**Hex27SCV**
Inherits from sierra::nalu::HexahedralP2Element

class `sierra::nalu::`**Hex27SCS**
Inherits from sierra::nalu::HexahedralP2Element

class `sierra::nalu::`**Hex8FEM**
Inherits from *sierra::nalu::MasterElement*

class `sierra::nalu::`**Quad3DSCS**
Inherits from *sierra::nalu::MasterElement*

class `sierra::nalu::`**Quad93DSCS**
Inherits from sierra::nalu::HexahedralP2Element

class `sierra::nalu::`**Tri3DSCS**
Inherits from *sierra::nalu::MasterElement*

2-D Topologies

class `sierra::nalu::`**Quad42DSCV**
Inherits from *sierra::nalu::MasterElement*

class `sierra::nalu::`**Quad42DSCS**
Inherits from *sierra::nalu::MasterElement*

class `sierra::nalu::`**Tri32DSCV**
Inherits from *sierra::nalu::MasterElement*

class `sierra::nalu::`**Tri32DSCS**
Inherits from *sierra::nalu::MasterElement*

Higher-order Element Topologies

class `sierra::nalu::`**HigherOrderHexSCV**
Inherits from *sierra::nalu::MasterElement*

class `sierra::nalu::`**HigherOrderHexSCS**
Inherits from *sierra::nalu::MasterElement*

class `sierra::nalu::`**HigherOrderQuad2DSCV**
Inherits from *sierra::nalu::MasterElement*

class `sierra::nalu::`**HigherOrderQuad2DSCS**
Inherits from *sierra::nalu::MasterElement*

Auxiliary Functions

class `sierra::nalu::`**AuxFunction**
Subclassed by *sierra::nalu::BoundaryLayerPerturbationAuxFunction*, sierra::nalu::ConstantAuxFunction, *sierra::nalu::ConvectingTaylorVortexPressureAuxFunction*, *sierra::nalu::ConvectingTaylorVortexPressureGradAuxFunction*, *sierra::nalu::ConvectingTaylorVortexVelocityAuxFunction*, sierra::nalu::FlowPastCylinderTempAuxFunction, *sierra::nalu::KovasznayPressureAuxFunction*, *sierra::nalu::KovasznayPressureGradientAuxFunction*, *sierra::nalu::KovasznayVelocityAuxFunction*, *sierra::nalu::LinearRampMeshDisplacementAuxFunction*, sierra::nalu::RayleighTaylorMixFracAuxFunction, *sierra::nalu::SinMeshDisplacementAuxFunction*,

sierra::nalu::SinProfileChannelFlowVelocityAuxFunction, *sierra::nalu::SteadyTaylorVortexGradPressureAuxFunction*, *sierra::nalu::SteadyTaylorVortexPressureAuxFunction*, *sierra::nalu::SteadyTaylorVortexVelocityAuxFunction*, *sierra::nalu::SteadyThermal3dContactAuxFunction*, *sierra::nalu::SteadyThermal3dContactDtDxAuxFunction*, *sierra::nalu::SteadyThermalContactAuxFunction*, sierra::nalu::TaylorGreenPressureAuxFunction, sierra::nalu::TaylorGreenVelocityAuxFunction, sierra::nalu::TornadoAuxFunction, sierra::nalu::VariableDensityMixFracAuxFunction, sierra::nalu::VariableDensityNonIsoTemperatureAuxFunction, sierra::nalu::VariableDensityPressureAuxFunction, sierra::nalu::VariableDensityVelocityAuxFunction, *sierra::nalu::WindEnergyAuxFunction*, sierra::nalu::WindEnergyTaylorVortexAuxFunction, sierra::nalu::WindEnergyTaylorVortexPressureAuxFunction, sierra::nalu::WindEnergyTaylorVortexPressureGradAuxFunction

ABL Utilities

class `sierra::nalu::`**`BoundaryLayerPerturbationAuxFunction`**
Add sinusoidal perturbations to the velocity field.

This function is used as an initial condition, primarily in Atmospheric Boundary Layer (ABL) flows, to trigger transition to turbulent flow during ABL precursor simulations.

Inherits from *sierra::nalu::AuxFunction*

Steady Taylor Vortex

class `sierra::nalu::`**`SteadyTaylorVortexVelocityAuxFunction`**
Inherits from *sierra::nalu::AuxFunction*

class `sierra::nalu::`**`SteadyTaylorVortexPressureAuxFunction`**
Inherits from *sierra::nalu::AuxFunction*

class `sierra::nalu::`**`SteadyTaylorVortexGradPressureAuxFunction`**
Inherits from *sierra::nalu::AuxFunction*

class `sierra::nalu::`**`SteadyTaylorVortexMomentumSrcElemSuppAlg`**
Inherits from sierra::nalu::SupplementalAlgorithm

class `sierra::nalu::`**`SteadyTaylorVortexMomentumSrcNodeSuppAlg`**
Inherits from sierra::nalu::SupplementalAlgorithm

Convecting Taylor Vortex

class `sierra::nalu::`**`ConvectingTaylorVortexVelocityAuxFunction`**
Inherits from *sierra::nalu::AuxFunction*

class `sierra::nalu::`**`ConvectingTaylorVortexPressureAuxFunction`**
Inherits from *sierra::nalu::AuxFunction*

class `sierra::nalu::`**`ConvectingTaylorVortexPressureGradAuxFunction`**
Inherits from *sierra::nalu::AuxFunction*

Kovasznay 2-D Flow

class `sierra::nalu::`**`KovasznayVelocityAuxFunction`**
Inherits from *sierra::nalu::AuxFunction*

class `sierra::nalu::`**`KovasznayPressureAuxFunction`**
Inherits from *sierra::nalu::AuxFunction*

class sierra::nalu::**KovasznayPressureGradientAuxFunction**
Inherits from *sierra::nalu::AuxFunction*

Steady Thermal MMS (2-D and 3-D)

class sierra::nalu::**SteadyThermal3dContactAuxFunction**
Inherits from *sierra::nalu::AuxFunction*

class sierra::nalu::**SteadyThermal3dContactDtDxAuxFunction**
Inherits from *sierra::nalu::AuxFunction*

template <typename AlgTraits>
class sierra::nalu::**SteadyThermal3dContactSrcElemKernel**
Inherits from sierra::nalu::Kernel

Public Functions

virtual void **execute**(SharedMemView<DoubleType **>&, SharedMemView<DoubleType *>&, ScratchViews<DoubleType>&)
Execute the kernel within a Kokkos loop and populate the LHS and RHS for the linear solve.

class sierra::nalu::**SteadyThermal3dContactSrcElemSuppAlgDep**
Inherits from sierra::nalu::SupplementalAlgorithm

class sierra::nalu::**SteadyThermalContact3DSrcNodeSuppAlg**
Inherits from sierra::nalu::SupplementalAlgorithm

class sierra::nalu::**SteadyThermalContactAuxFunction**
Inherits from *sierra::nalu::AuxFunction*

class sierra::nalu::**SteadyThermalContactSrcElemSuppAlg**
Inherits from sierra::nalu::SupplementalAlgorithm

class sierra::nalu::**SteadyThermalContactSrcNodeSuppAlg**
Inherits from sierra::nalu::SupplementalAlgorithm

Mesh Motion/Displacement Utilities

class sierra::nalu::**LinearRampMeshDisplacementAuxFunction**
Inherits from *sierra::nalu::AuxFunction*

class sierra::nalu::**SinMeshDisplacementAuxFunction**
Inherits from *sierra::nalu::AuxFunction*

class sierra::nalu::**WindEnergyAuxFunction**
Inherits from *sierra::nalu::AuxFunction*

Post-Processing Utilities

class sierra::nalu::**TurbulenceAveragingPostProcessing**

class sierra::nalu::**DataProbePostProcessing**

class sierra::nalu::**SolutionNormPostProcessing**

class sierra::nalu::**SurfaceForceAndMomentAlgorithm**
Inherits from sierra::nalu::Algorithm

class sierra::nalu::**SurfaceForceAndMomentWallFunctionAlgorithm**
Inherits from sierra::nalu::Algorithm

Writing Developer Documentation

Developer documentation should be written using Doxygen annotations directly in the source code. This allows the documentation to live with the code essentially as comments that Doxygen is able to extract automatically into a more human readable form. Doxygen requires special syntax markers to indicate comments that should be processed as inline documentation vs. generic comments in the source code. The Doxygen manual provides detailed information on the various markers available to tailor the formatting of auto-generated documentation. It is recommended that users document the classes and methods in the header file. A sample header file with specially formatted comments is shown below. You can download a copy of the file.

Listing 2.1: Sample C++ header file showing inline documentation using specially formatted comments.

```
/** @file example.h
 *  @brief Brief description of a documented file.
 *
 *  Longer description of a documented file.
*/

/** Here is a brief description of the example class.
 *
 * This is a more in-depth description of the class.
 * This class is meant as an example.
 * It is not useful by itself, rather its usefulness is only a
 * function of how much it helps the reader. It is in a sense
 * defined by the person who reads it and otherwise does
 * not exist in any real form.
 *
 * @note This is a note.
 *
 */

#ifndef EXAMPLECLASS_H
#define EXAMPLECLASS_H

class ExampleClass
{

public:

    /// Create an ExampleClass.
    ExampleClass();

    /** Create an ExampleClass with lot's of intial values.
     *
     * @param a This is a description of parameter a.
     * @param b This is a description of parameter b.
     *
     * The distance between \f$(x_1,y_1)\f$ and \f$(x_2,y_2)\f$ is
     * \f$\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}\f$.
     */
    ExampleClass(int a, float b);

    /** ExampleClass destructor description.
```

```
     */
    ~ExampleClass();

    /// This method does something.
    void DoSomething();

    /**
      * This is a method that does so
      * much that I must write an epic
      * novel just to describe how much
      * it truly does.
      */
    void DoNothing();

    /** Brief description of a useful method.
      * @param level An integer setting how useful to be.
      * @return Description of the output.
      *
      * This method does unbelievably useful things.
      * And returns exceptionally useful results.
      * Use it everyday with good health.
      * \f[
      *    |I_2|=\left| \int_{0}^T \psi(t)
      *        \left\{
      *           u(a,t)-
      *           \int_{\gamma(t)}^a
      *           \frac{d\theta}{k(\theta,t)}
      *           \int_{a}^\theta c(\xi)u_t(\xi,t)\,d\xi
      *        \right\} dt
      *     \right|
      * \f]
      */
    void* VeryUsefulMethod(bool level);

    /** Brief description of a useful method.
      * @param level An integer setting how useful to be.
      * @return Description of the output.
      *
      * - Item 1
      *
      *   More text for this item.
      *
      * - Item 2
      *   + nested list item.
      *   + another nested item.
      * - Item 3
      *
      * # Markdown Example
      * [Here is a link.](http://www.google.com/)
      */
    void* AnotherMethod(bool level);

protected:
    /** The protected methods can be documented and extracted too.
     *
     */
    void SomeProtectedMethod();
```

```
private:

    const char* fQuestion; ///< The question
    int fAnswer;           ///< The answer

}; // End of class ExampleClass

#endif // EXAMPLE_H
```

Once processed by Doxygen and embedded in Sphinx, the resulting documentation of the class looks as shown below:

class ExampleClass

Here is a brief description of the example class.

This is a more in-depth description of the class. This class is meant as an example. It is not useful by itself, rather its usefulness is only a function of how much it helps the reader. It is in a sense defined by the person who reads it and otherwise does not exist in any real form.

Note This is a note.

Public Functions

ExampleClass()

Create an *ExampleClass*.

ExampleClass(int *a*, float *b*)

Create an *ExampleClass* with lot's of intial values.

The distance between (x_1, y_1) and (x_2, y_2) is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.

Parameters

- a: This is a description of parameter a.
- b: This is a description of parameter b.

~ExampleClass()

ExampleClass destructor description.

void **DoSomething**()

This method does something.

void **DoNothing**()

This is a method that does so much that I must write an epic novel just to describe how much it truly does.

void ***VeryUsefulMethod**(bool *level*)

Brief description of a useful method.

This method does unbelievably useful things. And returns exceptionally useful results. Use it everyday with good health.

$$|I_2| = \left| \int_0^T \psi(t) \left\{ u(a,t) - \int_{\gamma(t)}^{a} \frac{d\theta}{k(\theta,t)} \int_a^{\theta} c(\xi) u_t(\xi,t)\, d\xi \right\} dt \right|$$

Return Description of the output.

Parameters

- `level`: An integer setting how useful to be.

void ***AnotherMethod**(bool *level*)
Brief description of a useful method.

•Item 1

More text for this item.

•Item 2

–nested list item.

–another nested item.

•Item 3

Return Description of the output.

Parameters

- `level`: An integer setting how useful to be.

Markdown Example

Here is a link.

Protected Functions

void **SomeProtectedMethod**()
The protected methods can be documented and extracted too.

Private Members

const char ***fQuestion**
The question.

int **fAnswer**
The answer.

Writing User Documentation

This documentation is written in Sphinx and is generated automatically on the http://nalu.readthedocs.io website every time the Nalu Github repo is updated. This documentation can also be built locally on your machine by using the instructions here. Sphinx uses restructured text (RST) to generate the documentation in many other formats, such as this html version. Refer to the primer on writing restructured text here.

Building the Documentation

This document describes how to build Nalu's documentation. The documentation is based on the use of Doxygen, Sphinx, and Doxylink. Therefore we will need to install these tools as well as some extensions of Sphinx that are utilized.

Install the Tools

Install CMake, Doxygen, Sphinx, Doxylink, and the extensions used. Doxygen uses the `dot` application installed with GraphViz. Sphinx uses a combination of extensions installed with `pip install` as well as some that come with Nalu located in the `_extensions` directory. Using Homebrew on Mac OS X, this would look something like:

```
brew install cmake
brew install python
brew install doxygen
brew install graphviz
pip2 install sphinx
pip2 install sphinxcontrib-bibtex
pip2 install breathe
pip2 install sphinx_rtd_theme
```

On Linux, CMake, Python, Doxygen, and GraphViz could be installed using your package manager, e.g. `sudo apt-get install cmake`.

Run CMake Configure

In the Nalu repository checkout, create your own or use the `build` directory that already exists in the repo. Change to your designated build directory and run CMake with `-DENABLE_DOCUMENTATION` on. For example:

```
cmake -DTrilinos_DIR:PATH=$(spack location -i nalu-trilinos) \
      -DYAML_DIR:PATH=$(spack location -i yaml-cpp) \
      -DCMAKE_BUILD_TYPE=RELEASE \
      -DENABLE_DOCUMENTATION:BOOL=ON \
      ..
```

If all of the main tools are found successfully, CMake should configure with the ability to build the documentation. If Sphinx or Doxygen aren't found, the configure will skip the documentation.

Make the Docs

In your designated build directory, issue the command `make docs` which should first build the Doxygen documentation and then the Sphinx documentation. If this completes successfully, the entry point to the documentation should be in `build/docs/html/index.html`.

Developer Workflow

This document describes a suggested developer workflow for Nalu.

Nalu Style Guide

1. No tabs. Remove them from your editor. Better yet, use eclipse and follow the xml style. Use the format here.
2. Use underscores for private data, e.g., `const double thePrivateData_`.
3. Use camel case for data members and classes unless it is silly (you get the idea).
4. Camel case on Class names always; non camel case for methods, e.g.,

```
const double Realm::get_me() {
  return hereIAm_; // hmmm... silly? your call
}
```

5. Use `const` when possible, however, do not try to be a member of the 'const' police force.
6. We need logic to launch some special physics. Try to avoid run time logic by designing with polymorphic/templates.
7. When possible, add classes that manage loading, field registration, setup and execute, e.g., SolutionNormPostProcessing, etc.

Contributing to Nalu

1. There is no rush to push. We only support production tested capability. Better yet, peform code verification and unit testing.
2. Always run the full regression test suite. No exceptions.
3. Peer review when fully appropriate (ask for a pull request).
4. If adding a new feature, include a regression test for this feature. Refer to the section of this documentation on adding a test *here*.

CHAPTER 3

Sierra Low Mach Module: Nalu - Theory Manual

The SIERRA Low Mach Module: Nalu (henceforth referred to as Nalu), developed at Sandia National Labs, represents a generalized unstructured, massively parallel, variable density turbulent flow capability designed for energy applications. This code base began as an effort to prototype Sierra Toolkit, *[EWS+10]*, usage along with direct parallel matrix assembly to the Trilinos, *[HBH+03]*, Epetra and Tpetra data structure. However, the simulation tool has evolved as a tool to support a variety of research projects germane to the energy sector including wind aerodynamic prediction and traditional gas-phase combustion applications.

Low Mach Number Derivation

The low Mach number equations are a subset of the fully compressible equations of motion (momentum, continuity and energy), admitting large variations in gas density while remaining acoustically incompressible. The low Mach number equations are preferred over the full compressible equations for low speed flow problems as the accoustics are of little consequence to the overall simulation accuracy. The technique avoids the need to resolve fast-moving acoustic signals. Derivations of the low Mach number equations can be found in found in Rehm and Baum, *[RB78]*, or Paolucci, *[Pao82]*.

The equations are derived from the compressible equations using a perturbation expansion in terms of the lower limit of the Mach number squared; hence the name. The asymptotic expansion leads to a splitting of pressure into a spatially constant thermodynamic pressure and a locally varying dynamic pressure. The dynamic pressure is decoupled from the thermodynamic state and cannot propagate acoustic waves. The thermodynamic pressure is used in the equation of state and to determine thermophysical properties. The thermodynamic pressure can vary in time and can be calculated using a global energy balance.

Asymptotic Expansion

The asymptotic expansion for the low Mach number equations begins with the full compressible equations in Cartesian coordinates. The equations are the minimum set required to propagate acoustic waves. The equations are written in

divergence form using Einstein notation (summation over repeated indices):

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_j}{\partial x_j} &= 0, \\
\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_j u_i}{\partial x_j} + \frac{\partial P}{\partial x_i} &= \frac{\partial \tau_{ij}}{\partial x_j} + \rho g_i, \\
\frac{\partial \rho E}{\partial t} + \frac{\partial \rho u_j H}{\partial x_j} &= -\frac{\partial q_j}{\partial x_j} + \frac{\partial u_i \tau_{ij}}{\partial x_j} + \rho u_i g_i.
\end{aligned}$$

The primitive variables are the velocity components, u_i, the pressure, P, and the temperature T. The viscous shear stress tensor is τ_{ij}, the heat conduction is q_i, the total enthalpy is H, the total internal energy is E, the density is ρ, and the gravity vector is g_i. The total internal energy and total enthalpy contain the kinetic energy contributions. The equations are closed using the following models and definitions:

$$\begin{aligned}
P &= \rho \frac{R}{W} T, \\
E &= H - P/\rho, \\
H &= h + \frac{1}{2} u_k u_k, \\
\tau_{ij} &= \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \mu \frac{\partial u_k}{\partial x_k} \delta_{ij}, \\
q_i &= -k \frac{\partial T}{\partial x_i}
\end{aligned}$$

The mean molecular weight of the gas is W, the molecular viscosity is μ, and the thermal conductivity is k. A Newtonian fluid is assumed along with the Stokes hypothesis for the stress tensor.

The equations are scaled so that the variables are all of order one. The velocities, lengths, and times are nondimensionalized by a characteristic velocity, U_∞, and a length scale, L. The pressure, density, and temperature are nondimensionalized by P_∞, ρ_∞, and T_∞. The enthalpy and energy are nondimensionalized by $C_{p,\infty} T_\infty$. Dimensionless variables are noted by overbars. The dimensionless equations are:

$$\begin{aligned}
\frac{\partial \bar{\rho}}{\partial \bar{t}} + \frac{\partial \bar{\rho} \bar{u}_j}{\partial \bar{x}_j} &= 0, \\
\frac{\partial \bar{\rho} \bar{u}_i}{\partial \bar{t}} + \frac{\partial \bar{\rho} \bar{u}_j \bar{u}_i}{\partial \bar{x}_j} + \frac{1}{\gamma \mathrm{Ma}^2} \frac{\partial \bar{P}}{\partial \bar{x}_i} &= \frac{1}{\mathrm{Re}} \frac{\partial \bar{\tau}_{ij}}{\partial \bar{x}_j} + \frac{1}{\mathrm{Fr}_i} \bar{\rho}, \\
\frac{\partial \bar{\rho} \bar{h}}{\partial \bar{t}} + \frac{\partial \bar{\rho} \bar{u}_j \bar{h}}{\partial \bar{x}_j} &= -\frac{1}{\mathrm{Pr}} \frac{1}{\mathrm{Re}} \frac{\partial \bar{q}_j}{\partial \bar{x}_j} + \frac{\gamma - 1}{\gamma} \frac{\partial \bar{P}}{\partial \bar{t}} \\
&\quad + \frac{\gamma - 1}{\gamma} \frac{\mathrm{Ma}^2}{\mathrm{Re}} \frac{\partial \bar{u}_i \bar{\tau}_{ij}}{\partial \bar{x}_j} + \bar{\rho} \bar{u}_i \frac{\gamma - 1}{\gamma} \frac{\mathrm{Ma}^2}{\mathrm{Fr}_i} \\
&\quad - \frac{\gamma - 1}{2} \mathrm{Ma}^2 \left(\frac{\partial \bar{\rho} \bar{u}_k \bar{u}_k}{\partial \bar{t}} + \frac{\partial \bar{\rho} \bar{u}_j \bar{u}_k \bar{u}_k}{\partial \bar{x}_j} \right).
\end{aligned}$$

The groupings of characteristic scaling terms are:

$$\begin{aligned}
\mathrm{Re} &= \frac{\rho_\infty U_\infty L}{\mu_\infty}, && \text{Reynoldsnumber}, \\
\mathrm{Pr} &= \frac{C_{p,\infty} \mu_\infty}{k_\infty}, && \text{Prandtlnumber}, \\
\mathrm{Fr}_i &= \frac{u_\infty^2}{g_i L}, && \text{Froudenumber}, \quad g_i \neq 0, \\
\mathrm{Ma} &= \sqrt{\frac{u_\infty^2}{\gamma R T_\infty / W}}, && \text{Machnumber},
\end{aligned}$$

where γ is the ratio of specific heats.

For small Mach numbers, $\mathrm{Ma} \ll 1$, the kinetic energy, viscous work, and gravity work terms can be neglected in the energy equation since those terms are scaled by the square of the Mach number. The inverse of Mach number squared remains in the momentum equations, suggesting singular behavior. In order to explore the singularity, the pressure, velocity and temperature are expanded as asymptotic series in terms of the parameter ϵ:

$$\bar{P} = \bar{P}_0 + \bar{P}_1\epsilon + \bar{P}_2\epsilon^2 \dots$$
$$\bar{u}_i = \bar{u}_{i,0} + \bar{u}_{i,1}\epsilon + \bar{u}_{i,2}\epsilon^2 \dots$$
$$\bar{T} = \bar{T}_0 + \bar{T}_1\epsilon + \bar{T}_2\epsilon^2 \dots$$

The zeroeth-order terms are collected together in each of the equations. The form of the continuity equation stays the same. The gradient of the pressure in the zeroeth-order momentum equations can become singular since it is divided by the characteristic Mach number squared. In order for the zeroeth-order momentum equations to remain well-behaved, the spatial variation of the $\bar{P}_0$ term must be zero. If the magnitude of the expansion parameter is selected to be proportional to the square of the characteristic Mach number, $\epsilon = \gamma \mathrm{Ma}^2$, then the $\bar{P}_1$ term can be included in the zeroeth-order momentum equation.

$$\frac{1}{\gamma \mathrm{Ma}^2}\frac{\partial \bar{P}}{\partial x_i} = \frac{\partial}{\partial x_i}\left(\frac{1}{\gamma \mathrm{Ma}^2}\bar{P}_0 + \frac{\epsilon}{\gamma \mathrm{Ma}^2}\bar{P}_1 + \dots\right) = \frac{\partial}{\partial x_i}\left(\bar{P}_1 + \epsilon\bar{P}_2 + \dots \right)$$

The form of the energy equation remains the same, less the kinetic energy, viscous work and gravity work terms. The P_0 term remains in the energy equation as a time derivative. The low Mach number equations are the zeroeth-order equations in the expansion including the P_1 term in the momentum equations. The expansion results in two different types of pressure and they are considered to be split into a thermodynamic component and a dynamic component. The thermodynamic pressure is constant in space, but can change in time. The thermodynamic pressure is used in the equation of state. The dynamic pressure only arises as a gradient term in the momentum equation and acts to enforce continuity. The unsplit dimensional pressure is

$$P = P_{th} + \gamma \mathrm{Ma}^2 P_1,$$

where the dynamic pressure, $p = P - P_{th}$, is related to a pressure coefficient

$$\bar{P}_1 = \frac{P - P_{th}}{\rho_\infty u_\infty^2} P_{th}.$$

The resulting unscaled low Mach number equations are:

$$\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_j}{\partial x_j} = 0,$$
$$\frac{\partial \rho u_i}{\partial t} + \frac{\partial \rho u_j u_i}{\partial x_j} + \frac{\partial P}{\partial x_i} = \frac{\partial \tau_{ij}}{\partial x_j} + (\rho - \rho_\circ)\, g_i,$$
$$\frac{\partial \rho h}{\partial t} + \frac{\partial \rho u_j h}{\partial x_j} = -\frac{\partial q_j}{\partial x_j} + \frac{\partial P_{th}}{\partial t},$$

where the ideal gas law becomes

$$P_{th} = \rho \frac{R}{W} T.$$

The hydrostatic pressure gradient has been subtracted from the momentum equation, assuming an ambient density of $\rho_\circ$. The stress tensor and heat conduction remain the same as in the original equations.

Supported Equation Set

This section provides an overview of the currently supported equation sets. Equations will be decribed in integral form with assumed Favre averaging. However, the laminar counterparts are supported in the code base and are obtain in the user file by ommitting a turbulence model specification.

Conservation of Mass

The continuity equation is always solved in the variable density form.

$$\int \frac{\partial \bar{\rho}}{\partial t}\, dV + \int \bar{\rho}\tilde{u}_i n_i\, dS = 0$$

Since Nalu uses equal-order interpolation (variables are collocated) stabilization is required. The stabilization choice will be developed in the pressure stabilization section.

Note that the use of a low speed compressible formulation requires that the fluid density be computed by an equation of state that uses the thermodynamic pressure. This thermodynamic pressure can either be computed based on a global energy/mass balance or allowed to be spatially varying. By modification of the continuity density time derivative to include the $\frac{\partial \rho}{\partial p}$ sensitivity, an equation that admits acoustic pressure waves is realized.

Conservation of Momentum

The integral form of the Favre-filtered momentum equations used for turbulent transport are

$$\begin{aligned}\int \frac{\partial \bar{\rho}\tilde{u}_i}{\partial t}\mathrm{d}V + \int \bar{\rho}\tilde{u}_i\tilde{u}_j n_j \mathrm{d}S = \int \tilde{\sigma}_{ij} n_j \mathrm{d}S - \int \tau_{ij}^{sgs} n_j \mathrm{d}S \\ + \int \left(\bar{\rho} - \rho_\circ\right) g_i \mathrm{d}V,\end{aligned} \tag{3.1}$$

where the subgrid scale turbulent stress τ_{ij}^{sgs} is defined as

$$\tau_{ij}^{sgs} \equiv \bar{\rho}(\widetilde{u_i u_j} - \tilde{u}_i\tilde{u}_j). \tag{3.2}$$

The Cauchy stress is provided by,

$$\sigma_{ij} = 2\mu\tilde{S}_{ij}^{*} - \bar{P}\delta_{ij}$$

and the traceless rate-of-strain tensor defined as follows:

$$\begin{aligned}\tilde{S}_{ij}^{*} &= \tilde{S}_{ij} - \frac{1}{3}\delta_{ij}\tilde{S}_{kk} \\ &= \tilde{S}_{ij} - \frac{1}{3}\frac{\partial \tilde{u}_k}{\partial x_k}\delta_{ij}.\end{aligned}$$

In a low Mach flow, as described in the low Mach theory section, the above pressure, $\bar{P}$ is the purturbation about the thermodynamic pressure, P^{th}. In a low speed compressible flow, i.e., flows confined to a closed domain with energy or mass addition in which the continuity equation has been modifed to accomodate accoustics, this pressure is interpreted at the thermodynamic pressure itself.

For LES, τ_{ij}^{sgs} that appears in Equation (3.1) and defined in Equation (3.2) represents the subgrid stress tensor that must be closed. The deviatoric part of the subgrid stress tensor is defined as

$$\tau_{ij}^{sgs} = \tau_{ij}^{sgs} - \frac{1}{3}\delta_{ij}\tau_{kk}^{sgs} \tag{3.3}$$

where the subgrid turbulent kinetic energy is defined as $\tau_{kk}^{sgs} = 2\bar{\rho}k$. Note that here, k represents the modeled turbulent kinetic energy as is formally defined as,

$$\bar{\rho}k = \frac{1}{2}\bar{\rho}(\widetilde{u_k u_k} - \tilde{u}_k\tilde{u}_k).$$

Model closures can use, Yoshikawa's approach when k is not transported:

$$\tau_{kk}^{sgs} = 2C_I\bar{\rho}\Delta^2|\tilde{S}|^2.$$

Above, $|\tilde{S}| = \sqrt{2\tilde{S}_{ij}\tilde{S}_{ij}}$.

For low Mach-number flows, a vast majority of the turbulent kinetic energy is contained at resolved scales. For this reason, the subgrid turbulent kinetic energy is not directly treated and, rather, is included in the pressure as an additional normal stress. The Favre-filtered momentum equations then become

$$\begin{aligned} &\int \frac{\partial \bar{\rho}\tilde{u}_i}{\partial t}\mathrm{d}V + \int \bar{\rho}\tilde{u}_i\tilde{u}_j n_j \mathrm{d}S + \int \left(\bar{P} + \frac{2}{3}\bar{\rho}k\right) n_i \mathrm{d}S = \\ &\int 2(\mu+\mu_t)\left(\tilde{S}_{ij} - \frac{1}{3}\tilde{S}_{kk}\delta_{ij}\right) n_j \mathrm{d}S + \int \left(\bar{\rho}-\rho_\circ\right) g_i \mathrm{d}V, \end{aligned} \tag{3.4}$$

where LES closure models for the subgrid turbulent eddy viscosity μ_t are either the constant coefficient Smagorinsky, WALE or the constant coefficient k_{sgs} model (see the turbulence section).

Earth Coriolis Force

For simulation of large-scale atmospheric flows, the following Coriolis force term can be added to the right-hand-side of the momentum equation ((3.1)):

$$\int -2\bar{\rho}\epsilon_{ijk}\Omega_j u_k \ \mathrm{d}V. \tag{3.5}$$

Here, Ω is the Earth's angular velocity vector, and ϵ_{ijk} is the Levi-Civita symbol denoting the cross product of the Earth's angular velocity with the local fluid velocity vector. Consider an "East-North-Up" coordinate system on the Earth's surface, with the domain centered on a latitude angle ϕ (changes in latitude within the computational domain are neglected). In this coordinate system, the integrand of (cor-term), or the Coriolis acceleration vector, is

$$2\bar{\rho}\omega \begin{bmatrix} u_n \sin\phi - u_u \cos\phi \\ -u_e \sin\phi \\ u_e \cos\phi \end{bmatrix}, \tag{3.6}$$

where $\omega \equiv ||\Omega||$. Often, in geophysical flows it is assumed that the vertical component of velocity is small and that the vertical component of the acceleration is small relative to gravity, such that the terms containing $\cos\phi$ are neglected. However, there is evidence that this so-called traditional approximation is not valid for some mesoscale atmospheric phenomena cite{Gerkema_etal:08}, and so the full Coriolis term is retained in Nalu. The implementation proceeds by first finding the velocity vector in the East-North-Up coordinate system, then calculating the Coriolis acceleration vector ((3.6)), then transforming this vector back to the model $x-y-z$ coordinate system. The coordinate transformations are made using user-supplied North and East unit vectors given in the model coordinate system.

ABL Forcing Source Terms

In LES simulations of wind plant atmospheric flows, it is often necessary to drive the flow a predetermined vertical velocity and/or temperature profile. In Nalu, this is achieved by adding appropriate source terms $\mathbf{S}_u$ to the momentum equations ((3.1)). The present implementation can vary the source terms as a function of time and space using either a user-defined table of previously computed source terms (e.g., from a *precursor* simulation or another model such as WRF), or compute the source term as a function of the transient flow solution using the following equation:

$$\mathbf{S}_u^n = \alpha_u \bar{\rho}\left(\frac{\mathbf{U}_{\mathrm{ref}}^n - \langle \mathbf{u}^n \rangle}{\Delta t^n}\right) \tag{3.7}$$

where $\langle \mathbf{u}^n \rangle$ is the horizontally averaged velocity at a given height and instance in time $t = t_n$, $\mathbf{U}_{\mathrm{ref}}^n$ are the desired velocities at the corresponding heights and time. The implementation allows the user to prescribe relaxation factors α_u for the source terms that are applied. Nalu uses a default value of 1.0 for the relaxation factors if no values are defined in the input file during initialization.

Filtered Mixture Fraction

The optional quantity used to identify the chemical state is the mixture fraction, Z. While there are many different definitions of the mixture fraction that have subtle variations that attempt to capture effects like differential diffusion, they can all be interpreted as a local mass fraction of the chemical elements that originated in the fuel stream. The mixture fraction is a conserved scalar that varies between zero in the secondary stream and unity in the primary stream and is transported in laminar flow by the equation,

$$\frac{\partial \rho Z}{\partial t} + \frac{\partial \rho u_j Z}{\partial x_j} = \frac{\partial}{\partial x_j}\left(\rho D \frac{\partial Z}{\partial x_j}\right), \tag{3.8}$$

where D is an effective molecular mass diffusivity.

Applying either temporal Favre filtering for RANS-based treatments or spatial Favre filtering for LES-based treatments yields

$$\int \frac{\partial \bar{\rho}\tilde{Z}}{\partial t}\mathrm{d}V + \int \bar{\rho}\tilde{u}_j\tilde{Z}n_j\mathrm{d}S = -\int \tau^{sgs}_{Z,j}n_j\mathrm{d}S + \int \bar{\rho}D\frac{\partial \tilde{Z}}{\partial x_j}n_j\mathrm{d}S, \tag{3.9}$$

where sub-filter correlations have been neglected in the molecular diffusive flux vector and the turbulent diffusive flux vector is defined as

$$\tau^{sgs}_{Z,j} \equiv \bar{\rho}\left(\widetilde{Zu_j} - \tilde{Z}\tilde{u}_j\right).$$

This subgrid scale closure is modeled using the gradient diffusion hypothesis,

$$\tau^{sgs}_{Z,j} = -\bar{\rho}D_t\frac{\partial Z}{\partial x_j},$$

where D_t is the turbulent mass diffusivity, modeled as $\bar{\rho}D_t = \mu_t/\mathrm{Sc}_t$ where μ_t is the modeled turbulent viscosity from momentum transport and Sc_t is the turbulent Schmidt number. The molecular mass diffusivity is then expressed similarly as $\bar{\rho}D = \mu/\mathrm{Sc}$ so that the final modeled form of the filtered mixture fraction transport equation is

$$\frac{\partial \bar{\rho}\tilde{Z}}{\partial t} + \frac{\partial \bar{\rho}\tilde{u}_j\tilde{Z}}{\partial x_j} = \frac{\partial}{\partial x_j}\left[\left(\frac{\mu}{\mathrm{Sc}} + \frac{\mu_t}{\mathrm{Sc}_t}\right)\frac{\partial \tilde{Z}}{\partial x_j}\right].$$

In integral form the mixture fraction transport equation is

$$\int \frac{\partial \bar{\rho}\tilde{Z}}{\partial t}\,dV + \int \bar{\rho}\tilde{u}_j\tilde{Z}n_j\,dS = \int \left(\frac{\mu}{\mathrm{Sc}} + \frac{\mu_t}{\mathrm{Sc}_t}\right)\frac{\partial \tilde{Z}}{\partial x_j}n_j\,dS.$$

Conservation of Energy

The integral form of the Favre-filtered static enthalpy energy equation used for turbulent transport is

$$\begin{aligned}\int \frac{\partial \bar{\rho}\tilde{h}}{\partial t}\mathrm{d}V + \int \bar{\rho}\tilde{h}\tilde{u}_j n_j\mathrm{d}S = &-\int \bar{q}_j n_j\mathrm{d}S - \int \tau^{sgs}_{h,j}n_j\mathrm{d}S - \int \frac{\partial \overline{q^r_i}}{\partial x_i}\mathrm{d}V \\ &+ \int\left(\frac{\partial \bar{P}}{\partial t} + \tilde{u}_j\frac{\partial \bar{P}}{\partial x_j}\right)\mathrm{d}V + \int \overline{\tau_{ij}\frac{\partial u_i}{\partial x_j}}\mathrm{d}V.\end{aligned} \tag{3.10}$$

The above equation is derived by starting with the total internal energy equation, subtracting the mechanical energy equation and enforcing the variable density continuity equation. Note that the above equation includes possible source terms due to thermal radiatitive transport, viscous dissipation, and pressure work.

The simple Fickian diffusion velocity approximation, Equation (3.19), is assumed, so that the mean diffusive heat flux vector $\bar{q}_j$ is

$$\bar{q}_j = -\overline{\left[\frac{\kappa}{C_p}\frac{\partial h}{\partial x_j} - \frac{\mu}{\mathrm{Pr}}\sum_{k=1}^{K} h_k \frac{\partial Y_k}{\partial x_j}\right]} - \overline{\frac{\mu}{\mathrm{Sc}}\sum_{k=1}^{K} h_k \frac{\partial Y_k}{\partial x_j}}.$$

If $Sc = Pr$, i.e., unity Lewis number ($Le = 1$), then the diffusive heat flux vector simplifies to $\bar{q}_j = -\frac{\mu}{\mathrm{Pr}}\frac{\partial \tilde{h}}{\partial x_j}$. In the code base, the user has the ability to either specify a laminar Prandtl number, which is a constant, or provide a property evaluator for thermal conductivity. Inclusion of a Prandtl number prevails and ensures that the thermal conductivity is computed base on $\kappa = \frac{C_p \mu}{Pr}$. The viscous dissipation term is closed by

$$\begin{aligned}\overline{\tau_{ij}\frac{\partial u_i}{\partial x_j}} &= \left((\mu+\mu_t)\left(\frac{\partial \tilde{u}_i}{\partial x_j}+\frac{\partial \tilde{u}_j}{\partial x_i}\right) - \frac{2}{3}\left(\bar{\rho}\tilde{k}+\mu_t\frac{\partial \tilde{u}_k}{\partial x_k}\right)\delta_{ij}\right)\frac{\partial \tilde{u}_i}{\partial x_j} \\ &= \left[2\mu\tilde{S}_{ij} + 2\mu_t\left(\tilde{S}_{ij} - \frac{1}{3}\tilde{S}_{kk}\delta_{ij}\right) - \frac{2}{3}\bar{\rho}\tilde{k}\delta_{ij}\right]\frac{\partial \tilde{u}_i}{\partial x_j}.\end{aligned}$$

The subgrid scale turbulent flux vector τ_h^{sgs} in Equation (3.10) is defined as

$$\tau_{hu_j} \equiv \bar{\rho}\left(\widetilde{hu_j} - \tilde{h}\tilde{u}_j\right).$$

As with species transport, the gradient diffusion hypothesis is used to close this subgrid scale model,

$$\tau_{h,j}^{sgs} = -\frac{\mu_t}{\mathrm{Pr}_t}\frac{\partial \tilde{h}}{\partial x_j},$$

where Pr_t is the turbulent Prandtl number and μ_t is the modeled turbulent eddy viscosity from momentum closure. The resulting filtered and modeled turbulent energy equation is given by,

$$\begin{aligned}\int \frac{\partial \bar{\rho}\tilde{h}}{\partial t}\mathrm{d}V + \int \bar{\rho}\tilde{h}\tilde{u}_j n_j \mathrm{d}S = &\int\left(\frac{\mu}{\mathrm{Pr}}+\frac{\mu_t}{\mathrm{Pr}_t}\right)\frac{\partial \tilde{h}}{\partial x_j}n_j\mathrm{d}S - \int\frac{\partial \bar{q}_i^r}{\partial x_i}\mathrm{d}V \\ &+ \int\left(\frac{\partial \bar{P}}{\partial t} + \tilde{u}_j\frac{\partial \bar{P}}{\partial x_j}\right)\mathrm{d}V + \int\overline{\tau_{ij}\frac{\partial u_j}{\partial x_j}}\mathrm{d}V.\end{aligned} \tag{3.11}$$

The turbulent Prandtl number must have the same value as the turbulent Schmidt number for species transport to maintain unity Lewis number.

Review of Prandtl, Schmidt and Unity Lewis Number

For situations where a single diffusion coefficient is applicable (e.g., a binary gas system) the Lewis number is defined as:

$$\mathrm{Le} = \frac{\mathrm{Sc}}{\mathrm{Pr}} = \frac{\alpha}{D}. \tag{3.12}$$

If the diffusion rates of energy and mass are equal,

$$\mathrm{Sc} = \mathrm{Pr} \text{ and } \mathrm{Le} = 1. \tag{3.13}$$

For completeness, the thermal diffusivity, Prandtl and Schmidt number are defined by,

$$\alpha = \frac{\kappa}{\rho c_p}, \tag{3.14}$$

$$\mathrm{Pr} = \frac{c_p\mu}{\kappa} = \frac{\mu}{\rho\alpha}, \tag{3.15}$$

and

$$\mathrm{Sc} = \frac{\mu}{\rho D}, \tag{3.16}$$

where c_p is the specific heat, κ, is the thermal conductivity and α is the thermal diffusivity.

Thermal Heat Conduction

For non-isothermal object response that may occur in conjugate heat transfer applications, a simple single material heat conduction equation is supported.

$$\int \rho C_p \frac{\partial T}{\partial t} \mathrm{d}V + \int q_j n_j \mathrm{d}S = \int S \mathrm{d}V. \tag{3.17}$$

where q_j is again the energy flux vector, however, now in the following temperature form:

$$q_j = -\kappa \frac{\partial T}{\partial x_j}.$$

Conservation of Species

The integral form of the Favre-filtered species equation used for turbulent transport is

$$\int \frac{\partial \bar{\rho} \tilde{Y}_k}{\partial t} \mathrm{d}V + \int \bar{\rho} \tilde{Y}_k \tilde{u}_j n_j \mathrm{d}S = -\int \tau^{sgs}_{Y_k,j} n_j \mathrm{d}S - \int \overline{\rho Y_k \hat{u}_{j,k}} n_j \mathrm{d}S + \int \overline{\dot{\omega}_k} \mathrm{d}V, \tag{3.18}$$

where the form of diffusion velocities (see Equation (3.19)) assumes the Fickian approximation with a constant value of diffusion velocity for consistency with the turbulent form of the energy equation, Equation (3.10). The simplest form is Fickian diffusion with the same value of mass diffusivity for all species,

$$\hat{u}_{j,k} = -D \frac{1}{Y_k} \frac{\partial Y_k}{\partial x_j}. \tag{3.19}$$

The subgrid scale turbulent diffusive flux vector $\tau^{sgs}_{Y_k j}$ is defined as

$$\tau^{sgs}_{Y_k,j} \equiv \bar{\rho} \left(\widetilde{Y_k u_j} - \tilde{Y}_k \tilde{u}_j \right).$$

The closure for this model takes on its usual gradient diffusion hypothesis, i.e.,

$$\tau^{sgs}_{Y_k,j} = -\frac{\mu_t}{\mathrm{Sc}_t} \frac{\partial \tilde{Y}_k}{\partial x_j},$$

where Sc_t is the turbulent Schmidt number for all species and μ_t is the modeled turbulent eddy viscosity from momentum closure.

The Favre-filtered and modeled turbulent species transport equation is,

$$\int \frac{\partial \bar{\rho} \tilde{Y}_k}{\partial t} \mathrm{d}V + \int \bar{\rho} \tilde{Y}_k \tilde{u}_j n_j \mathrm{d}S = \int \left(\frac{\mu}{\mathrm{Sc}} + \frac{\mu_t}{\mathrm{Sc}_t} \right) \frac{\partial \tilde{Y}_k}{\partial x_j} n_j \mathrm{d}S + \int \overline{\dot{\omega}}_k \mathrm{d}V. \tag{3.20}$$

If transporting both energy and species equations, the laminar Prandtl number must be equal to the laminar Schmidt number and the turbulent Prandtl number must be equal to the turbulent Schmidt number to maintain unity Lewis number. Although there is a species conservation equation for each species in a mixture of n species, only $n-1$ species equations need to be solved since the mass fractions sum to unity and

$$\tilde{Y}_n = 1 - \sum_{j \neq n}^{n} \tilde{Y}_j.$$

Finally, the reaction rate source term is expected to be added based on an operator split approach wherebye the set of ODEs are integrated over the full time step. The chemical kinetic source terms can be sub-integrated within a time step using a stiff ODE integrator package.

The following system of ODEs are numerically integrated over a time step Δt for a fixed temperature and pressure starting from the initial values of gas phase mass fraction and density,

$$\dot{Y}_k = \frac{\dot{\omega}_k\left(Y_k\right)}{\rho}\ .$$

The sources for the sub-integration are computed with the composition and density at the new time level which are used to approximate a mean production rate for the time step

$$\dot{\omega}_k \approx \frac{\rho^* Y_k^* - \rho Y_k}{\Delta t}\ .$$

Subgrid-Scale Kinetic Energy One-Equation LES Model

The subgrid scale kinetic energy one-equation turbulence model, or k^{sgs} model, *[Dav97]*, represents a simple LES closure model. The transport equation for subgrid turbulent kinetic energy is given by

$$\int \frac{\partial \bar{\rho} k^{\mathrm{sgs}}}{\partial t}\mathrm{d}V + \int \bar{\rho} k^{\mathrm{sgs}} \tilde{u}_j n_j \mathrm{d}S = \int \frac{\mu_t}{\sigma_k}\frac{\partial k^{\mathrm{sgs}}}{\partial x_j} n_j \mathrm{d}S + \int \left(P_k^{\mathrm{sgs}} - D_k^{\mathrm{sgs}}\right)\mathrm{d}V. \tag{3.21}$$

The production of subgrid turbulent kinetic energy, P_k^{sgs}, is modeled by,

$$P_k \equiv -\overline{\rho u_i'' u_j''}\frac{\partial \tilde{u}_i}{\partial x_j}, \tag{3.22}$$

while the dissipation of turbulent kinetic energy, D_k^{sgs}, is given by

$$D_k^{\mathrm{sgs}} = \rho C_\epsilon \frac{k^{\mathrm{sgs}\,\frac{3}{2}}}{\Delta},$$

where the grid filter length, Δ, is given in terms of the grid cell volume by

$$\Delta = V^{\frac{1}{3}}.$$

The subgrid turbulent eddy viscosity is then provided by

$$\mu_t = C_{\mu_\epsilon}\Delta k^{\mathrm{sgs}\,\frac{1}{2}},$$

where the values of C_ϵ and C_{μ_ϵ} are 0.845 and 0.0856, respectively.

For simulations in which a buoyancy source term is desired, the code supports the Rodi form,

$$P_b = \beta \frac{\mu^T}{Pr} g_i \frac{\partial T}{\partial x_i}.$$

Shear Stress Transport (SST) RANS Model Suite

Although Nalu is primarily expected to be a LES simulation tool, RANS modeling is supported through the activation of the SST equation set.

It has been observed that standard 1998 $k-\omega$ models display a strong sensitivity to the free stream value of ω (see Mentor, *[MKL03]*). To remedy, this, an alternative set of transport equations have been used that are based on smoothly blending the $k-\omega$ model near a wall with $k-\epsilon$ away from the wall. Because of the relationship between ω and ϵ, the transport equations for turbulent kinetic energy and dissipation can be transformed into equations involving k and ω. Aside from constants, the transport equation for k is unchanged. However, an additional cross-diffusion term is

present in the ω equation. Blending is introduced by using smoothing which is a function of the distance from the wall, $F(y)$. The transport equations for the Mentor 2003 model are then

$$\int \frac{\partial \bar{\rho} k}{\partial t} \mathrm{d}V + \int \bar{\rho} k \tilde{u}_j n_j \mathrm{d}S = \int (\mu + \hat{\sigma}_k \mu_t) \frac{\partial k}{\partial x_j} n_j + \int (P_k^\omega - \beta^* \bar{\rho} k \omega) \, \mathrm{d}V,$$

$$\int \frac{\partial \bar{\rho} \omega}{\partial t} \mathrm{d}V + \int \bar{\rho} \omega \tilde{u}_j n_j \mathrm{d}S = \int (\mu + \hat{\sigma}_\omega \mu_t) \frac{\partial \omega}{\partial x_j} n_j + \int 2(1-F) \frac{\bar{\rho} \sigma_{\omega 2}}{\omega} \frac{\partial k}{\partial x_j} \frac{\partial \omega}{\partial x_j} \mathrm{d}V + \int \left(\frac{\hat{\gamma}}{\nu_t} P_k^\omega - \hat{\beta} \bar{\rho} \omega^2 \right) \mathrm{d}V.$$

The model coefficients, $\hat{\sigma}_k$, $\hat{\sigma}_\omega$, $\hat{\gamma}$ and $\hat{\beta}$ must also be blended, which is represented by

$$\hat{\phi} = F \phi_1 + (1 - F) \phi_2.$$

where $\sigma_{k1} = 0.85$, $\sigma_{k2} = 1.0$, $\sigma_{\omega 1} = 0.5$, $\sigma_{\omega 2} = 0.856$, $\gamma_1 = \frac{5}{9}$, $\gamma_2 = 0.44$, $\beta_1 = 0.075$ and $\beta_2 = 0.0828$. The blending function is given by

$$F = \tanh(arg_1^4),$$

where

$$arg_1 = \min \left(\max \left(\frac{\sqrt{k}}{\beta^* \omega y}, \frac{500 \mu}{\bar{\rho} y^2 \omega} \right), \frac{4 \bar{\rho} \sigma_{\omega 2} k}{C D_{k\omega} y^2} \right).$$

The final parameter is

$$CD_{k\omega} = \max \left(2 \bar{\rho} \sigma_{\omega 2} \frac{1}{\omega} \frac{\partial k}{\partial x_j} \frac{\partial \omega}{\partial x_j}, 10^{-10} \right).$$

An important component of the SST model is the different expression used for the turbulent viscosity,

$$\mu_t = \frac{a_1 \bar{\rho} k}{\max(a_1 \omega, S F_2)},$$

where F_2 is another blending function given by

$$F_2 = \tanh(arg_2^2).$$

The final parameter is

$$arg_2 = \max \left(\frac{2\sqrt{k}}{\beta^* \omega y}, \frac{500 \mu}{\bar{\rho} \omega y^2} \right).$$

Direct Eddy Simulation (DES) Formulation

The DES technique is also supported in the code base when the SST model is activated. This model seeks to formally relax the RANS-based approach and allows for a theoretical basis to allow for transient flows. The method follows the method of Temporally Filtered NS formulation as decribed by Tieszen, *[TDB05]*.

The SST DES model simply changes the turbulent kinetic energy equation to include a new minimum scale that manipulates the dissipation term.

$$D_k = \frac{\rho k^{3/2}}{l_{DES}},$$

where l_{DES} is the min($l_{SST}, c_{DES} l_{DES}$). The constants are given by, $l_{SST} = \frac{k^{1/2}}{\beta^* \omega}$ and c_{DES} represents a blended set of DES constants: $c_{DES_1} = 0.78$ and $c_{DES_2} = 0.61$. The length scale, l_{DES} is the maximum edge length scale touching a given node.

Solid Stress

A fully implicit CVFEM (only) linear elastic equation is supported in the code base. This equation is either used for true solid stress prediction or for smoothing the mesh due to boundary mesh motion (either through fluid structure interaction (FSI) or prescribed mesh motion).

Consider the displacement for component i, u_i equation set,

$$\rho\frac{\partial^2 u_i}{\partial t^2} - \frac{\partial \sigma_{ij}}{\partial x_j} = F_i, \tag{3.23}$$

where the Cauchy stress tensor, σ_{ij} assuming Hooke's law is given by,

$$\sigma_{ij} = \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) + \lambda\frac{\partial u_k}{\partial x_k}\delta_{ij}. \tag{3.24}$$

Above, the so-called Lame coefficients, Lame's first parameter, λ (also known as the Lame modulus) and Lame's second parameter, μ (also known as the shear modulus) are provided as functions of the Young's modulus, E, and Poisson's ratio, ν; here shown in the context of a isotropic elastic material,

$$\mu = \frac{E}{2\left(1+\nu\right)}, \tag{3.25}$$

and

$$\lambda = \frac{E\nu}{\left(1+\nu\right)\left(1-2\nu\right)}. \tag{3.26}$$

Note that the above notation of u_i to represent displacement is with respect to the classic definition of current and model coordinates,

$$x_i = X_i + u_i \tag{3.27}$$

where x_i is the position, relative to the fixed, or previous position, X_i.

The above equations are solved for mesh displacements, u_i. The supplemental relationship for solid velocity, v_i is given by,

$$v_i = \frac{\partial u_i}{\partial t}. \tag{3.28}$$

Numerically, the velocity might be obtained by a backward Euler or BDF2 scheme,

$$v_i = \frac{\gamma_1 u_i^{n+1} + \gamma_2 u_i^n + \gamma_3 u_i^{n-1}}{\Delta t} \tag{3.29}$$

Moving Mesh

The code base supports three notions of moving mesh: 1) linear elastic equation system that computes the stress of a solid 2) solid body rotation mesh motion and 3) mesh deformation via an external source.

The linear elastic equation system is activated via the standard equation system approach. Properties for the solid are specified in the material block. Mesh motion is prescribed by the input file via the `mesh_motion` block. Here, it is assumed that the mesh motion is solid rotation. For fluid/structure interaction (FSI) a mesh smoothing scheme is used to propagate the surface mesh displacement obtained by the solids solve. Simple mesh smoothing is obtained via a linear elastic solve in which the so-called Lame constants are proportional to the inverse of the dual volume. This allows for boundary layer mesh locations to be stiff while free stream mesh elements to be soft.

Additional mesh motion terms are required for the Eulerian fluid mechanics solve. Using the geometric conservative law the time and advection source term for a general scalar ϕ can be written as:

$$\int \frac{\rho\phi}{\partial t}\, dV + \int \rho\phi\left(u_j - v_j\right) n_j\, dS + \int \rho\phi \frac{\partial v_k}{\partial x_j}\, dV, \tag{3.30}$$

where u_j is the fluid velocity and v_j is the mesh velocity. Mesh velocities and the mesh velocity spatial derivatives are provided by the mesh smoothing solve. Activating the external mesh deformation or mesh motion block will result in the velocity relative to mesh calculation in the advection terms. The line command for source term, "gcl" must be activated for each equation for the volume integral to be included in the set of PDE solves. Finally, transfers are expected between the physics. For example, the solids solve is to provide mesh displacements to the mesh smoothing realm. The mesh smoothing procedure provides the boundary velocity, mesh velocity and projected nodal gradients of the mesh velocity to the fluids realm. Finally, the fluids solve is to provide the surface force at the desired solids surface. Currently, the pressure is transfered from the fluids realm to the solids realm. The ideal view of FSI is to solve the solids pde at the half time step. As such, in time, the $P^{n+\frac{1}{2}}$ is expected. The `fsi_interface` input line command attribute is expected to be set at these unique surfaces. More advanced FSI coupling techniques are under development by a current academic partner.

Radiative Transport Equation

The spatial variation of the radiative intensity corresponding to a given direction and at a given wavelength within a radiatively participating material, $I(s)$, is governed by the Boltzmann transport equation. In general, the Boltzmann equation represents a balance between absorption, emission, out-scattering, and in-scattering of radiation at a point. For combustion applications, however, the steady form of the Boltzmann equation is appropriate since the transient term only becomes important on nanosecond time scales which is orders of magnitude shorter than the fastest chemical.

Experimental data shows that the radiative properties for heavily sooting, fuel-rich hydrocarbon diffusion flames ($10^{-4}\%$ to $10^{-6}\%$ soot by volume) are dominated by the soot phase and to a lesser extent by the gas phase. Since soot emits and absorbs radiation in a relatively constant spectrum, it is common to ignore wavelength effects when modeling radiative transport in these environments. Additionally, scattering from soot particles commonly generated by hydrocarbon flames is several orders of magnitude smaller that the absorption effect and may be neglected. Moreover, the phase function is rarely known. However, for situations in which the phase function can be approximated by the iso-tropic scattering assumption, i.e., an intensity for direction k has equal probability to be scattered in any direction l, the appropriate form of the Botzmann radiative transport is

$$s_i \frac{\partial}{\partial x_i} I\left(s\right) + \left(\mu_a + \mu_s\right) I\left(s\right) = \frac{\mu_a \sigma T^4}{\pi} + \frac{\mu_s}{4\pi} G, \tag{3.31}$$

where μ_a is the absorption coeffiecient, μ_s is the scattering coefficeint, $I(s)$ is the intensity along the direction s_i, T is the temperature and the scalar flux is G. The black body radiation, I_b, is defined by $\frac{\sigma T^4}{\pi}$. Note that for situations in which the scattering coefficient is zero, the RTE reduces to a set of liniear, decoupled equations for each intensity to be solved.

The flux divergence may be written as a difference between the radiative emission and mean incident radiation at a point,

$$\frac{\partial q_i^r}{\partial x_i} = \mu_a \left[4\sigma T^4 - G\right], \tag{3.32}$$

where G is again the scalar flux. The flux divergence term is the same regardless of whether or not scattering is active. The quantity, $G/4\pi$, is often referred to as the mean incident intensity. Note that when the scattering coefficient is non-zero, the RTE is coupled over all intensity directions by the scalar flux relationship.

The scalar flux and radiative flux vector represent angular moments of the directional radiative intensity at a point,

$$G = \int_0^{2\pi} \int_0^{\pi} I\left(s\right) \sin\theta_{zn} d\theta_{zn} d\theta_{az},$$

$$q_i^r = \int_0^{2\pi} \int_0^{\pi} I\left(s\right) s_i \sin\theta_{zn} d\theta_{zn} d\theta_{az},$$

where θ_{zn} and θ_{az} are the zenith and azimuthal angles respectively as shown in Figure Fig. 3.1.

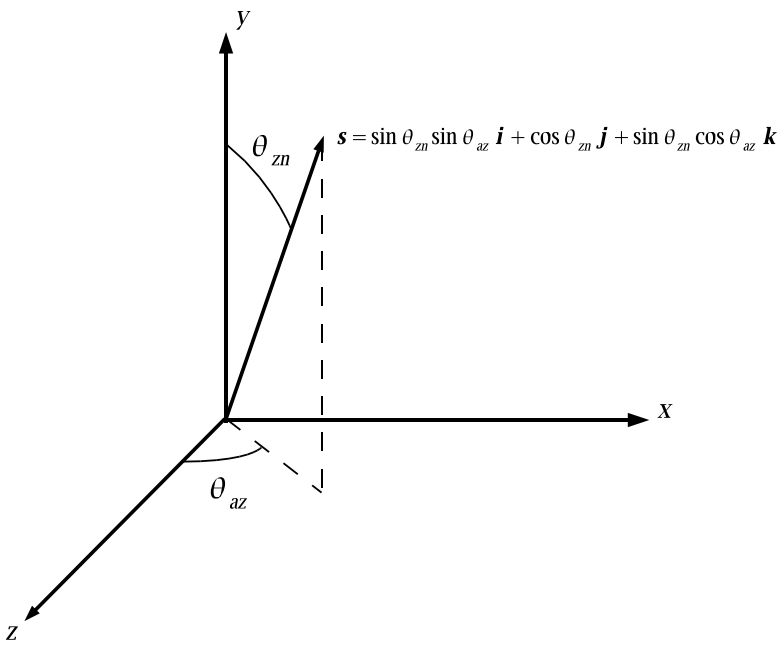

Fig. 3.1: Ordinate Direction Definition, $\mathbf{s} = \sin\theta_{zn}\sin\theta_{az}\mathbf{i} + \cos\theta_{zn}\mathbf{j} + \sin\theta_{zn}\cos\theta_{az}\mathbf{k}$.

The radiation intensity must be defined at all portions of the boundary along which $s_i n_i < 0$, where n_i is the outward directed unit normal vector at the surface. The intensity is applied as a weak flux boundary condition which is determined from the surface properties and temperature. The diffuse surface assumption provides reasonable accuracy for many engineering combustion applications. The intensity leaving a diffuse surface in all directions is given by

$$I\left(s\right) = \frac{1}{\pi}\left[\tau\sigma T_\infty^4 + \epsilon\sigma T_w^4 + \left(1-\epsilon-\tau\right)K\right], \tag{3.33}$$

where ϵ is the total normal emissivity of the surface, τ is the transmissivity of the surface, T_w is the temperature of the boundary, T_∞ is the environmental temperature and H is the incident radiation, or irradiation (incoming radiative flux). Recall that the relationship given by Kirchoff's Law that relates emissivity, transmissivity and reflectivity, ρ, is

$$\rho + \tau + \epsilon = 1.$$

where it is implied that $\alpha = \epsilon$.

Discretization Approach

Nalu supports two discretizations: control volume finite element and (CVFEM) edge-based vertex centered (EBVC). Each are finite volume forumations and each solve for the primitives are are each considered vertex-based schemes. Considerable testing has provided a set of general rules as to which scheme is optimal. In general, all equations and boundary conditions support either equation discretization with exception of the solid stress equation which has only been implemented for the CVFEM technique.

For generalized unstructured meshes that have poor quality, CVFEM has been shown to excell in accuracy and robustness. This is mostly due to the inhearant accuracy limitation for the non-orthogonal correction terms that appear in the diffusion term and pressure stabilization for the EBVC scheme. For generalized unstructured meshes of decent quality, either scheme is ideal. Finally, for highly structured meshes with substantail aspect ratios, the edge-based scheme is ideal.

In general, the edge-based scheme is at least two times faster per iteration than the element-based scheme. For some classes of flows, it can be up to four times faster. However, due to the lagged coupling between the projected nodal gradient equation and the dofs, on meshes with high non-orthogonality, nonlinear residual convergence can be delayed.

CVFEM Dual Mesh

The classic low Mach algorithm uses the finite volume technique known as the control volume finite element method, see Schneider, *[SR87]*, or Domino, *[Dom06]*. Control volumes (the mesh dual) are constructed about the nodes, shown in Figure Fig. 3.2 (upper left). Each element contains a set of sub-faces that define control-volume surfaces. The sub-faces consist of line segments (2D) or surfaces (3D). The 2D segments are connected between the element centroid and the edge centroids. The 3D surfaces (not shown here) are connected between the element centroid, the element face centroids, and the edge centroids. Integration points also exist within the sub-control volume centroids.

Recent work by Domino, *[Dom14]*, has provided a proof-of-concept higher order CVFEM implementation whereby the linear basis and dual mesh definition is extended to higher order. The current code base supports the usage of P=2 elements (quadratic) for both 2D and 3D quad/hex topologies. This method has been formally demonstrated to be third-order spatially accurate and second-order in-time accurate. General polynomial promotion has been deployed in the higher order github branch. Figure Fig. 3.2 illustrates a general polynomial promotion from P=1 to P=6 and demonstrated spectral convergence using the method of manufactured solutions in Figure Fig. 3.3.

When using CVFEM, the discretized equations described in this manual are evaluated at either subcontrol-surface integration points (terms that have been integrated by parts) or at the subcontrol volume (time and source terms). Interpolation within the element is obtained by the standard elemental basis functions,

$$\phi_{ip} = \sum N_k^{ip} \phi_k. \tag{3.34}$$

where the index k represents a loop over all nodes in the element.

Gradients at the subcontrol volume surfaces are obtained by taking the derivative of Eq. (3.34), to obtain,

$$\frac{\partial \phi_{ip}}{\partial x_j} = \sum \frac{\partial N_{j,k}^{ip}}{\partial x_j} \phi_k. \tag{3.35}$$

The usage of the CVFEM methods results in the canonical 27-point stencil for a structured hexahedral mesh.

Edge-Based Discretization

In the edge-based discretization, the dual mesh defined in the CVFEM method is used to pre-process both dual mesh nodal volumes (needed in source and time terms) and edge-based area vectors (required for integrated-by-parts quantities, e.g., advection and diffusion terms).

Consider Figure Fig. 3.4, which is the original set of CVFEM dual mesh quadrature points shown above in Figure Fig. 3.2. Specifically, there are four subcontrol volumes about node 5 that contribute to the nodal volume dual mesh. In an edge-based scheme, the time and source terms use single point quadrature by assembling these four subcontrol volume contributions (eight in 3D) into one single nodal volume. In most cases, source terms may include gradients that are obtained by using the larger element-based stencil.

The same reduction of gauss points is realized for the area vector. Consider the edge between nodes 5 and 6. In the full CVFEM approach, subcontrol surfaces within the top element (5,6,9,8) and bottom element (2,3,6,5) are reduced

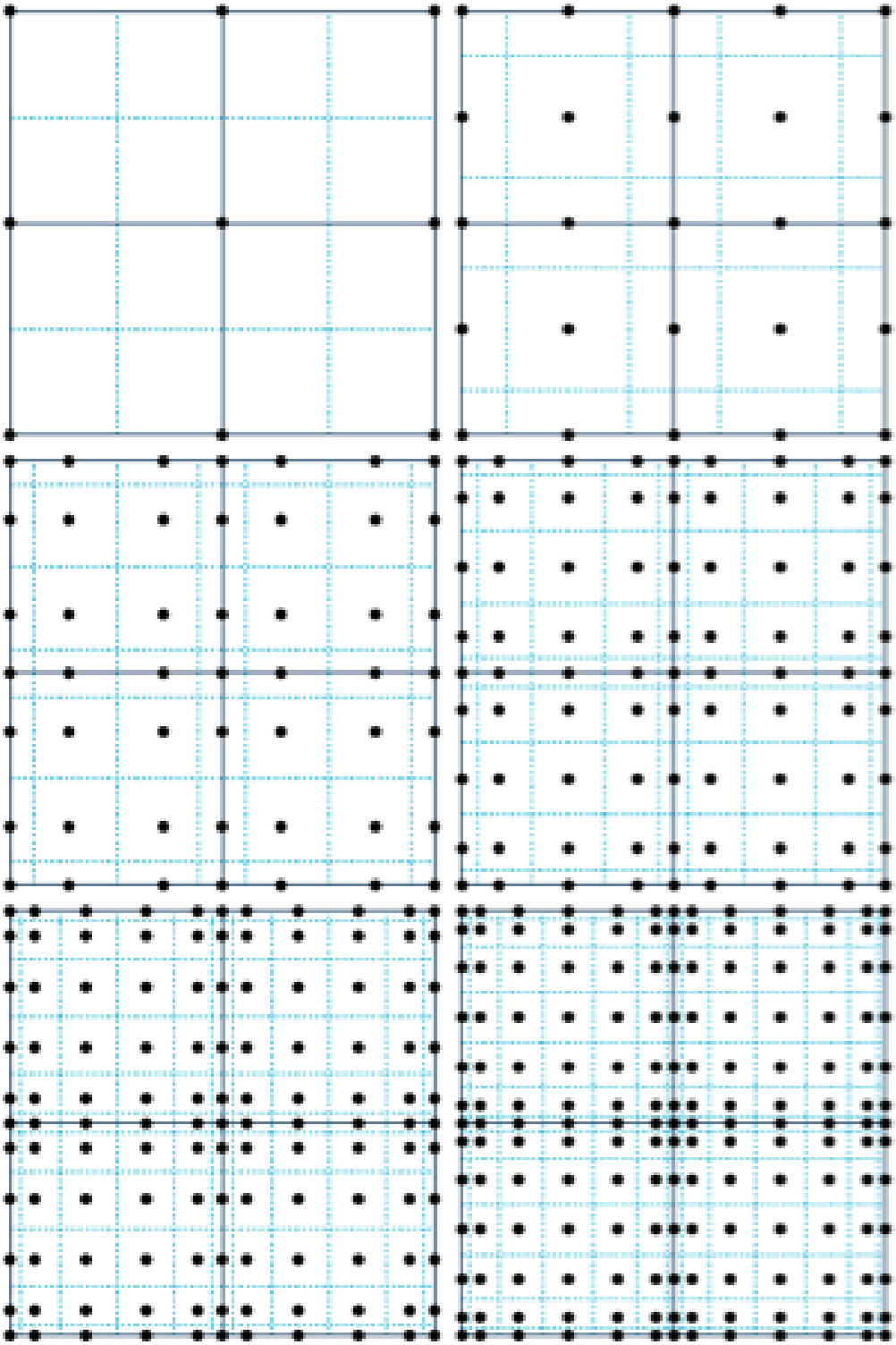

Fig. 3.2: Polynomial promotion for a canonical CVFEM quad element patch from $P = 1$ to $P = 6$.

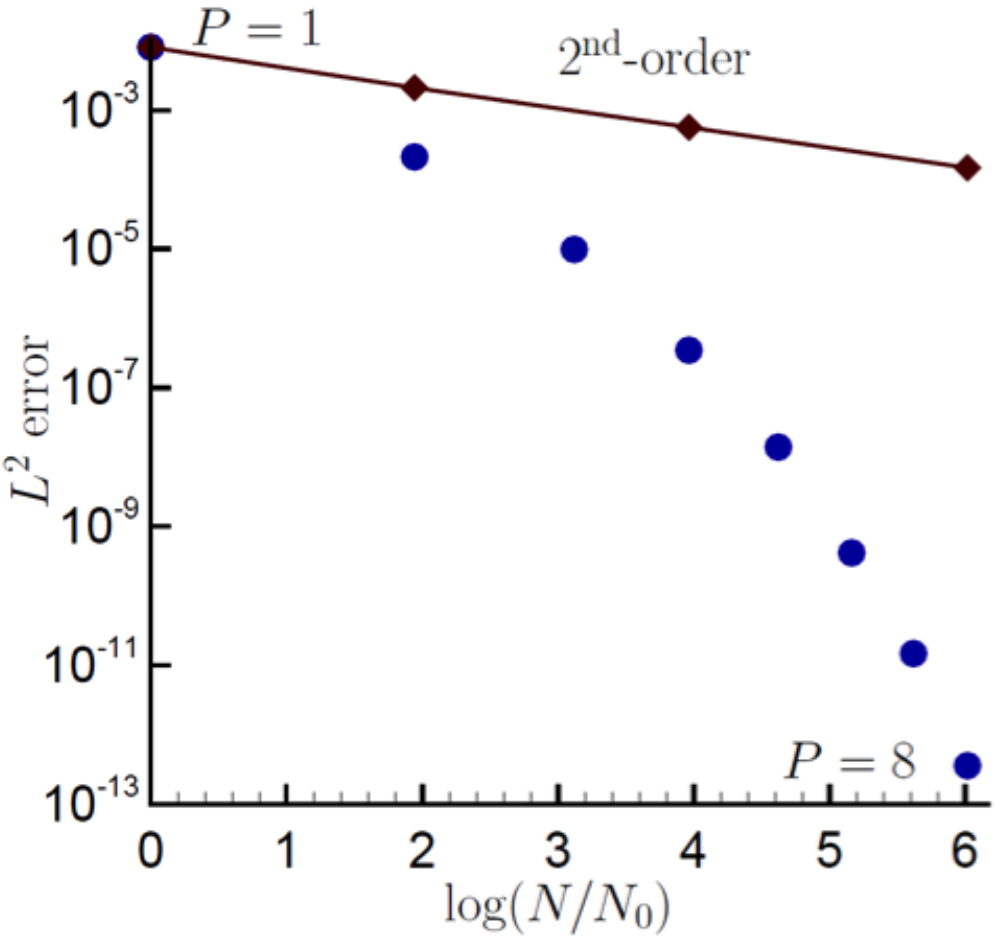

Fig. 3.3: A recent spectral convergence plot using the Method of Manufactured Solutions for $P = 1$ through $P = 8$.

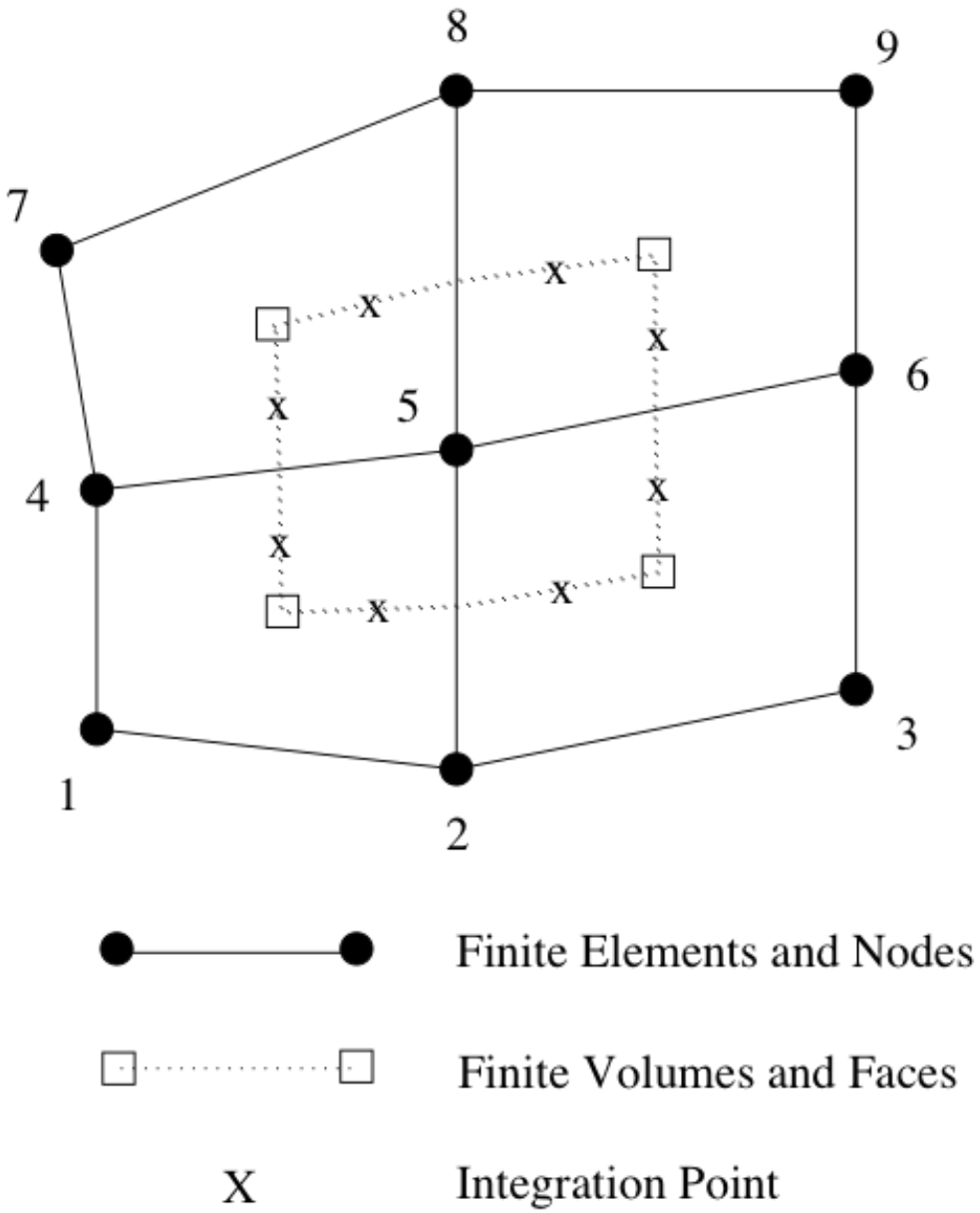

Fig. 3.4: A control volume centered about a finite-element node in a collection of 2-D quadrilateral elements (from *[Dom06]*.)

to a single area vector at the edge midpoint of nodes 5 and 6. Therefore, advection and diffusion is now done in a manner very consistent with a cell centered scheme, i.e., classic "left"/"right" states.

The consolidation of time and source terms to nodal locations along with advection and diffusion at the edge midpoint results in a canonical five-point stencil in 2D and seven in 3D. Note the ability to handle hybrid meshes is readily peformed one nodal volume and edge area are pre-processed. Edges and nodes are the sole topology that are iterated, thus making this scheme highly efficient, although inherantly limited to second order spatial order of accuracy.

In general, the edge-based scheme is second order spatially accurate. Formal verification has been done to evaluate the accuracy of the EBVC relative to other implemented methods (Domino, *[Dom14]*). The edge-based scheme, which is based on dual mesh post-processing, represents a commonly used finite volume method in gas dynamics applications. The method also lends itself to psuedo-higher order methodologies by the blending of extrapolated values using the projected nodal gradient and gauss point values (as does CVFEM). This provides a fourth order accurate diffusion and advection operator on a structured mesh.

The use of a consistent mass matrix is less apparent in edge-based schemes. However, if desired, the full element-based stencil can be used by iterating elements and assembling to the nodes.

The advantage of edge-based schemes over cell centered schemes is that the scheme naturally allows for a mixed elemental discretization. Projected nodal gradients can be element- or edge-based. LES filters and nodal gradients can also exploit the inherant elemental basis that exists in the pure CVFEM approach. In our experience, the optimal scheme on high quality meshes uses the CVFEM for the continuity solve and EBVC discretization for all other equations. This combination allows for the full CVFEM diffusion operator for the pressure Poisson equation and the EBVC approach for equations where inverse Reynolds scaling reduces the importance of the diffusion operator. This scheme can be activated by the use of the `use_edges: yes` Realm line command in combination of the LowMachEOM system line command, `element_continuity_eqs: yes`.

Projected Nodal Gradients

In the edge or element-based algorithm, projected nodal gradients are commonplace. Projected nodal gradients are used in the fourth order pressure stabilization terms, higher order upwind methods, discontinuity capturing operators (DCO) and turbulence source terms. For an edge-based scheme, they are also used in the diffusion term when non-orthogonality of the mesh is noted.

There are many procedures for determining the projected nodal gradient ranging from element-based schemes to edge-based approached. In general, the projected nodal gradient is viewed as an L_2 minimization between the discontinuous Gauss-point value and the continuous nodal value. The projected nodal gradient, in an L_2 sence is given by,

$$\int w G_j \phi dV = \int \frac{\partial \phi}{\partial x_j} dV. \tag{3.36}$$

Using integration-by-parts and a piece-wise constant test function, the above equation is written as,

$$\int w_I G_j \phi dV = \int \phi_{ip} n_j dS. \tag{3.37}$$

For a lumped L2 projected nodal gradient, the approach is based on a Green-Gauss integration,

$$G_j \phi = \frac{\int \phi_{ip} A_j}{dV}. \tag{3.38}$$

In the above lumped mass matrix approach, the value at the integration point can either be based on the CVFEM dual mesh evaluated at the subcontrol surface, i.e., the line command option, `element` or the `edge`, which evaluates the term at the edge midpoint using the assemble edge area vector. In all cases, the lumped mass matrix approach is strickly second order accurate. When running higher order CVFEM, a consistent mass matrix appraoch is required to maintain design order of the overall discretization. This is strickly due to the pressure stabilization whose accuracy can be affected by the form of the projected nodal gradient (see the Nalu theory manual or a variety of SNL-based publications).

In the description that follows, $\bar{G_j}\phi$ represent the average nodal gradient evaluated at the integration point of interest.

The choice of projected nodal gradients is specified in the input file for each dof. Keywords `element` or `edge` are used to define the form of the projection. The forms of the projected nodal gradients is arbitrary relative to the choosed underlying discretization. For strongly non-orthogonal meshes, it is recommended to use an element-based projected nodal gradient for the continuity equation when the EBVC method is in use. In some limited cases, e.g., pressure, mixture fraction and enthalpy, the `manage-png` line command can be used to solve the simple linear system for the consistent mass matrix.

Time and Source Terms

Time and source terms also volumetric contributions and also use the dual nodal volume. In both discretization approaches, this assembly is achieved as a simple nodal loop. In some cases, e.g., the k_{sgs} partial differential equation, the source term can use projected nodal gradients.

$$\int \frac{\partial \rho \phi}{\partial t} dV = \int S_\phi dV$$

Diffusion

As already noted, for the CVFEM method, the diffusion term at the subcontrol surface integration points use the the elemental shape functions and derivatives. For the standard diffusion term, and using Eq. (3.35), the CVFEM diffusion operator contribution at a given integration point (here simply demonstrated for a 2D edge with prescribed area vector) is as follows,

$$-\int \Gamma \frac{\partial \phi}{\partial x_j} A_j = -\Gamma_{ip} \left[\left(\frac{\partial N_0^{ip}}{\partial x} \phi_0 + \frac{\partial N_1^{ip}}{\partial x} \phi_1 \right) A_x + \left(\frac{\partial N_0^{ip}}{\partial y} \phi_0 + \frac{\partial N_1^{ip}}{\partial y} \phi_1 \right) A_y \right]$$

Standard Gauss point locations at the subcontrol surfaces can be shifted to the edge-midpoints for a more stable (monotonic) diffusion operator that is better conditioned for high aspect ratio meshes.

For the edge-based diffusion operator, special care is noted as there is no ability to use the elemental basis to define the diffusion operator. As with cell-centered schemes, non-orthogonal contributions for the diffusion operator arise due to a difference in direction between the assembled edge area vector and the distance vector between nodes on an edge. On skewed meshes, this non-orthogonality can not be ignored.

Following the work of Jasek, *[Jas96]*, the over-relaxed approach is used. The form of any gradient for direction j for field ϕ is

$$\frac{\partial \phi}{\partial x_j}_{ip} = \bar{G_j}\phi + \left[(\phi_R - \phi_L) - \bar{G_l}\phi dx_l \right] \frac{A_j}{A_k dx_k}. \tag{3.39}$$

In the above expression, we are iterating edges with a Left node L and Right node R along with edge-area vector, A_j. The $\bar{G_j}\phi$ is simple averaging of the left and right nodes to the edge midpoint. In general, a standard edge-based diffusion term is written as,

$$\begin{aligned} -\int \Gamma \frac{\partial \phi}{\partial x_j} A_j = -\Gamma_{ip} \Bigg[& \left(\bar{G_x}\phi A_x + \bar{G_y}\phi A_y \right) + (\phi_R - \phi_L) \frac{A_x A_x + A_y A_y}{A_x dx_x + A_y dx_y} \\ & - \left(\bar{G_x}\phi dx_x + \bar{G_y}\phi dx_y \right) \frac{A_x A_x + A_y A_y}{A_x dx_x + A_y dx_y} \Bigg]. \end{aligned}$$

Momentum Stress

The viscous stress tensor, τ_{ij} is formed based on the standard gradients defined above for either the edge or element-based discretization. The viscous force for component i is given by,

$$-\int \tau_{ij} A_j = -\int \mu_{ip} \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) A_j.$$

For example, the x and y-component of viscous force is given by,

$$F_x = -\mu_{ip} \left(\frac{\partial u_x}{\partial x} A_x + \frac{\partial u_x}{\partial y} A_y \right) - \mu_{ip} \left(\frac{\partial u_x}{\partial x} A_x + \frac{\partial u_y}{\partial x} A_y \right),$$
$$F_y = -\mu_{ip} \left(\frac{\partial u_y}{\partial x} A_x + \frac{\partial u_y}{\partial y} A_y \right) - \mu_{ip} \left(\frac{\partial u_x}{\partial y} A_x + \frac{\partial u_y}{\partial y} A_y \right).$$

Note that the first part of the viscous stress is simply the standard diffusion term. Note that the so-called non-solonoidal viscous stress contribution is frequently written in terms of projected nodal gradients. However, for CVFEM this procedure is rarely used given the elemental basis definition. As such, the use of shape function derivatives is clear.

The viscous stress contribution at an integration point for CVFEM (again using the 2D example with variable area vector) can be written as,

$$\begin{aligned} F_x = -\Gamma_{ip} \Bigg[& \left(\frac{\partial N_0^{ip}}{\partial x} u_{x0} + \frac{\partial N_1^{ip}}{\partial x} u_{x1} \right) A_x + \left(\frac{\partial N_0^{ip}}{\partial y} u_{x0} + \frac{\partial N_1^{ip}}{\partial y} u_{x1} \right) A_y \\ & + \left(\frac{\partial N_0^{ip}}{\partial x} u_{x0} + \frac{\partial N_1^{ip}}{\partial x} u_{x1} \right) A_x + \left(\frac{\partial N_0^{ip}}{\partial y} u_{y0} + \frac{\partial N_1^{ip}}{\partial x} u_{y1} \right) A_y \Bigg], \\ F_y = -\Gamma_{ip} \Bigg[& \left(\frac{\partial N_0^{ip}}{\partial x} u_{y0} + \frac{\partial N_1^{ip}}{\partial x} u_{y1} \right) A_x + \left(\frac{\partial N_0^{ip}}{\partial y} u_{y0} + \frac{\partial N_1^{ip}}{\partial y} u_{y1} \right) A_y \\ & + \left(\frac{\partial N_0^{ip}}{\partial y} u_{x0} + \frac{\partial N_1^{ip}}{\partial y} u_{x1} \right) A_x + \left(\frac{\partial N_0^{ip}}{\partial y} u_{y0} + \frac{\partial N_1^{ip}}{\partial x} u_{y1} \right) A_y \Bigg]. \end{aligned}$$

For the edge-based diffusion operator, the value of ϕ is substituted for the component of velocity, u_i in the Eq. (3.39).

$$\frac{\partial u_i}{\partial x_j}_{ip} = \bar{G_j} u_i + \left[(u_{iR} - u_{iL}) - \bar{G_l} u_i dx_l \right] \frac{A_j}{A_k dx_k}.$$

Common approaches in the cell-centered community are to use the projected nodal gradients for the $\frac{\partial u_j}{\partial x_i}$ stress component. However, in Nalu, the above form of equation is used.

Substituting the relations of the velocity gradients for the x and y-componnet of force above provides the following expression used for the viscous stress contribution:

$$\begin{aligned} F_x = -\mu_{ip} \Bigg[& \left(\bar{G_x} u_x A_x + \bar{G_y} u_x A_y \right) + (u_{xR} - u_{xL}) \frac{A_x A_x + A_y A_y}{A_x dx + A_y dy} \\ & - \left(\bar{G_x} u_x dx + \bar{G_y} u_x dy \right) \frac{A_x A_x + A_y A_y}{A_x dx + A_y dy} \Bigg] \\ & - \mu_{ip} \Bigg[\bar{G_x} u_x A_x + \bar{G_x} u_y A_y + (u_{xR} - u_{xL}) \frac{A_x A_x}{A_x dx + A_y dy} \\ & + \left(u_{yR} - u_{yL} \right) \frac{A_x A_y}{A_x dx + A_y dy} \\ & - \left(\bar{G_x} u_x dx + \bar{G_y} u_x dy \right) \frac{A_x A_x}{A_x dx + A_y dy} \\ & - \left(\bar{G_x} u_y dx + \bar{G_y} u_y dy \right) \frac{A_x A_y}{A_x dx + A_y dy} \Bigg], \end{aligned}$$

$$
\begin{aligned}
F_y = -\mu_{ip} \Bigg[& \left(\bar{G_x u_y} A_x + \bar{G_y u_y} A_y\right) + \left(u_{y\,R} - u_{y\,L}\right) \frac{A_x A_x + A_y A_y}{A_x dx + A_y dy} \\
& - \left(\bar{G_x u_y} dx + \bar{G_y u_y} dy\right) \frac{A_x A_x + A_y A_y}{A_x dx + A_y dy} \Bigg] \\
& - \mu_{ip} \Bigg[\bar{G_y u_x} A_x + \bar{G_y u_y} A_y + \left(u_{y\,R} - u_{y\,L}\right) \frac{A_y A_y}{A_x dx + A_y dy} \\
& + \left(u_{x\,R} - u_{x\,L}\right) \frac{A_y A_x}{A_x dx + A_y dy} \\
& - \left(\bar{G_x u_y}] dx + \bar{G_y u_y} dy\right) \frac{A_y A_y}{A_x dx + A_y dy} \\
& - \left(\bar{G_x u_x} dx + \bar{G_y u_x} dy\right) \frac{A_y A_x}{A_x dx + A_y dy} \Bigg],
\end{aligned}
$$

where above, the first [] and second [] represent the $\frac{\partial u_i}{\partial x_j} A_j$ and $\frac{\partial u_j}{\partial x_i} A_j$ contributions, respectively.

One can use this expression to recognize the ideal LHS sensitivities for row and columns for component u_i.

Advection Stabilization

In general, advection for both the edge and element-based scheme is identical with standard exception of the location of the integration points. The full advection term is simply written as,

$$ADV_\phi = \int \rho u_j \phi_{ip} A_j = \sum \dot{m} \phi_{ip}, \tag{3.40}$$

where ϕ is u_i, Z, h, etc.

The evaluation of ϕ_{ip} defines the advection stabilization choice. In general, the advection choice is a cell Peclet blending between higher order upwind (ϕ_{upw}) and a generalized un-stabilized central (Galerkin) operator, ϕ_{gcds},

$$\phi_{ip} = \eta \phi_{upw} + (1 - \eta) \phi_{gcds}. \tag{3.41}$$

In the above equation, η is a cell Peclet blending. The generalized central operator can take on a pure second order or pseudo fourth order form (see below). For the classic Peclet number functional form (see Equation (3.42)) a hybrid upwind factor, γ, can be used to ensure that no stabilization is added ($\eta = 0$) or that full upwind stabilization is included (as will be shown, even with limiter functions). The hybrid upwind factor allows one to modify the functional blending function; values of unity result in the normal blending function response in Figure Fig. 3.5; values of zero yield a pure central operator, i.e., blending function of zero; values >> unity result in a blending function value of unity, i.e., pure upwind. The constant A is implemented with a value of 5. The value of this constant can not be changed via the input file. Note that this functional form is named the "classic" form within the input file.

The classic cell Peclet blending function is given by

$$\eta = \frac{\gamma \mathrm{Pe}^2}{5 + \gamma \mathrm{Pe}^2}. \tag{3.42}$$

The classic Peclet functional form makes it difficult to dial in the exact point at which the Peclet factor transitions from pure upwind to pure central. Therefore, an alternative form is provided that has a hyperbolic tangeant functional form. This form allows one to specify the transition point and the width of the transition (see Equation (3.43)). The general tanh form is as follows,

$$\eta = \frac{1}{2}[(a + b) + (b - a) tanh(\frac{\mathrm{Pe} - \mathrm{c_{trans}}}{c_{width}})] \tag{3.43}$$

Above, the constant c_{trans} represents the transition Peclet number while c_{width} represents the width of the transition. The value of λ is simply the shift between of the raw tanh function from zero while δ is the difference between the

max Peclet factor (unity) and the minimum value prior to normalization. This approach ensures that the function starts at zero and asymptotes to unity,

$$\eta = \frac{1}{2}[1 + tanh(\frac{\mathrm{Pe} - \mathrm{c_{trans}}}{c_{width}})].$$

The cell-Peclet number is computed for each sub-face in the element from the two adjacent left (L) and right (R) nodes,

$$\mathrm{Pe} = \frac{\frac{1}{2}\left(u_{R,i} + u_{L,i}\right)\left(x_{R,i} - x_{L,i}\right)}{\nu}.$$

A dot-product is implied by repeated indices.

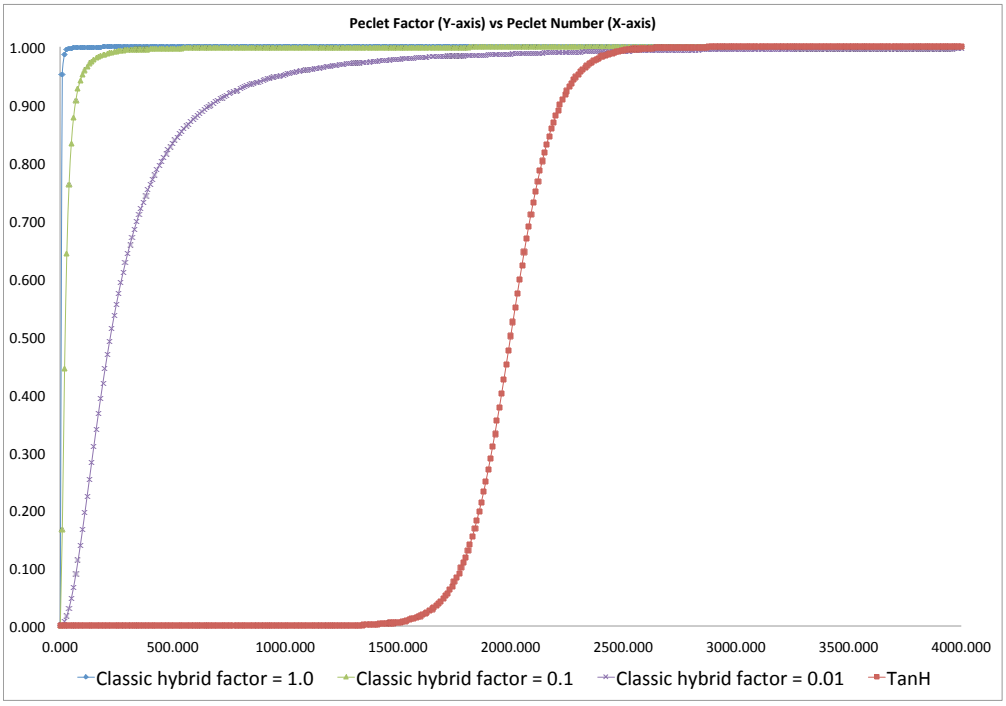

Fig. 3.5: Cell-Peclet number blending function outlining classic (varying the hybrid factor γ from 1.0, 0.1 and 0.01; again $A = 5$) and tanh functional form ($c_{trans} = 2000$ and $c_{width} = 200$).

The upwind operator, ϕ_{upw} is computed based on a blending of the extrapolated state (using the projected nodal gradient) and the linear interpolated state. Second or third order upwind is provided based on the value of α_{upw} blending

$$\begin{aligned} \phi_{upw} = \alpha_{upw}\tilde{\phi}^L_{upw} + \left(1 - \alpha_{upw}\right)\phi_{cds}; \dot{m} > 0, \\ \alpha_{upw}\tilde{\phi}^R_{upw} + \left(1 - \alpha_{upw}\right)\phi_{cds}; \dot{m} < 0. \end{aligned} \tag{3.44}$$

The extrapolated value based on the upwinded left (ϕ^L) or right (ϕ^R) state,

$$\begin{aligned} \tilde{\phi}^L_{upw} = \phi^L + d^L_j \frac{\partial \phi^L}{\partial x_j}, \\ \tilde{\phi}^R_{upw} = \phi^R - d^R_j \frac{\partial \phi^R}{\partial x_j}. \end{aligned} \tag{3.45}$$

The distance vectors are defined based on the distances between the L/R points and the integration point (for both edge or element-based),

$$\begin{aligned} d_j^L &= x_j^{ip} - x_j^L, \\ d_j^R &= x_j^R - x_j^{ip}. \end{aligned} \tag{3.46}$$

In the case of all transported quantities, a Van Leer limiter of the extrapolated value is supported and can be activated within the input file (using the solution options "limiter" specification).

Second order central is simply written as a linear combination of the nodal values,

$$\phi_{cds} = \sum N_k^{ip} \phi_k. \tag{3.47}$$

where N_k^{ip} is either evaluated at the subcontrol surface or edge midpoint. In the case of the edge-based scheme, the edge midpoint evaluation provides for a skew symmetric form of the operator.

The generalized central difference operator is provided by blending with the extrapolated values and second order Galerkin,

$$\phi_{gcds} = \frac{1}{2}\left(\hat{\phi}_{upw}^L + \hat{\phi}_{upw}^R\right), \tag{3.48}$$

where,

$$\begin{aligned} \hat{\phi}_{upw}^L &= \alpha\tilde{\phi}_{upw}^L + (1-\alpha)\,\phi_{cds}, \\ \hat{\phi}_{upw}^R &= \alpha\tilde{\phi}_{upw}^R + (1-\alpha)\,\phi_{cds}. \end{aligned} \tag{3.49}$$

The value of α provides the type of psuedo fourth order stencil and is specified in the user input file.

The above set of advection operators can be used to define an idealized one dimensional stencil denoted by ($i-2$, $i-1$, i, $i+1$, $i+2$), where i represents the particular row for the given transported variable. Below, in the table, the stencil can be noted for each value of α and α_{upw}.

$i-2$	$i-1$	i	$i+1$	$i+2$	α	α_{upw}
0	$-\frac{1}{2}$	0	$+\frac{1}{2}$	0	0	n/a
$+\frac{1}{8}$	$-\frac{6}{8}$	0	$+\frac{6}{8}$	$-\frac{1}{8}$	$\frac{1}{2}$	n/a
$+\frac{1}{12}$	$-\frac{8}{12}$	0	$+\frac{8}{12}$	$-\frac{1}{12}$	$\frac{2}{3}$	n/a
$+\frac{1}{4}$	$-\frac{5}{4}$	$+\frac{3}{4}$	$+\frac{1}{4}$	0	$\dot{m} > 0$	1
0	$-\frac{1}{4}$	$-\frac{3}{4}$	$+\frac{5}{4}$	$-\frac{1}{4}$	$\dot{m} < 0$	1
$+\frac{1}{6}$	$-\frac{6}{6}$	$+\frac{3}{6}$	$+\frac{2}{6}$	0	$\dot{m} > 0$	$\frac{1}{2}$
0	$-\frac{2}{6}$	$-\frac{3}{6}$	$+\frac{6}{6}$	$-\frac{1}{6}$	$\dot{m} < 0$	$\frac{1}{2}$

It is noted that by varying these numerical parameters, both high quality, low dissipation operators suitable for LES usage or limited, monotonic operators suitable for RANS modeling can be accomodated.

Pressure Stabilization

A number of papers describing the pressure stabilization approach that Nalu uses are in the open literature, Domino, *[Dom06]*, *[Dom08]*, *[Dom14]*. Nalu supports an incremental fourth order approximate projection scheme with time step scaling. By scaling, it is implied that a time scale based on either the physical time step or a combined elemental advection and diffusion time scale based on element length along with advection and diffusional parameters. An alternative to the approaximate projection concept is to view the method as a variational multiscale (VMS) method wherebye the momentum residual augments the continuity equation. This allows for a diagonal entry for the pressure degree of freedom.

Here, the fine-scale momentum residual is written in terms of a projected momentum residual evaluated at the Gauss point,

$$\mathbf{R}(u_i) = (\frac{\partial p}{\partial x_j} - G_j p). \tag{3.50}$$

The above equation is derived simply by writing a fine-scale momentum equation at the Gauss-points and using the nodal projected residual to reconstruct the individual terms. Therefore, the continuity equation solved, using the VMS-based projected momentum residual, is

$$\int \frac{\partial \bar{\rho}}{\partial t}\, dV + \int \left(\bar{\rho}\hat{u}_i + \tau G_i \bar{P}\right) n_i \, dS = \int \tau \frac{\partial \bar{P}}{\partial x_i} n_i \, dS.$$

Above, $G_i\bar{P}$ is defined as a L2 nodal projection of the pressure gradient. Note that the notion of a provisional velocity, $\hat{u}_i$, is used to signify that this velocity is the product of the momentum solve. The difference between the projected nodal gradient interpolated to the gauss point and the local gauss point pressure gradient provides a fourth order pressure stabilization term. This term can also be viewed as an algebraic form for the momentum residual. For the continuity equation only, a series of element-based options that shift the integration points to the edges of the iterated element is an option.

The Role of $\dot{m}$

In all of the above equations, the advection term is written in terms of a linearized mass flow rate including a sum over all subcontrol surface integration points, Eq (3.40). The mass flow rate includes the full set of stabilization terms obtained from the continuity solve,

$$\dot{m} = \left(\bar{\rho}\hat{u}_i + \tau G_i \bar{P} - \tau \frac{\partial \bar{P}}{\partial x_i}\right) n_i \, dS.$$

The inclusion of the pressure stabilization terms in the advective transport for the primitives is a required step for ensuring that the advection velocity is mass conserving. In practice, the mass flow rate is stored at each integration point in the mesh (edge midpoints for the edge-based scheme and subcontrol surfaces for the element-based scheme). When the mixed CVFEM/EBVC scheme is used, the continuity equation solves for a subcontrol-surface value of the mass flow rate. These values are assembled to the edge for use in the EBVC discretization approach. Therefore, the storage for mass flow rate is higher.

RTE Stabilization

The RTE is solved using the method of discrete ordinates using the symmetric Thurgood quadrature set. The discrete ordinates method is one in which discrete directions of the intensity are solved. The quadrature order, N, defines the number of ordinate directions that are solved in a given iteration. In the case of non-scattering media, this results is a set of decoupled linear partial differential equations. For the symmetric Thurgood set, the number of ordinate directions is given by $8N^2$. Values of N that are required for suitable accuracy starts at $N = 2$ with more than adequate resolution at $N = 4$.

For each ordinate direction, a weight is provided, w_k (not to be confused with the test function w). For each intensity ordinate direction, I_k, integrated quantities such as scalar flux and radiative heat flux are computed as,

$$G = \sum I_k w_k$$

and,

$$q_j = \sum I_k s_j w_k.$$

The stabilization that is used in the RTE equation can be placed in the class of residual-based stabilization. In this particular implementation, the scaled residual of the RTE equation is added. This implementation has its roots in the classic variational multiscale (VMS).

In the VMS framework, the degree of freedom is decomposed in terms of its resolved and fine scale, $I + I'$. Without specific definition of the test function, the weighted residual statement for the RTE within a VMS framework is given by,

$$\int w \left(s_i \frac{\partial}{\partial x_i} \left(I\left(s\right) + I'\left(s\right) \right) + \left(\mu_a + \mu_s \right) \left(I\left(s\right) + I'\left(s\right) \right) - \frac{\mu_a \sigma T^4}{\pi} - \frac{\mu_s}{4\pi} G \right) \mathrm{d}V = 0. \tag{3.51}$$

Grouping resolved and fine scale terms results in an equation takes the form of a standard Galerkin contribution in addition to the fine structure statement,

$$\begin{aligned} \int w \left(s_i \frac{\partial}{\partial x_i} I\left(s\right) + \left(\mu_a + \mu_s \right) I - \frac{\mu_a \sigma T^4}{\pi} - \frac{\mu_s}{4\pi} G \right) \mathrm{d}V \\ + \int w \left(s_i \frac{\partial}{\partial x_i} I'\left(s\right) + \left(\mu_a + \mu_s \right) I' \right) \mathrm{d}V = 0. \end{aligned} \tag{3.52}$$

Note that the isotropic source term has not contributed to the VMS framework other than through the right hand source term.

In general, gradients in the fine scale quantity are to be avoided. Therefore, the first term in the second line of Eq. (3.52) is integrated by parts to yield the following form (note the boundary term, $\int_\Gamma$ that is shown below is frequently dropped)

$$\begin{aligned} \int w \left(s_i \frac{\partial}{\partial x_i} I\left(s\right) + \left(\mu_a + \mu_s \right) I - \frac{\mu_a \sigma T^4}{\pi} - \frac{\mu_s}{4\pi} G \right) \mathrm{d}V \\ - \int I' s_i \frac{\partial w}{\partial x_i} \mathrm{d}V + \int_\Gamma w s_i I' n_i \mathrm{d}S + \int w (\mu_a + \mu_s) I' \mathrm{d}V = 0. \end{aligned} \tag{3.53}$$

The following ansatz, which now includes the classic stabilization parameter, τ, provides closure of the above fine scale equation,

$$I' = -\tau \left(s_i \frac{\partial}{\partial x_i} I\left(s\right) + \left(\mu_a + \mu_s \right) I\left(s\right) - \frac{\mu_a \sigma T^4}{\pi} - \frac{\mu_s}{4\pi} G \right) = -\tau R(s) \tag{3.54}$$

Substituting Eq. (3.54) into Eq. (3.53) yields,

$$\begin{aligned} \int w \left(s_i \frac{\partial}{\partial x_i} I\left(s\right) + \left(\mu_a + \mu_s \right) I - \frac{\mu_a \sigma T^4}{\pi} - \frac{\mu_s}{4\pi} G \right) \mathrm{d}V \\ + \int \tau s_i \frac{\partial w}{\partial x_i} R(s) \mathrm{d}V - \int_\Gamma \tau w R(s) s_i n_i \mathrm{d}S - \int \tau w (\mu_a + \mu_s) R(s) \mathrm{d}V = 0. \end{aligned} \tag{3.55}$$

In the above equation, the residual of the intensity equation for ordinate s is denoted by $R(s)$. A compact form of the equation is provided by defining a modified test function, $\tilde{w}$, (again note retention of the stabilized boundary term)

$$\begin{aligned} \int \tilde{w} \left(s_i \frac{\partial}{\partial x_i} I\left(s\right) + \left(\mu_a + \mu_s \right) I - \frac{\mu_a \sigma T^4}{\pi} - \frac{\mu_s}{4\pi} G \right) \mathrm{d}V \\ - \beta \int_\Gamma \tau w R(s) s_i n_i \mathrm{d}S = 0. \end{aligned} \tag{3.56}$$

where $\tilde{w}$ is simply equal to,

$$\tilde{w} = w + \tau \left(s_j \frac{\partial w}{\partial x_j} + \alpha (\mu_a + \mu_s) w \right). \tag{3.57}$$

When $\alpha = -1$, we have the above VMS derivation; for $\alpha = 1$, Galerkin Least Squares is realized; finally for $\alpha = 0$, we have SUPG. For any formulation other than VMS, the residual contribution at the boundaries of the domain is dropped ($\beta = 0$).

The full residual-based equation is placed in divergence form,

$$\int \tilde{w}\left(\frac{\partial}{\partial x_i}s_i I\left(s\right)+\left(\mu_a+\mu_s\right)I\left(s\right)-\frac{\mu_a\sigma T^4}{\pi}-\frac{\mu_s}{4\pi}G\right)\mathrm{d}V \\ -\beta\int_\Gamma \tau w R(s)s_i n_i \mathrm{d}S=0. \tag{3.58}$$

and split into its Galerkin and stabilized contributions,

$$\int w\left(\frac{\partial}{\partial x_i}s_i I\left(s\right)+\left(\mu_a+\mu_s\right)I\left(s\right)-\frac{\mu_a\sigma T^4}{\pi}-\frac{\mu_s}{4\pi}G\right)\mathrm{d}V \\ +\int \tau s_j\frac{\partial w}{\partial x_j}R(s)\mathrm{d}V \\ +\alpha\int \tau w(\mu_a+\mu_s)R(s)\mathrm{d}V \\ -\beta\int_\Gamma \tau w R(s)s_i n_i \mathrm{d}S=0. \tag{3.59}$$

Note that the first term in the above equation is integrated by parts,

$$\int w\frac{\partial}{\partial x_i}s_i I\left(s\right)\mathrm{d}V=-\int I\left(s\right)s_i\frac{\partial w}{\partial x_i}\mathrm{d}V+\int_\Gamma w s_i I\left(s\right)n_i\mathrm{d}S.$$

Again, the usage of Γ provides emphasis that the contribution is a boundary (exposed face) condition. Therefore, the full VMS-based stabilized RTE equation is as follows,

$$\int\left(-I\left(s\right)s_i\frac{\partial w}{\partial x_i}+\left(\mu_a+\mu_s\right)I\left(s\right)-\frac{\mu_a\sigma T^4}{\pi}-\frac{\mu_s}{4\pi}G\right)\mathrm{d}V \\ +\int_\Gamma w s_i I\left(s\right)n_i\mathrm{d}S \\ +\int\tau s_j\frac{\partial w}{\partial x_j}R(s)\mathrm{d}V \\ +\alpha\int\tau w(\mu_a+\mu_s)R(s)\mathrm{d}V \\ -\beta\int_\Gamma\tau w R(s)s_i n_i\mathrm{d}S=0. \tag{3.60}$$

Definition of the test function

Following the work of Martinez, *[Mar05]*, the test function is chosen to be a piecewise-constant value within the control volume, $w = w_I$ and zero outside of this control volume (Heaviside). A key property of this function, as pointed out by Martinez, is that its gradient is a distribution of delta functions on the control volume boundary:

$$\frac{\partial w_I}{\partial x_i}=-\mathbf{n}_I\delta(\mathbf{x}-\mathbf{x}_{\Gamma_I}) \tag{3.61}$$

where Γ_I is boundary of control volume I and $\mathbf{n}_I$ is the outward normal on that boundary. Substituting this relationship into the residual equation provides the final form of vertex-centered finite volume RTE stabilized equation,

$$\int I\left(s\right)s_i n_i\mathrm{d}S+\int\left(\left(\mu_a+\mu_s\right)I\left(s\right)-\frac{\mu_a\sigma T^4}{\pi}-\frac{\mu_s}{4\pi}G\right)\mathrm{d}V \\ +\int_\Gamma s_i I\left(s\right)n_i\mathrm{d}S \\ -\int\tau R(s)s_i n_i\mathrm{d}S+\alpha\int\tau(\mu_a+\mu_s)R(s)\mathrm{d}V-\beta\int_\Gamma\tau R(s)s_i n_i\mathrm{d}S=0. \tag{3.62}$$

Given this equation, either an edge-based or element-based scheme can be used. For $\alpha = 0$ and $\beta = 0$, it is noted that classic SUCV is obtained. The second line of Eq. (3.62) represents a boundary contribution. This is where the intensity boundary condition (Eq. (3.128)) is applied. As noted in the RTE equation section, when $s_j n_j$ is greater than zero, the interpolated intensity based on the surface nodal values is used. However, when $s_j n_j$ is less than zero, the intensity boundary condition value is used. Since the original RTE equation was integrated by parts, a natural surface flux contribution is applied. In alternative discretization approaches, e.g., the SUPG FEM-based Sierra Thermal Radiation Module: Syrinx code, the RTE is not integrated by parts. Therefore, no boundary term exists, and, therefore, a dirichlet bc is applied. At corner nodes, this approach can lead to non-intuitive approaches since the corner node might have surface facets that are both incoming and outgoing. Weak integration of the flux term eliminated this complexity.

The form of τ

The value of the stabilization parameter τ can take on a variety of forms. A classic derivation provides the form of τ to be broken out into two forms, $\tau_{adv} = \frac{h}{2}$ and $\tau_{diff} = \frac{1}{(\mu_a+\mu_s)}$. An ad-hoc blending is given by,

$$\tau = \left(\frac{1}{\frac{2}{h}^2 + (\mu_a + \mu_s)^2} \right)^{\frac{1}{2}} \tag{3.63}$$

Finally, the classic GFEM form of τ is given by use of the metric tensor for the element mapping is noted,

$$\tau = \beta^* [s_i g_{ij} s_j]^{-\frac{1}{2}}, \tag{3.64}$$

with β^* equal to unity for SUCV and $\frac{2}{15^{\frac{1}{2}}}$ for FEM.

Pure Edge-based Upwind Method

The residual-based stabilization apporach can lead to predicting negative intensities. This is simply due to the fact that the stabilization approach (SUPG) is a linear approach. Extensions of this residual-based stabilization to include a discontinuity capturing operator (DCO) are underway. This adds a non-linear stabilization approach that will, hopefully, eliminate negative intensity predictions.

Alternatively, a first order upwind approach has been implemented by using EBVC discretization. At this point, no higher order upwind extensions have been implemented. For the upwind implementation, the equation solved is,

$$\begin{aligned} \int I(s)\, s_i n_i \mathrm{d}S + \int \left((\mu_a + \mu_s) I(s) - \frac{\mu_a \sigma T^4}{\pi} - \frac{\mu_s}{4\pi} G \right) \mathrm{d}V \\ + \int_\Gamma s_i I(s)\, n_i \mathrm{d}S = 0. \end{aligned} \tag{3.65}$$

In the above equation, the "advection operator", $I(s)\, s_i n_i \mathrm{d}S$ is approximated as using the "upwind" intensity, e.g., if $s_j n_j$ is greater than zero, the left nodal value is used.

Finite Element SUPG Form

For the FEM, the test function is the standard weighting. Assuming a pure SUPG formulation, i.e., $\alpha = \beta = 0$ in Equation (3.60), thereby reducing the final form to the following:

$$\begin{aligned} \int \left(-I(s)\, s_i \frac{\partial w}{\partial x_i} + w[(\mu_a + \mu_s) I(s) - \frac{\mu_a \sigma T^4}{\pi} - \frac{\mu_s}{4\pi} G] \right) \mathrm{d}V \\ + \int_\Gamma w s_i I(s)\, n_i \mathrm{d}S \\ + \int \tau s_j \frac{\partial w}{\partial x_j} R(s) \mathrm{d}V \end{aligned} \tag{3.66}$$

The weak boundary condition is applied in a similar manner as with the CVFEM and EBVC form, however, using the appropriate FEM test test function definition. Finally, the form of τ follows the above CVFEM form.

Nonlinear Stabilization Operator (NSO)

An alternative to classic Peclet number blending is the usage of a discontinuity capturing operator (DCO), or in the low Mach context a nonlinear stabilization operator (NSO). In this method, an artifical viscosity is defined that is a function of the local residual and scaled computational gradients. Viable usages for the NSO can be advection/diffusion problems in addition to the aforementioned RTE VMS approach.

The formal finite element kernel for a NSO approach is as follows,

$$\sum_e \int_\Omega \nu(\mathbf{R}) \frac{\partial w}{\partial x_i} g^{ij} \frac{\partial \phi}{\partial x_j} d\Omega, \tag{3.67}$$

where $\nu(\mathbf{R})$ is the artifical viscosity which is a function of the pde fine-scale residual and g^{ij} is the covariant metric tensor).

For completeness, the covariant and contravarient metric tensor are given by,

$$g^{ij} = \frac{\partial x_i}{\partial \xi_k} \frac{\partial x_j}{\partial \xi_k}, \tag{3.68}$$

and

$$g_{ij} = \frac{\partial \xi_k}{\partial x_i} \frac{\partial \xi_k}{\partial x_j}, \tag{3.69}$$

where $\xi = (\xi_1, \xi_2, \xi_3)^T$. The form of $\nu(\mathbf{R})$ currently used is as follows,

$$\nu = \sqrt{\frac{\mathbf{R_k R_k}}{\frac{\partial \phi}{\partial x_i} g^{ij} \frac{\partial \phi}{\partial x_j}}}. \tag{3.70}$$

The classic paper by Shakib (*[SHZ91]*) represents the genesis of this method which was done in the accoustically compressible context.

A residual for a classic advection/diffusion/source pde is simply the fine scale residual computed at the gauss point,

$$\hat{\mathbf{R}} = \frac{\partial \rho \phi}{\partial t} + \frac{\partial}{\partial x_j}(\rho u_j \phi - \mu^{eff} \frac{\partial \phi}{\partial x_j}) - S \tag{3.71}$$

Note that the above equation requires a second derivative whose source is the diffusion term. The first derivative is generally determined by using projected nodal gradients. As will be noted in the pressure stabilization section, the advection term carries the pressure stabilization terms. However, in the above equation, these terms are not explicity noted. Therefore, an option is to subtract the fine scale continuity equation to obtain the final general form of the source term,

$$\mathbf{R} = \hat{\mathbf{R}} - \phi(\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_j}{\partial x_j}). \tag{3.72}$$

An alternative to the fine-scale PDE is a form that is found by differencing the linearized form of the residual from the nonlinear residual,

$$\mathbf{R} = \frac{\partial \rho u_j \phi}{\partial x_j} - (\phi \frac{\partial \rho u_j}{\partial x_j} + \rho u_j \frac{\partial \phi}{\partial x_j}). \tag{3.73}$$

The above resembles a commutation error in the nonlinear advection term.

In general, the NSO-ν is prone to percision issues when the gradients are very close to zero. As such, the value of ν is limited to a first-order like value. This parameter is proposed as follows: if an operator were written as a Galerkin (un-stabilized) plus a diffusion operator, what is the value of the diffusion coefficient such that first-order upwind is obtained for the combined operator? This upwind limited value of ν provides the highest value that this coefficient can (or should) be. The current form of the limited upwind ν is as follows,

$$\nu^{upw} = C_{upw}(\rho u_i g_{ij} \rho u_j)^{\frac{1}{2}} \tag{3.74}$$

where C_{upw} is generally taked to be 0.1.

Using a piecewise-constant test function suitable for CVFEM and EBVC schemes (the reader is refered to the VMS RTE section), Eq. (3.67) can be written as,

$$-\sum_e \int_\Gamma \nu(\mathbf{R}) g^{ij} \frac{\partial \phi}{\partial x_j} n_i dS. \tag{3.75}$$

A fourth order form, which writes the stabilization as the difference between the Gauss-point gradient and the projected nodal gradient interpolated to the Gauss-point, is also supported,

$$-\sum_e \int_\Gamma \nu(\mathbf{R}) g^{ij} (\frac{\partial \phi}{\partial x_j} - G_j \phi) n_i dS. \tag{3.76}$$

NSO Based on Kinetic Energy Residual

An alternative formulation explored is to share the general kernal form shown in Equation (3.76), however, compute ν based on a fine-scale kinetic energy residual. In this formulation, the fine-scale kinetic energy residual is obtained from the fine-scale momentum residual dotted with velocity. As with the continuity stabilization approach, the fine-scale momentum residual is provided by Equation (3.77). Therefore, the fine-scale kinetic energy is written as,

$$\mathbf{R}_{ke} = \frac{u_j(\frac{\partial p}{\partial x_j} - G_j p)}{2}, \tag{3.77}$$

while the denominator for ν now includes the gradient in ke,

$$\nu = \sqrt{\frac{\mathbf{R}_{ke}\mathbf{R}_{ke}}{\frac{\partial ke}{\partial x_i} g^{ij} \frac{\partial ke}{\partial x_j}}}. \tag{3.78}$$

The kinetic energy is simply given by,

$$ke = \frac{u_k u_k}{2} \tag{3.79}$$

The kinetic energy form of ν is used for all equation sets with transformation by usage of a turbulent Schmidt/Prandtl number.

Local or Projected NSO Diffusive Flux Coefficient

While the NSO kernel is certainly evaluated at the subcontrol surfaces, the evaluation of ν can be computed by a multitude of approaches. For example, the artificial diffusive flux coefficient can be computed locally (with local residuals and local metric tensors) or can be projected to the nodes (via an L_{oo} or L_2 projection) and re-interpolated to the gauss points. The former results in a sharper local value while the later results in a more filtered-like value. The code base only supports a local NSO ν calculation.

General Findings

In general, the NSO approach seems to work best when running the fourth-order option as the second-order implementation still looks more diffuse. When compared to the standard MUSCL-limited scheme, the NSO is the preferred choice. More work is underway to improve stabilization methods. Note that a limited set of NSOs are activated in the code base with specific interest on scalar transport, e.g, momentum, mixture fraction and static enthalpy transport. When using the 4^{th} order method, the consistent mass matrix approach for the projected nodal gradients is supported for higher order.

NSO as a Turbulence Model

The kinetic energy residual form has been suggested to be used as a turbulence model (Guermond and Larios, 2015). However, inspection of the above NSO kernel form suggests that the model form is not symmetric. Rather, in the context of turbulence modeling, is closer to the metric tensor acting on the difference between the rate of strain and antisymmetric tensor. As such, the theory developed, e.g., for eigenvalue perturbations of the stress tensor (see Jofre and Domino, 2017) can not be applied. In this section, a new form of the NSO is provided in an effort to be used for an LES closure.

In this proposed NSO formulation, the subgrid stress tensor, $\tau_{ij}^{sgs} = \overline{u_i u_j} - \bar{u}_i \bar{u}_j$, is given by,

$$\tau_{ij}^{sgs} = -2\rho\nu g^{ij}(S_{ij} - \frac{1}{3}\frac{\partial u_k}{\partial x_k}\delta_{ij}) = -2\rho\nu g^{ij} S_{ij}^{*}. \tag{3.80}$$

Interestingly, the units of ν are of an inverse time scale while the product $2\rho\nu g^{ij}$ can be viewed as an non-isotropic eddy viscosity, μ_{ij}^{t}.

The first order clipping may be relaxed by defining ν as,

$$\nu = \frac{|\mathbf{R}_{ke}|}{||ke||_{\infty}}. \tag{3.81}$$

The above form would be closer to what Guermond uses and would avoid the divide-by-zero noted in regions of uniform flow.

Turbulence Modeling

Unlike a RANS approach which models most or all of the turbulent fluctuations, LES directly solves for all resolved turbulent length scales and only models the smallest scales below the grid size. In this way, a majority of the problem-dependent, energy-containing turbulent structure is directly solved in a model-free fashion. The subgrid scales are closer to being isotropic than the resolved scales, and they generally act to dissipate turbulent kinetic energy cascaded down from the larger scales in momentum-driven turbulent flows. Modeling of these small scales is generally more straightforward than RANS approaches, and overall solutions are usually more tolerant to LES modeling errors because the subgrid scales comprise such a small portion of the overall turbulent structure. While LES is generally accepted to be much more accurate than RANS approaches for complex turbulent flows, it is also significantly more expensive than equivalent RANS simulations due to the finer grid resolution required. Additionally, since LES results in a full unsteady solution, the simulation must be run for a long time to gather any desired time-averaged statistics. The tradeoff between accuracy and cost must be weighed before choosing one method over the other.

The separation of turbulent length scales required for LES is obtained by using a spatial filter rather than the RANS temporal filter. This filter has the mathematical form

$$\overline{\phi(\boldsymbol{x}, t)} \equiv \int_{-\infty}^{+\infty} \phi(\boldsymbol{x}', t) G(\boldsymbol{x}' - \boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}', \tag{3.82}$$

which is a convolution integral over physical space $\boldsymbol{x}$ with the spatially-varying filter function G. The filter function has the normalization property $\int_{-\infty}^{+\infty} G(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 1$, and it has a characteristic length scale Δ so that it filters out turbulent length scales smaller than this size. In the present formulation, a simple "box filter" is used for the filter function,

$$G(\boldsymbol{x}' - \boldsymbol{x}) = \begin{cases} 1/V & : \quad (\boldsymbol{x}' - \boldsymbol{x}) \in \mathcal{V} \\ 0 & : \quad \text{otherwise} \end{cases},$$

where V is the volume of control volume $\mathcal{V}$ whose central node is located at $\boldsymbol{x}$. This is essentially an unweighted average over the control volume. The length scale of this filter is approximated by $\Delta = V^{\frac{1}{3}}$. This is typically called the grid filter, as it filters out scales smaller than the computational grid size.

Similar to the RANS temporal filter, a variable can be represented in terms of its filtered and subgrid fluctuating components as

$$\phi = \bar{\phi} + \phi'.$$

For most forms of the filter function $G(\boldsymbol{x})$, repeated applications of the grid filter to a variable do not yield the same result. In other words, $\bar{\bar{\phi}} \neq \bar{\phi}$ and therefore $\overline{\phi'} \neq 0$, unlike with the RANS temporal averages.

As with the RANS formulation, modeling is much simplified in the presence of large density variations if a Favre-filtered approach is used. A Favre-filtered variable $\tilde{\phi}$ is defined as

$$\tilde{\phi} \equiv \frac{\overline{\rho\phi}}{\bar{\rho}}$$

and a variable can be decomposed in terms of its Favre-filtered and subgrid fluctuating component as

$$\phi = \tilde{\phi} + \phi''.$$

Again, note that the useful identities for the Favre-filtered RANS variables do not apply, so that $\tilde{\bar{\phi}} \neq \tilde{\phi}$ and $\overline{\phi''} \neq 0$. The Favre-filtered approach is used for all LES models in Nalu.

Standard Smagorinsky LES Model

The standard Smagorinsky LES closure model approximates the subgrid turbulent eddy viscosity using a mixing length-type model, where the LES grid filter size Δ provides a natural length scale. The subgrid eddy viscosity is modeled simply as (Smagorinsky)

$$\mu_t = \rho \left(C_s \Delta\right)^2 |\tilde{S}|, \tag{3.83}$$

The constant coefficient C_s typically varies between 0.1 and 0.24 and should be carefully tuned to match the problem being solved (Rogallo and Moin, *[RM84]*). The default value of 0.17 is assigned in the code base.

Although this model is desirable due to its simplicity and efficiency, care should be taken in its application. It is known to predict subgrid turbulent eddy viscosity proportional to the shear rate in the flow, independent of the local turbulence intensity. Non-zero subgrid turbulent eddy viscosity is even predicted in completely laminar regions of the flow, sometimes even preventing a natural transition to turbulence. The model also does not asymptotically replicate near wall behavior without either dampening or a dynamic procedure.

Wall Adapting Local Eddy-Viscosity, WALE

The WALE model of Ducros el al., *[DNP98]*, properly captures the asymptotic behavior for flows that are wall bounded. In this model, the turbulent viscosity is given by,

$$\mu_t = \rho \left(C_w \Delta\right)^2 \frac{\left(S_{ij}^d S_{ij}^d\right)^{3/2}}{\left(S_{ij} S_{ij}\right)^{5/2} + \left(S_{ij}^d S_{ij}^d\right)^{5/4}}, \tag{3.84}$$

with the constant C_w of 0.325 and a standard filter, Δ related to the volume, $V^{\frac{1}{3}}$. The rate of strain tensor is defined as,

$$S_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{3.85}$$

while S_{ij}^d is,

$$S_{ij}^d = \frac{1}{2}\left(g_{ij}^2 + g_{ji}^2\right). \tag{3.86}$$

Finally, the velocity gradient squared ters are

$$g_{ij}^2 = \frac{\partial u_i}{\partial x_k}\frac{\partial u_k}{\partial x_j} \tag{3.87}$$

and

$$g_{ji}^2 = \frac{\partial u_j}{\partial x_k}\frac{\partial u_k}{\partial x_i}. \tag{3.88}$$

One Equation k^{sgs}

See k^{sgs} pde section.

SST RANS Model

As noted, Nalu does support a SST RANS-based model (the reader is referred to the SST equation set description).

Wall Models

Flows are either expected to be fully resolved or, alternatively, under-resolved where wall functions are used. A classic law of the wall has been implemented in Nalu. Wall models to handle adverse pressure gradients are planned. For more information of the form of wall models, please refer to the boundary condition section of this manual.

Supported Boundary Conditions

Inflow Boundary Condition

Continuity

Continuity uses a flux boundary condition with the incoming mass flow rate based on the user specified values for velocity,

$$\dot{m}_c = \rho^{spec} u_j^{spec} A_j.$$

As this is a vertex-based code, at inflow and Dirichlet wall boundary locations, the continuity equation uses the specified velocity within the inflow boundary condition block.

Momentum, Mixture Fraction, Enthalpy, Species, k_{sgs}, k and ω

These degree-of-freedoms (DOFs) each use a Dirichlet value with the specified user value. For all Dirichlet values, the row is zeroed with a unity placed on the diagonal. The residual is zeroed and set to the difference between the current value and user specified value.

Wall Boundary Conditions

Continuity

Continuity uses a no-op.

Momentum

When resolving the boundary layer, Momentum again uses a no-slip Dirichlet condition., e.g., $u_i = 0$.

In the case of a wall model, a classic wall function is applied. The wall shear stress enters the discretization of the momentum equations by the term

$$\int \tau_{ij} n_j dS = -F_{wi}. \tag{3.89}$$

Wall functions are used to prescribe the value of the wall shear stress rather than resolving the boundary layer within the near-wall domain. The fundamental momentum law of the wall formulation, assuming fully-developed turbulent flow near a no-slip wall, can be written as,

$$u^+ = \frac{u_{\|}}{u_\tau} = \frac{1}{\kappa} \ln\left(E y^+\right), \tag{3.90}$$

where u^+ is defined by the the near-wall parallel velocity, $u_{\|}$, normalized by the wall friction velocity, u_τ. The wall friction velocity is related to the turbulent kinetic energy by,

$$u_\tau = C_\mu^{1/4} k^{1/2}. \tag{3.91}$$

by assuming that the production and dissipation of turbulence is in local equilibrium. The wall friction velocity is also computed given the density and wall shear stress,

$$u_\tau = (\frac{\tau_w}{\rho})^{0.5}.$$

The normalized perpendicular distance from the point in question to the wall, y^+, is defined as the following:

$$y^+ = \frac{\rho Y_p}{\mu} \left(\frac{\tau_w}{\rho}\right)^{1/2} = \frac{\rho Y_p u_\tau}{\mu}. \tag{3.92}$$

The classical law of the wall is as follows:

$$u^+ = \frac{1}{\kappa} \ln(y^+) + C, \tag{3.93}$$

where κ is the von Karman constant and C is the dimensionless integration constant that varies based on authorship and surface roughness. The above expression can be re-written as,

$$u^+ = \frac{1}{\kappa} \ln(y^+) + \frac{1}{\kappa} \ln(\exp(\kappa C)), \tag{3.94}$$

or simplified to the following expression:

$$\begin{aligned} u^+ &= \frac{1}{\kappa}\left(\ln(y^+) + \ln(\exp(\kappa C))\right) \\ &= \frac{1}{\kappa}\ln(Ey^+). \end{aligned} \tag{3.95}$$

In the above equation, E, is referred to in the text as the dimensionless wall roughness parameter and is described by,

$$E = \exp(\kappa C). \tag{3.96}$$

In Nalu, κ is set to the value of 0.42 while the value of E is set to 9.8 for smooth walls (White suggests values of $\kappa = 0.41$ and $E = 7.768$.). The viscous sublayer is assumed to extend to a value of y^+ = 11.63.

The wall shear stress, τ_w, can be expressed as,

$$\tau_w = \rho u_\tau^2 = \rho u_\tau \frac{u_\parallel}{u^+} = \frac{\rho\kappa u_\tau}{\ln\left(Ey^+\right)} u_\parallel = \lambda_w u_\parallel, \tag{3.97}$$

where λ_w is simply the grouping of the factors from the law of the wall. For values of y^+ less than 11.63, the wall shear stress is given by,

$$\tau_w = \mu \frac{u_\parallel}{Y_p}. \tag{3.98}$$

The force imparted by the wall, for the i_{th} component of velocity, can be written as,

$$F_{w,i} = -\lambda_w A_w u_{i\parallel}, \tag{3.99}$$

where A_w is the total area over which the shear stress acts.

The use of a general, non-orthogonal mesh adds a slight complexity to specifying the force imparted on the fluid by the wall. As shown in Equation (3.99), the velocity component parallel to the wall must be determined. Use of the unit normal vector, n_j, provides an easy way to determine the parallel velocity component by the following standard vector projection:

$$\Pi_{ij} = \left[\delta_{ij} - n_i n_j\right]. \tag{3.100}$$

Carrying out the projection of a general velocity, which is not necessarily parallel to the wall, yields the velocity vector parallel to the wall,

$$u_{i\parallel} = \Pi_{ij} u_j = u_i\left(1 - {n_i}^2\right) - \sum_{j=1;j\neq j}^{n} u_j n_i n_j. \tag{3.101}$$

Note that the component that acts on the particular i^{th} component of velocity,

$$-\lambda_w A_w \left(1 - n_i n_i\right) u_{i\parallel}, \tag{3.102}$$

provides a form that can be potentially treated implicitly; i.e., in a way to augment the diagonal dominance of the central coefficient of the i^{th} component of velocity. The use of residual form adds a slight complexity to this implicit formulation only in that appropriate right-hand-side source terms must be added.

Mixture Fraction

If a value is specified for each quantity within the wall boundary condition block, a Dirichlet condition is applied. If no values are specified, a zero flux condition is applied.

Enthalpy

If the temperature is specified within the wall boundary condition block, a Dirichlet condition is always specified. Wall functions for enthalpy transport have not yet been implemented.

The simulation tool supports multi-physics coupling via conjugate heat transfer and radiative heat transfer. Coupling parameters required for the thermal boundary condition are post processed by the fluids or PMR Realm. For conjugate and radiative coupling, the thermal solve provides the surface temperature. From the surface temperature, a wall enthalpy is computed and used.

Thermal Heat Conduction

If a temperature is specified in the wall block, and the surface is not an interface condition, then a Dirichlet approach is used. If conjugate heat transfer is included, then the boundary condition applied is as follows,

$$-\kappa \frac{\partial T}{\partial x_j} n_j dS = h(T - T^o) dS,$$

where h is the heat transfer coefficient and T^o is the reference temperature. The details of how these quantities are computed are currently omitted in this manual. In general, the quantities are post processed from the fluids temperature field. A surface-based gradient is computed on the boundary face. Nodes on the face augment a heat transfer coefficient field while nodes off the face contribute to a reference temperature.

For radiative heat transfer, the boundary condition applied is as follows:

$$-\kappa \frac{\partial T}{\partial x_j} n_j dS = \epsilon(\sigma T^4 - H) dS,$$

where H is again the irradiation provided by the RTE solve.

If no temperature is specified or an adiabatic line command is used, a zero flux condition is applied.

Species

If a value is specified for each quantity within the wall boundary condition block, a Dirichlet condition is applied. If no values are specified, a zero flux condition is applied.

Atmospheric Boundary Layer Surface Conditions

Monin-Obukhov Theory

Consider atmospheric flow over a flat but non-smooth surface; the coordinate system convention is that flow is along the x-axis, while the z-axis is oriented normal to the surface. The surface layer is the relatively thin layer near the surface where strong wind and temperature gradients exist. Turbulence within this layer can be generated through mechanisms of both shear and thermal convection; the relative contributions of these two mechanisms is determined by the stability state of the atmosphere. The stability state is characterized by the Monin-Obukhov length:

$$L = -\frac{u_\tau^3 \theta_{ref}}{\kappa g (\overline{w'\theta'})_s};$$

u_τ is the friction velocity, defined as the square root of the magnitude of the Reynolds shear stress at the surface, or

$$u_\tau = \left(\overline{w'u'}^2 + \overline{w'u'}^2 \right)^{1/4} = \sqrt{\frac{\tau_s}{\rho_s}}$$

θ_{ref} is a reference (virtual potential) temperature associated with the air within the surface layer; for example, the average temperature within the surface layer. $\kappa \approx 0.41$ is the von Karman constant, and g is the acceleration of gravity. $\overline{w'\theta'}_s$ is the surface turbulent temperature flux. Both the turbulent shear stress and turbulent temperature flux are approximately constant within the surface layer.

Applying a gradient diffusion model for the turbulent temperature flux leads to:

$$\overline{w'\theta'}_s = -k_T \frac{\partial \theta}{\partial z}$$

The sign of L is then connected to the sign of the temperature gradient within the surface layer. Three regimes are delineated:

- $\frac{1}{L} > 0, \quad \frac{\partial \theta}{\partial z} > 0$, stable stratification
- $\frac{1}{L} = 0, \quad \frac{\partial \theta}{\partial z} = 0$, neutral stratification
- $\frac{1}{L} < 0, \quad \frac{\partial \theta}{\partial z} < 0$, unstable stratification

Monin-Obukhov theory postulates the following similarity laws for mean velocity parallel to the surface and temperature,

$$\frac{\kappa z}{u_\tau} \frac{\partial \overline{u}_{||}}{\partial z} = \phi_m \left(\frac{z}{L} \right), \tag{3.103}$$

$$\frac{\kappa z u_\tau}{\overline{w'\theta'}_s} \frac{\partial \overline{\theta}}{\partial z} = \phi_h \left(\frac{z}{L} \right), \tag{3.104}$$

where the forms of the non-dimensional functions ϕ_m and ϕ_h are determined from empirical observations. Analytical functions have been fit to the data; these are not given here, rather, we present the integrated form of ((3.103)) and ((3.104)), since these are the forms required by the code implementation.

For neutral stratification, $\phi_m = 1$ and we recover the logarithmic profile for a "fully rough" surface,

$$\overline{u}_{||}(z) = \frac{u_\tau}{\kappa} \ln \frac{z}{z_0}, \tag{3.105}$$

where z_0 is the characteristic roughness height. Note that viscous scaling involving surface viscosity and density properties is not required with this form of the logarithmic profile, since the roughness height is large enough to eliminate the presence of a laminar sublayer and buffer layer.

For stable stratification, the surface layer profiles take the form

$$\overline{u}_{||}(z) = \frac{u_\tau}{\kappa} \left(\ln \frac{z}{z_0} + \gamma_m \frac{z}{L} \right) \tag{3.106}$$

$$\overline{\theta}(z) = \overline{\theta}(z_0) + \frac{\theta_*}{\kappa} \left(\alpha_h \ln \frac{z}{z_0} + \gamma_h \frac{z}{L} \right) \tag{3.107}$$

θ_* is calculated from the temperature flux and friction velocity as $\theta_* = -\frac{\overline{w'\theta'}_s}{u_\tau}$, and γ_m, α_h, and γ_h are constants specified below.

For unstable stratification, the surface layer profiles take the form

$$\overline{u}_{||}(z) = \frac{u_\tau}{\kappa} \left(\ln \frac{z}{z_0} - \psi_m \left(\frac{z}{L} \right) \right) \tag{3.108}$$

$$\overline{\theta}(z) = \overline{\theta}(z_0) + \alpha_h \frac{\theta_*}{\kappa} \left(\ln \frac{z}{z_0} - \psi_h \left(\frac{z}{L} \right) \right) \tag{3.109}$$

where

$$\psi_m\left(\frac{z}{L}\right) = 2\ln\frac{1+x}{2} + \ln\frac{1+x^2}{2} - 2\tan^{-1}x + \frac{\pi}{2}, \quad x = \left(1-\beta_m\frac{z}{L}\right)^{1/4}, \tag{3.110}$$

$$\psi_h\left(\frac{z}{L}\right) = \ln\frac{1+y}{2}, \quad y = \left(1-\beta_h\frac{z}{L}\right)^{1/2}. \tag{3.111}$$

The constants used in ((3.106)) – ((3.111)) are *[Dye74]*

$$\kappa = 0.41, \;\; \alpha_h = 1, \;\; \beta_m = 16, \;\; \beta_h = 16, \;\; \gamma_m = 5.0, \;\; \gamma_h = 5.0.$$

ABL Wall Function

The equations from the preceeding section can be used to formulate a wall function boundary condition for simulation of atmospheric boundary layers. The user-specified inputs to this boundary condition are: roughness length, z_0, and surface heat flux, $q_s = \rho C_p \overline{w'\theta'})_s$. The surface layer profile model is evaluated for each surface boundary flux integration point; the wall-normal distance of the "first point off the wall" is taken to be one fourth of the length of the nearest edge intersecting the boundary face. The boundary condition is specified weakly through the imposition of a surface shear stress and surface heat flux.

The procedure for applying the boundary condition is as follows:

1. Determine the stratification state of the boundary layer by calculating the sign of the Monin-Obukhov length scale.
2. Solve the appropriate profile equation, either ((3.105)), ((3.106)), or ((3.108)), for the friction velocity u_τ. For the neutral case, u_τ can be solved for directly. For the stable and unstable cases, u_τ must be solved for iteratively because L appears in these equations and L depends on u_τ.
3. The surface shear stress is calculated as $\tau_s = \rho_s u_\tau^2$. For calculating left-hand-side Jacobian entries, the form (3.112) is used, where ψ' is zero for a neutral profile, $-\gamma_m z/L$ for a stable profile, and $\psi_h(z/L)$ for an unstable profile. The Jacobian entries follow directly from this form.
4. The user specified surface heat flux is applied to the enthalpy equation. Evaluation of surface temperature is not required for the boundary condition specification. However, if surface temperature is required for evaluation of other quantities within the code, the appropriate surface layer temperature profile should be used, either ((3.107)) or ((3.109)).

$$\tau_{s_i} = \lambda_s u_{||_i} = \frac{\kappa\rho u_\tau}{\log(z/z_0) - \psi'(z/L)}, \tag{3.112}$$

Moeng Wall Function

The Monin-Obukhov expressions only truly hold in a mean sense, and are not necessarily valid when used to specify an instantaneous value for the surface shear stress in a large eddy simulation. Moeng *[Moe84]* developed a local surface stress condition that utilizes horizontally-averaged quantities, for which the M-O relationships are assumed to hold. This boundary condition is derived by first assuming that the local tangential shear stress vector can be written using a drag law:

$$\tau_s = C_D u_{||_p} \mathbf{u}_{||_p} \tag{3.113}$$

Here, C_D is the drag coefficient, $\mathbf{u}_{||_p}$ is the surface-tangential velocity vector evaluated at the near-surface discretization point, and $u_{||_p}$ denotes the magnitude of this velocity vector.

An expression for τ_s is derived in the Appendix of Moeng *[Moe84]*, by writing the velocity as the sum of a horizontally averaged mean component and a fluctuation about this mean:

$$\mathbf{u}_{||_p} = \left\langle \mathbf{u}_{||_p} \right\rangle + \mathbf{u}''_{||_p}$$

There are two main assumptions in the derivation. The first is that the instantaneous version of the drag law((3.113)) is identical to the horizontally-averaged version. The second assumption is[1]

$$u''_{||_p} \mathbf{u}''_{||_p} \approx \left\langle u''_{||_p} \mathbf{u}''_{||_p} \right\rangle$$

After algebraic manipulations, the resulting vector expression is

$$\tau_s = \langle \tau_s \rangle \left(\frac{u_{||_p} \left\langle \mathbf{u}_{||_p} \right\rangle + \left\langle u_{||_p} \right\rangle \left[\mathbf{u}_{||_p} - \left\langle \mathbf{u}_{||_p} \right\rangle \right]}{\left\langle u_{||_p} \right\rangle \left\langle \mathbf{u}_{||_p} \right\rangle} \right) \tag{3.114}$$

The procedure to calculate the surface stress at a boundary integration point is as follows.

1. Calculate the horizontally-averaged quantities $\left\langle u_{||_p} \right\rangle$ and $\left\langle \mathbf{u}_{||_p} \right\rangle$.
2. Use the M-O velocity profile relationships *[Dye74]* to calculate an average friction velocity u_τ.
3. Use the relationship ((3.112)) to calculate the components of $\langle \tau_s \rangle$.
4. Calculate the local surface shear stress using((3.114)).

Jacobian entries are required to populate the left-hand side matrix for the terms resulting from ((3.114)). For convenience, we write the vector quantities as tensors with subscripts denoting the vector component indices. We need an expression for the sensitivity of the shear stress, applied at the boundary face integration point, to the velocity components at the l^{th} grid node, or $\frac{\partial \tau_{s_i}^{(ip)}}{\partial u_j^{(l)}}$.

Differentiating ((3.114)) with respect to $u_j^{(l)}$ gives

$$\frac{\partial \tau_{s_i}^{(ip)}}{\partial u_j^{(l)}} = \frac{\left\langle \tau_s^{(ip)} \right\rangle_i}{\left\langle u_{||}^{(ip)} \right\rangle} \frac{\partial u_{||}^{(ip)}}{\partial u_j^{(l)}} + \frac{\left\langle \tau_s^{(ip)} \right\rangle_i}{\left\langle u_{||i}^{(ip)} \right\rangle} \frac{\partial u_{||i}^{(ip)}}{\partial u_j^{(l)}} \tag{3.115}$$

The first term involves a partial derivative of the tangential velocity magnitude, while the second term involves the partial derivative of the tangential velocity component. Applying the chain rule to the first term gives

$$\frac{\partial u_{||}^{(ip)}}{\partial u_j^{(l)}} = \frac{1}{u_{||}^{(ip)}} u_{||k}^{(ip)} \frac{\partial u_{||k}^{(ip)}}{\partial u_j^{(l)}}, \tag{3.116}$$

where summation is implied over the repeated index k. The second partial derivative in ((3.115)) is seen to appear also in ((3.116)). It remains to write an expression for this derivative, which is done by first writing the tangential velocity vector at the boundary face integration point in terms of the Cartesian velocity components:

$$u_{||i}^{(ip)} = (1 - n_i n_j)\delta_{ij} u_i^{(ip)} - n_i n_j (1 - \delta_{ij}) u_j^{(ip)} \tag{3.117}$$

with summation over the j index. The integration point velocity components are calculated from the face nodes using

$$u_i^{(ip)} = \sum_{l=1}^{N_n} \phi^{(l)}(x_{ip}) u_i^{(l)} \tag{3.118}$$

Substituting ((3.118)) into ((3.117)), followed by ((3.117)) into ((3.116)) gives

$$\frac{\partial u_{||}^{(ip)}}{\partial u_j^{(l)}} = \frac{1}{u_{||}^{(ip)}} u_{||k}^{(ip)} \sum_{j=1}^{3} \sum_{l=1}^{N_n} (1 - n_i n_j)\delta_{ij} \phi^{(l)}(x_{ip}) - n_i n_j (1 - \delta_{ij}) \phi^{(l)}(x_{ip})$$

[1] Or, at least, that the difference between these quantities is small relative to other terms, see Moeng *[Moe84]*.

Turbulent Kinetic Energy, k_{sgs} LES model

When the boundary layer is assumed to be resolved, the natural boundary condition is a Dirichlet value of zero, $k_{sgs} = 0$.

When the wall model is used, a standard wall function approach is used with the assumption of equal production and dissipation.

The turbulent kinetic energy production term is consistent with the law of the wall formulation and can be expressed as,

$$P_{kw} = \tau_w \frac{\partial u_{\|}}{\partial y}. \tag{3.119}$$

The parallel velocity, $u_{\|}$, can be related to the wall shear stress by,

$$\tau_w \frac{u^+}{y^+} = \mu \frac{u_{\|}}{Y_p}. \tag{3.120}$$

Taking the derivative of both sides of Equation (3.120), and substituting this relationship into Equation (3.119) yields,

$$P_{kw} = \frac{\tau_w^2}{\mu} \frac{\partial u^+}{\partial y^+}. \tag{3.121}$$

Applying the derivative of the law of the wall formulation, Equation (3.90), provides the functional form of $\partial u^+ / \partial y^+$,

$$\frac{\partial u^+}{\partial y^+} = \frac{\partial}{\partial y^+} \left[\frac{1}{\kappa} \ln \left(E y^+ \right) \right] = \frac{1}{\kappa y^+}. \tag{3.122}$$

Substituting Equation (3.90) within Equation (3.121) yields a commonly used form of the near wall production term,

$$P_{kw} = \frac{{\tau_w}^2}{\rho \kappa u_\tau Y_p}. \tag{3.123}$$

Assuming local equilibrium, $P_k = \rho \epsilon$, and using Equation (3.123) and Equation (3.91) provides the form of wall shear stress is given by,

$$\tau_w = \rho C_\mu^{1/2} k. \tag{3.124}$$

Under the above assumptions, the near wall value for turbulent kinetic energy, in the absence of convection, diffusion, or accumulation is given by,

$$k = \frac{u_\tau^2}{C_\mu^{1/2}}. \tag{3.125}$$

This expression for turbulent kinetic energy is evaluated at the boundary faces of the exposed wall boundaries and is area-assembled to the nodal value for use in a Dirichlet condition.

Turbulent Kinetic Energy and Specific Dissipation SST Low Reynolds Number Boundary conditions

For the turbulent kinetic energy equation, the wall boundary conditions follow that described for the k_{sgs} model, i.e., $k = 0$.

A Dirichlet condition is also used on ω. For this boundary condition, the ω equation depends only on the near-wall grid spacing. The boundary condition is given by,

$$\omega = \frac{6\nu}{\beta_1 y^2},$$

which is valid for $y^+ < 3$.

Turbulent Kinetic Energy and Specific Dissipation SST High Reynolds Number Boundary conditions

The high Reynolds approach uses the law of the wall assumption and also follows the description provided in the wall modeling section with only a slight modification in constant syntax,

$$k = \frac{u_\tau^2}{\sqrt{\beta^*}}. \tag{3.126}$$

In the case of ω, an analytic expression is known in the log layer:

$$\omega = \frac{u_\tau}{\sqrt{\beta^*}\kappa y},$$

which is independent of k. Because all these expressions require y to be in the log layer, they should absolutely not be used unless it can be guaranteed that $y^+ > 10$, and $y^+ > 25$ is preferable. Automatic blending is not currently supported.

Solid Stress

The boundary conditions applied are either force provided by a static pressure,

$$F_i^n = \int \bar{P} n_i dS, \tag{3.127}$$

or a Dirichlet condition, i.e., $u_i = u_i^{spec}$, on the displacement field. Above, F_i^n is the force for component i due to a prescribed [static] pressure.

Intensity

The boundary condition for each intensity assumes a grey, diffuse surface as,

$$I\left(s\right) = \frac{1}{\pi}\left[\tau\sigma T_\infty^4 + \epsilon\sigma T_w^4 + \left(1-\epsilon-\tau\right)K\right]. \tag{3.128}$$

Open Boundary Condition

Open boundary conditions require far more care. In general, open bcs are assembled by iterating faces and the boundary integration points on the exposed face. The parent element is also required since oftentimes gradients are used (for momentum). For an open boundary condition the flow can either leave or enter the domain depending on what the computed mass flow rate at the exposed boundary integration point is.

Continuity

For continuity, the boundary mass flow rate must also be computed. This value is stored and used for the other equations that require advection. The same formula is used for the pressure-stabilized mass flow rate. However, the local pressure gradient for each boundary contribution is based on the difference between the interior integration point and the user-specified pressure which takes on the boundary value. The interior integration point is determined by linear interpolation. For CVFEM, full elemental averaging is used while in EBVC discretization, the midpoint value between the nearest node and opposing node to the boundary integration point is used. In both discretization approaches, non-orthogonal corrections are required. This procedure has been very important for stability for CVFEM tet-based meshes where a natural non-orthogonality exists between the boundary and interior integration point.

Momentum

For momentum, the normal component of the stress is subtracted out we subtract out the normal component of the stress. The normal stress component for component i can be written as $F_k n_k n_i$. The tangential component for component i is simply, $F_i - F_k n_k n_i$. As an example, the tangential viscous stress for component x is,

$$F_x^T = F_x - (F_x n_x + F_y n_y) n_x,$$

which can be written in general component form as,

$$F_i^T = F_i(1 - n_i n_i) - \sum_{i!=j} F_j n_i n_j.$$

Finally, the normal stress contribution is applied based on the user specified pressure,

$$F_i^N = P^{Spec} A_i.$$

For CVFEM, the face gradient operators are used for the thermal stress terms. For EBVC discretization, from the boundary integration point, the nearest node (the "Right" state) is used as well as the opposing node (the "Left" state). The nearest node and opposing node are used to compute gradients required for any derivatives. This equation follows the standard gradient description in the diffusion section with non-orthogonal corrections used. In this formulation, the area vector is taken to be the exposed area vector. Non-orthogonal terms are noted when the area vector and edge vector are not aligned.

For advection, If the flow is leaving the domain, we simply advect the nearest nodal value to the boundary integration point. If the flow is coming into the domain, we simply confine the flow to be normal to the open boundary integration point area vector. The value entrained can be the nearest node or an upstream velocity value defined by the edge midpoint value.

Mixture Fraction, Enthalpy, Species, k_{sgs}, k and ω

Open boundary conditions assume a zero normal gradient. When flow is entering the domain, the far-field user supplied value is used. Far field values are used for property evaluations. When flow is leaving the domain, the flow is advected out consistent with the choice of interior advection operator.

Symmetry Boundary Condition

Continuity, Mixture Fraction, Enthalpy, Species, k_{sgs}, k and ω

Zero diffusion is applied at the symmetry bc.

Momentum

A symmetry boundary is one that is described by removal of the tangential stress. Therefore, only the normal component of the stress is applied:

$$F_x^n = (F_x n_x + F_y n_y) n_x,$$

which can be written in general component form as,

$$F_i^n = F_j n_j n_i.$$

Periodic Boundary Condition

A parallel multiple-periodic boundary condition is supported. Mappings are created between master/slave surface node pairs. The node pairs are obtained from a parallel search and are expected to be unique. The node pairs are used to map the slave global id to that of the master. This allows the linear system to include matrix rows for only a subset of the overall set of nodes. Moreover, a periodic assembly for assembled quantities is managed via: $m+ = s$ and $s = m$, where m and s are master/slave nodes, respectively. For each parallel assembled quantity, e.g., dual volume, turbulence quantities, etc., this procedure is used. Periodic boxes and periodic couette and channel flow have been simulated in this code base. Tow forms of parallel searches exist and are supported (one through the Boost TPL and another through the STK Search module).

Non-conformal Boundary Condition

A surface-based approach based on a DG method has been discussed in the 2010 CTR summer proceedings by Domino, *[Dom10]*. Both the edge- and element-based formulation currently exists in the code base using the CVFEM and EBVC approaches.

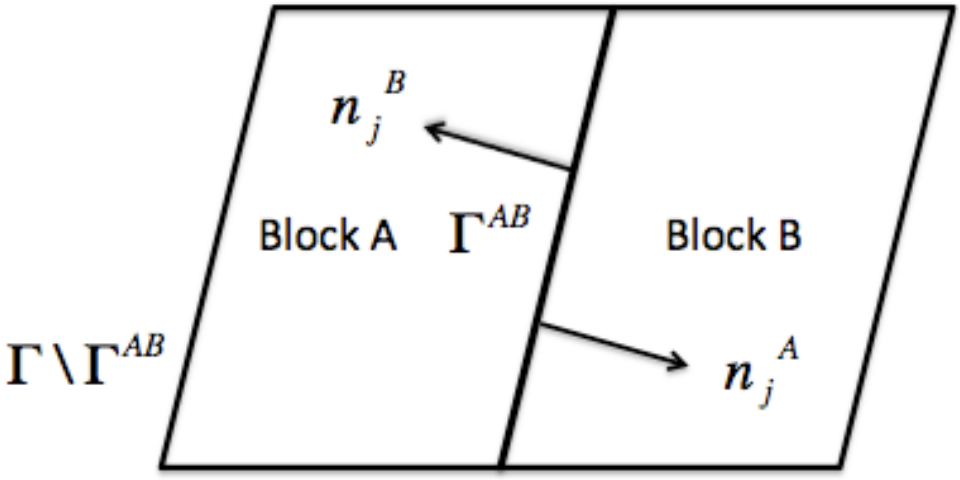

Fig. 3.6: Two-block example with one common surface, Γ_{AB}.

Consider two domains, A and B, which have a common interface, Γ_{AB}, and a set of interfaces not in common, $\Gamma \backslash \Gamma_{AB}$ (see Figure Fig. 3.6), and assume that the solution of the time-dependent advection/diffusion equation is to be solved in both domains. Each domain has a set of outwardly pointing normals. In this cartoon, the interface is well resolved, although in practice this may not be the case.

An interior penalty approach is constructed at each integration point at the exposed surface set. The numerical flux for a general scalar ϕ is constructed at the current integration point which is based on the current (A) and opposing (B) elemental contributions,

$$\int \hat{Q}^A dS = \int [\frac{(q_j^A n_j^A - q_j^B n_j^B)}{2} + \lambda^A(\phi^A - \phi^B)] dS^A + \dot{m}^A \frac{(\phi^A + \phi^B)}{2} + \eta \frac{|\dot{m}^A|}{2}(\phi^A - \phi^B), \tag{3.129}$$

where q_j^A and q_j^B are the diffusive fluxes computed using the current and opposing elements and normals are outward facing. The penalty coefficient λ^A contains the diffusive contributions averaged over the two elements,

$$\lambda^A = \frac{(\Gamma^A/L^A + \Gamma^B/L^B)}{2}. \tag{3.130}$$

Above, Γ^k is the diffusive flux coefficient evaluated at current and opposing element location, respectively, and L^k is an elemental length scale normal to the surface (again for current and opposing locations, A and :math‘B‘). When upwinding is activated, the value of η is unity.

As written in Equation (3.129), the default convection and diffusion term is a Galerkin approach, i.e., equally averaged between the current and opposing face. The standard advection term is given by,

$$\int \rho \hat{u}_j \phi n_j dS. \tag{3.131}$$

For surface A, the form is as follows:

$$\int \rho \hat{u}_j^A \phi n_j^A dS^A = \dot{m}^A \frac{\phi^A + \phi^B}{2}, \tag{3.132}$$

with the nonconformal mass flow rate given by,

$$\dot{m}^A = [\frac{(\rho u_j^A + \gamma(\tau G_j^A p - \tau \frac{\partial p^A}{\partial x_j}))n_j^A - (\rho u_j^B + \gamma(\tau G_j^B p - \tau \frac{\partial p^B}{\partial x_j}))n_j^B}{2} + \lambda^A(p^A - p^B)]dS^A. \tag{3.133}$$

In the above set of expressions, the consistent definition of $\hat{u}_j$, i.e., the convecting velocity including possible pressure stabilization terms, is retained.

As with the interior advection scheme, the mass flow rate involves pressure stabilization terms. The value of γ defines whether or not the full pressure stabilization terms are included in the mass flow rate expression. Equation (3.133) also forms the continuity nonconformal boundary contribution.

With the substitution of η to be unity, the effective convective term is as follows:

$$\int \rho \hat{u}_j \phi n_j^A dS^A = \frac{(\dot{m}^A + |\dot{m}^A|)\phi^A + (\dot{m}^A - |\dot{m}^A|)\phi^B}{2}. \tag{3.134}$$

Note that this form reduces to a standard upwind operator.

Since this algorithm is a dual pass approach, a numerical flux can be written for the integration point on block B,

$$\int \hat{Q}^B dS = \int [\frac{(q_j^B n_j^B - q_j^A n_j^A)}{2} + \lambda^B(\phi^B - \phi^A)]dS^A + \dot{m}^B \frac{(\phi^B + \phi^A)}{2} + \eta \frac{|\dot{m}^B|}{2}(\phi^B - \phi^A). \tag{3.135}$$

As with Equation (3.135), $\dot{m}^B$ (see Equation (3.136)) is of similar form to $\dot{m}^A$,

$$\dot{m}^B = [\frac{(\rho u_j^B + \gamma(\tau G_j^B p - \tau \frac{\partial p^B}{\partial x_j}))n_j^B - (\rho u_j^A + \gamma(\tau G_j^A p - \tau \frac{\partial p^A}{\partial x_j}))n_j^A}{2} + \lambda^A(p^B - p^A)]dS^B. \tag{3.136}$$

For low-order meshes with curved surface, faceting will occur. In this case, the outward facing normals may not be (sign)-unity factors of each other. In this case, it may be adventageous to define the opposing outward normal as, $n_j^B = -n_j^A$.

Domino, *[Dom10]* provided an overview of a FEM fluids implementation. In such a formulation, the interior penalty term appears, i.e.,

$$\int_{\Gamma_{AB}} \frac{\partial w^A}{\partial x_j} n_j \lambda(\phi^A - \phi^B) d\Gamma,$$

and

$$\int_{\Gamma_{BA}} \frac{\partial w^B}{\partial x_j} n_j \lambda(\phi^B - \phi^A) d\Gamma.$$

Although the sign of this term is often debated in the literature, the above set of expressions acts to increase penalty term stencil to include the full element contribution. As the CVFEM uses a piecewise-constant test function, this term is currently neglected.

Average fluxes are computed based on the current and opposing integration point locations. The appropriate DG terms are assembled as boundary conditions first with block A integration points as $current$ (integrations points for block B are $opposing$) and then with block B integration points as $current$ (surfaces for block A are, therefore, $opposing$). Figure Fig. 3.6 graphically demonstrates the procedure in which integration point values of the flux and penalty term are computed on the block A surface and at the projected location of block B.

A parallel search is conducted to project the current integration point location to the opposing element exposed face. The search, therefore, provides the isoparametric coordinates on the opposing element. Elemental shape functions and shape function derivatives are used to construct the numerical flux for both the edge- and element-based scheme. The location of the Gauss points on the current element are either the Gauss Labatto or Gauss Legendre locations (input file specification). For each equation (momentum, continuity, enthalpy, etc.) the numerical flux is computed at each exposed non-conformal surface.

As noted, for most equations other than continuity and heat condition, the numerical flux includes advection and diffusion contributions. The diffusive contribution is easily provided using elemental shape function derivatives at the current and opposing surface.

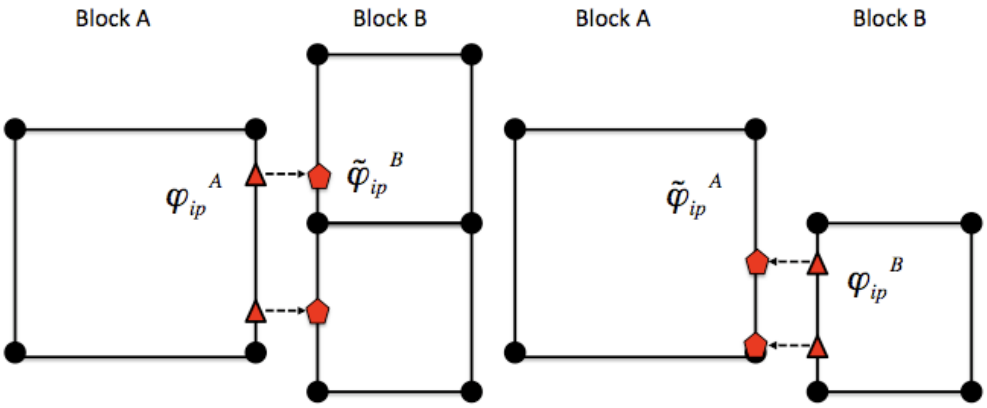

Fig. 3.7: Description of the numerical flux calculation for the DG algorithm. The value of fluxes and penalty values on the current block (A) and the opposing block (B) are used for the calculation of numerical fluxes. $\tilde{\varphi}$ represents the projected value.

Above, special care is taken for the value of the mass flow rate at the non-conformal interface. Also, note that the above written form does not upwind the advective flux, although the code allows for an upwinded approach. In general, the advective term contains contributions from both elements identified at the interface, specifically.

The penalty coefficient for the mass flow rate at the non-conformal boundary points is again a function of the blended inverse length scale at the current and opposing element surface location. The form of the mass flow rate above provides the continuity contribution and the form of the mass flow rate used in the scalar non-conformal flux contribution.

The full connectivity for element integration and opposing elements is within the linear system. As such, for sliding mesh configurations, the linear system connectivity graph changes each time step. Recent prototyping of the dG-based and the overset scheme has allowed this method to be used across both disparate low-order topologies (see Figure Fig. 3.8 and Figure Fig. 3.9).

Overset

Nalu supports simulations using an overset mesh methodology to model complex geometries. Currently the codebase supports two approaches to determine overset mesh connectivity:

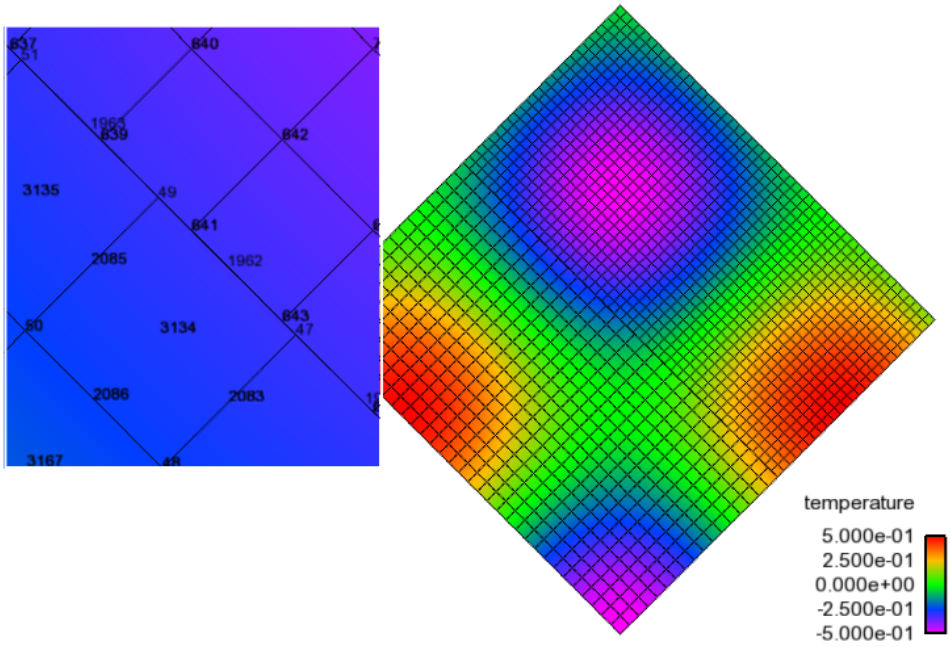

Fig. 3.8: A low-order and high-order block interface (P=1 quad4 and P=2 quad9) for a MMS temperature solution. In this image, the inset image is a close-up of the nodal Ids near the interface that highlights the quad4 and quad9 interface.

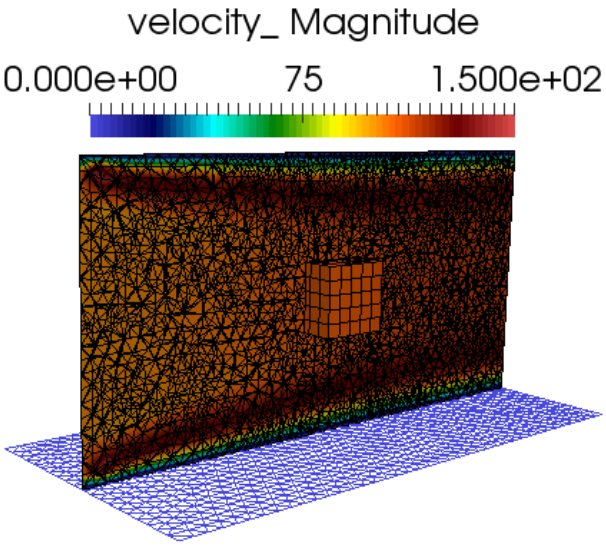

Fig. 3.9: Discontinuous Galerkin non-conformal interface mixed topology (hex8/tet4).

1. Overset mesh hole-cutting algorithm based on native STK search routines, and
2. Hole-cutting and donor/reception determination using the TIOGA (Topology Independent Overset Grid Assembly) TPL.

The native STK based overset grid assembly (OGA) requires no additional packages, but is limited to simple geometries where the search and hole-cutting procedure works only simple rectangular boundaries (for the inner mesh) that are aligned along the major axes. On the other hand, TIOGA based hole cutting is capable of performing overset grid assembly on arbitrary mesh geometries and orientation, supports generalized mesh motion, and can determine donor/recipient status with multiple meshes overlapping in the same space. A specific use-case for the need to perform OGA on multiple meshes is the simulation of a wind turbine in an atmospheric boundary layer, where the turbine blade, nacelle, and the background ABL mesh might all overlap near the rotor hub.

Overset Grid Assembly using Native STK Search

The overset descriptions begins with the basic background mesh (block 1) and overset mesh (block 2) depicted in Figure Fig. 3.10. Also shown in this figure is the reduction outer surface of block 2 (light blue). Elements within this reduced overset block will be determined by a parallel search. The collection of elements within this bounding box will be skinned to form a surface on which orphan nodes are placed. Elements within this volume are set in a new internally managed inactive block. These mesh entities are fully removed from the overall matrix for each dof. Elements within this volume are provided a masking integer element varibale of unity to select out of the visualizattion tool. Therefore, orphan nodes live at the external boundary of block 2 and along the reduced surface. The parallel search provides the mapping of orphan node and owning element from which the state can be constructed.

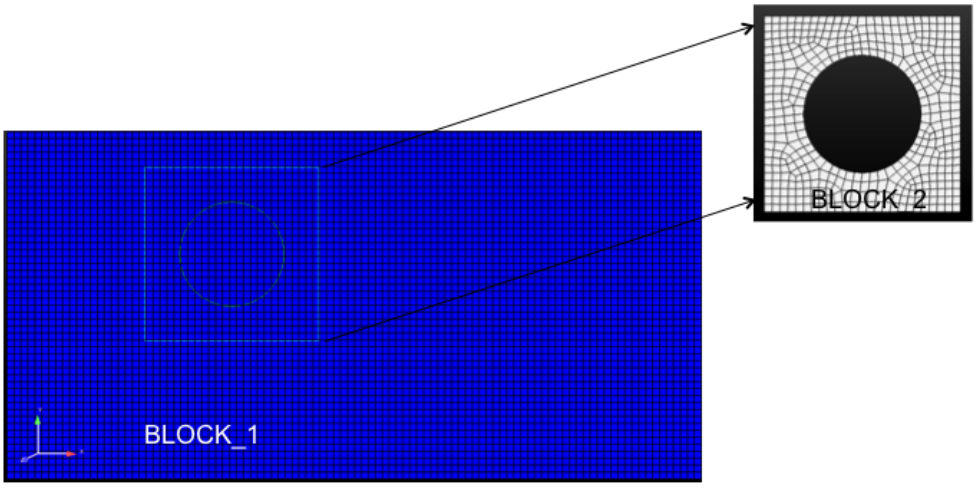

Fig. 3.10: Two-block use case describing background mesh (block 1) and overset mesh (block 2).

After the full search and overset initialization, this simple example yields the original block 1 and 2, the newly created inactive block 3, the original surface of the overset mesh and the new skinned surface (101) of the inactive block (Figure Fig. 3.11).

A simple heat conduction example is provided in Figure Fig. 3.12 where the circular boundary is set at a temperature of 500 with all external boundaries set to adiabatic.

As noted before, every orphan node lies within an owning element. Sufficient overlap is required to make the system well posed. A fully implicit procedure is provided by writing the orphan node value as a linear constraint of the owning element (Figure Fig. 3.13).

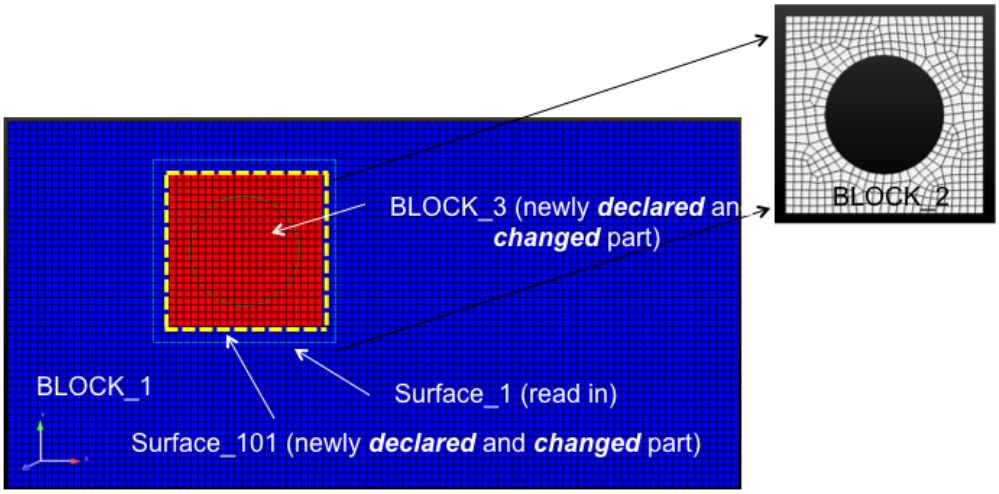

Fig. 3.11: Three-block and two surface, post over set initialization.

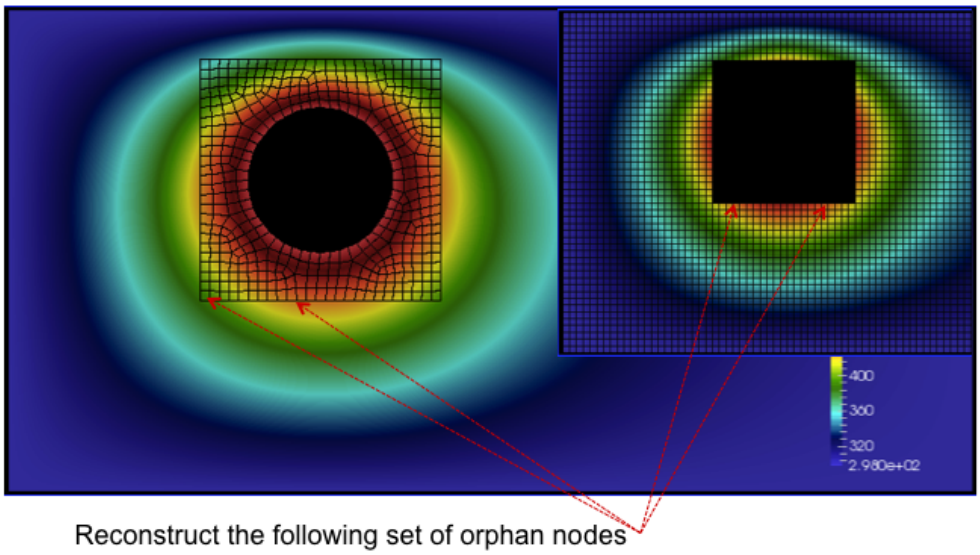

Fig. 3.12: A simple heat conduction example providing the overset mesh and donor orphan nodes.

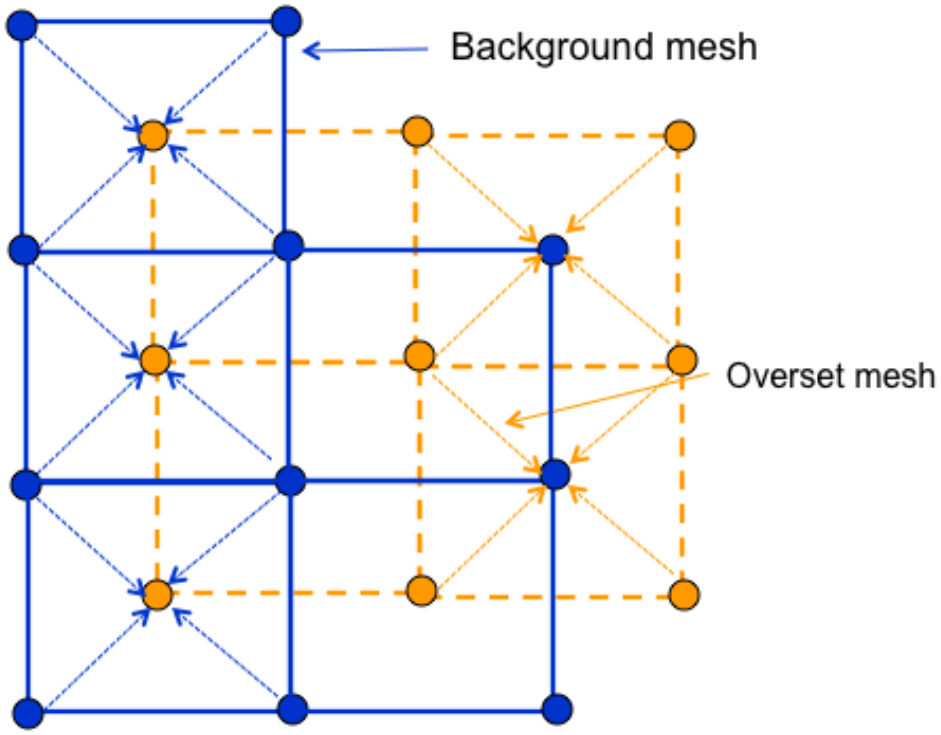

Fig. 3.13: Orphan nodes for background and overset mesh for which a fully implicit constraint equation is written.

For completeness, the constraint equation for any dof ϕ^o is simply,

$$\phi^o - \sum N_k \phi_k = 0. \tag{3.137}$$

As noted, full sensitivities are provided in the linear system by constructing a row entry with the columns of the nodes within the owning element and the orphan node itself.

Finally, a mixed hex/tet mesh configuration example (overset mesh is tet while background is hex) is provided in Figure Fig. 3.14.

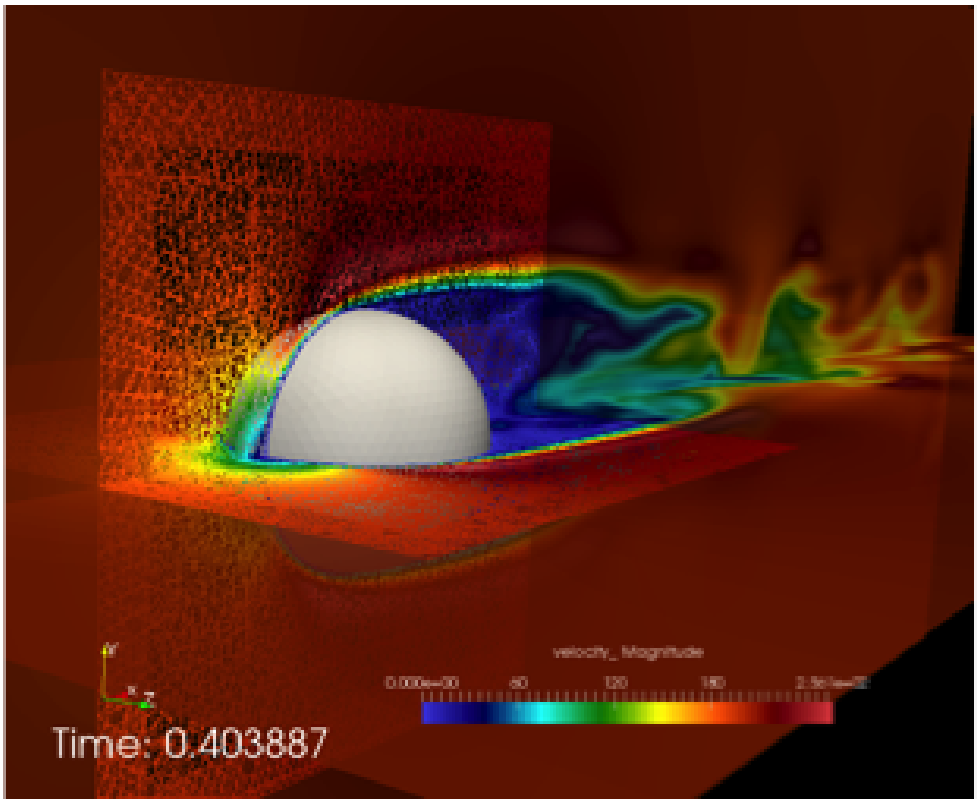

Fig. 3.14: Flow past a three-dimensional sphere using a hybrid topology (hex/tet) mesh configuration.

Overset Grid Assembly using TIOGA

Topology Independent Overset Grid Assembler (TIOGA) is an open-source connectivity package that was developed as an academic/research counterpart for PUNDIT (the overset grid assembler used in NASA/Army CREATE A/V program and HELIOS). The base library has been modified to remove the limitation where each MPI rank could only own one mesh block. The code has been extended to handle multiple mesh blocks per MPI rank to support Nalu's mesh decomposition strategies.

TIOGA uses a different nomenclature for overset mesh assembly. A brief description is provided here to familiarize users with the differences in nomenclature used in the previous section. When determining overset connectivity, TIOGA ends up assigning `IBLANK` values to the nodes in a mesh. The `IBLANK` field is an integer field that determines the status of the node which can be one of three states:

field point

A field point is a node that behaves as a normal mesh point, i.e., the equations are solved on these nodes and the linear system assembly proceeds as normal. The *field points* are indicated by an `IBLANK` value of 1.

fringe point

A fringe point is a receptor on the receving mesh where the solution field is mapped from the donor element. A fringe point is indicated by an `IBLANK` value of `-1`. Fringe points are how information is transferred between the participating meshes. Note that fringe points are referred to as *orphan points* in the STK based overset description.

hole point

A hole point is a node on a mesh that occurs inside a solid body being modeled in another mesh. These points have no valid solution for the equations solved and should not participate in the linear system.

In addition to the `IBLANK` status, the following terms are useful when using TIOGA

donor element

The element that is used to *interpolate* field data from donor mesh to a recipient mesh. While TIOGA provides flow interpolation routines, the current implementation in Nalu uses the `MasterElement` classes in Nalu to maintain consistency between the STK and the TIOGA overset implementations.

orphan points

The term orphan point is used differently in TIOGA than the STK based overset implementation. TIOGA refers to nodes as orphan points when there it cannot find a suitable donor element for those nodes that are considered fringe points. This can happen when the nodes on the enclosing element are themselves labeled fringe points.

Unlike the STK based hole cutting approach, that uses predefined bounding boxes to determine donor/receptor locations, TIOGA uses the element volume as the metric to determine the field and fringe points. The high level hole cutting algorithm can be described in the following steps:

- Determine and tag hole points that are fully enclosed within solid bodies, tag neighboring points to be fringe points.
- Determine and flag all mandatory fringe points, e.g., embedded boundaries of interior meshes.
- Determine fringe locations for the exterior meshes where information is transferred back from interior meshes to the exterior/background mesh.

In the current integration, only the hole-cutting and donor/receptor information is processed by the TIOGA library. The linear system assembly, specifically the constraint equations for the fringe points are managed by the same classes that are used with the native STK hole-cutting approach.

Figure Fig. 3.15 shows the field and fringe points as determined by TIOGA during the hole-cutting process. The central white region shows the mesh points of the interior mesh. The salmon colored region shows the overlapping field points where the flow equations are solved on both participating meshes. The green-ish boundary shows the mandatory fringe points for the interior mesh along its outer boundary. The interior boundary of the overlap region are the fringe points for the background mesh where information is transferred from the interior mesh. The extent of the overlap region is determined by the number of element layers necessary to ensure adequate separation between the fringe boundaries on the participating meshes.

Figure `tioga-overset-cyl` shows the resulting overset assembly for cylinder mesh and a background mesh with an intermediate refinement zone. The hole points (inside the cylinder) have been removed from the linear system for both the intermediate and background mesh. The magenta region shows the overlap of field points of the cylinder and the intermediate mesh. And the yellow region shows the overlap between the background and the intermediate mesh.

Figures Fig. 3.17 and Fig. 3.18 shown the velocity and vorticity contours for the flow past a cylinder simulated using the overset mesh methodology with TIOGA overset connectivity.

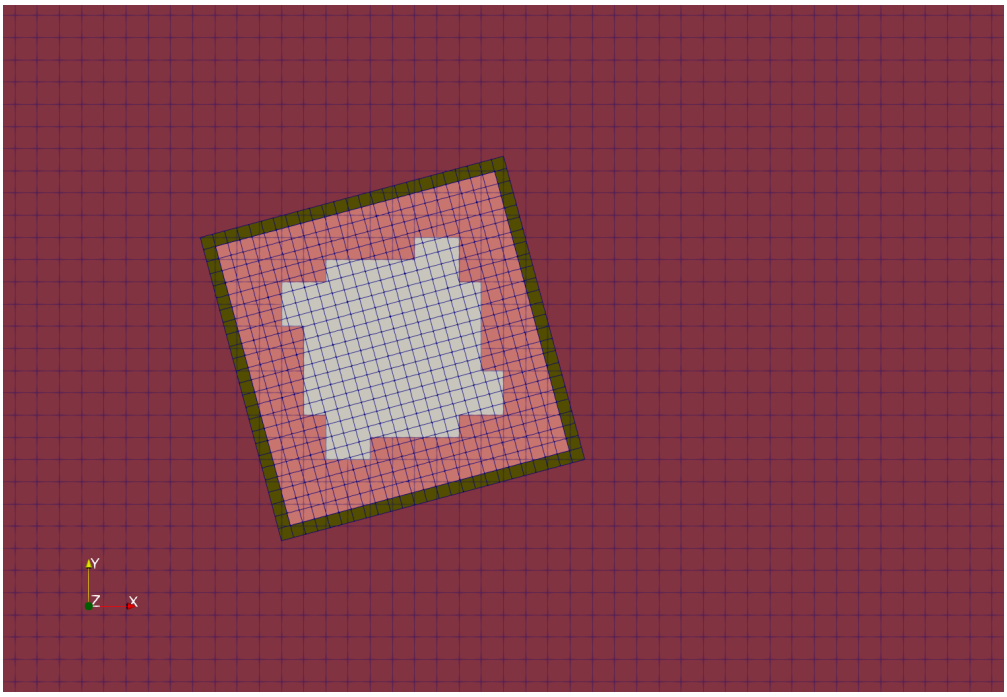

Fig. 3.15: TIOGA overset hole cutting for a rotated internal mesh configuration showing the field and fringe locations.

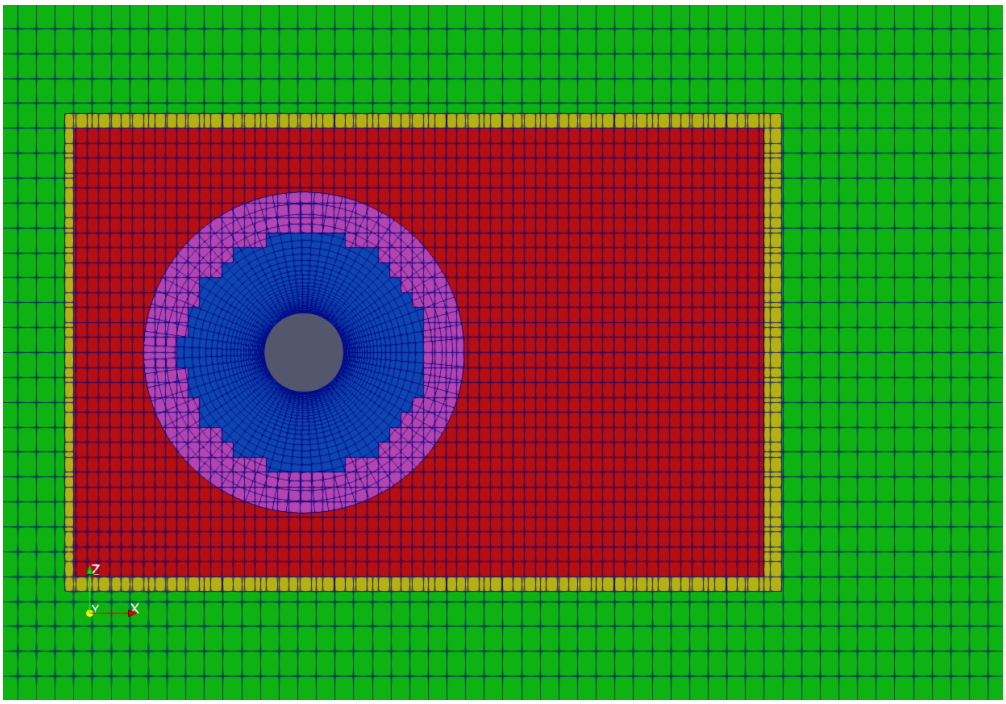

Fig. 3.16: Overset mesh configuration for simulating flow past a cylinder using a three mesh setup: near-body, body-fitted cylinder mesh, intermediate refined mesh, and coarse background mesh.

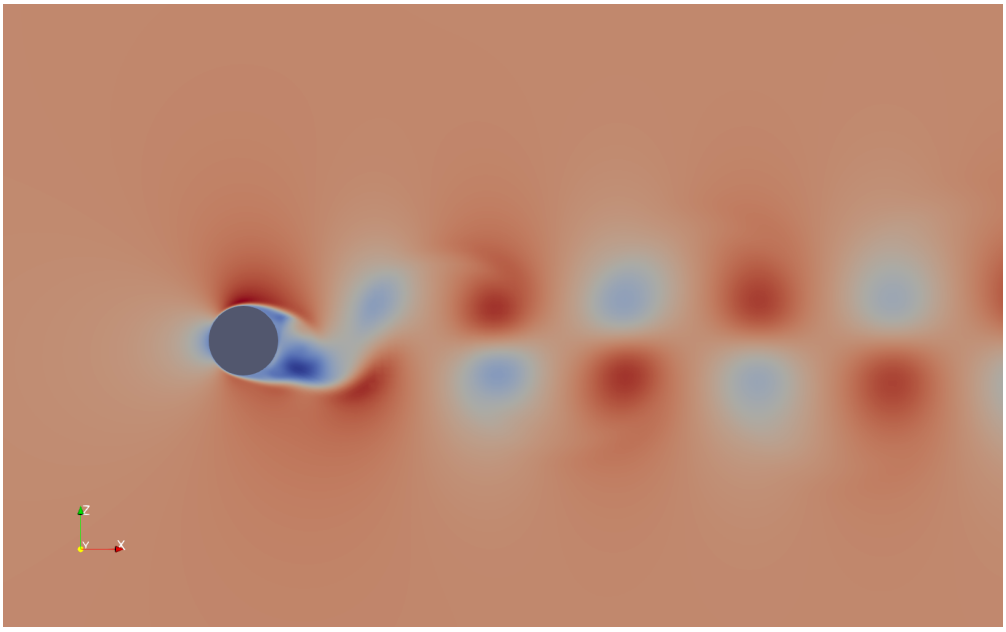

Fig. 3.17: Velocity field for a flow past cylinder simulating using an overset mesh methodology with TIOGA mesh connectivity approach.

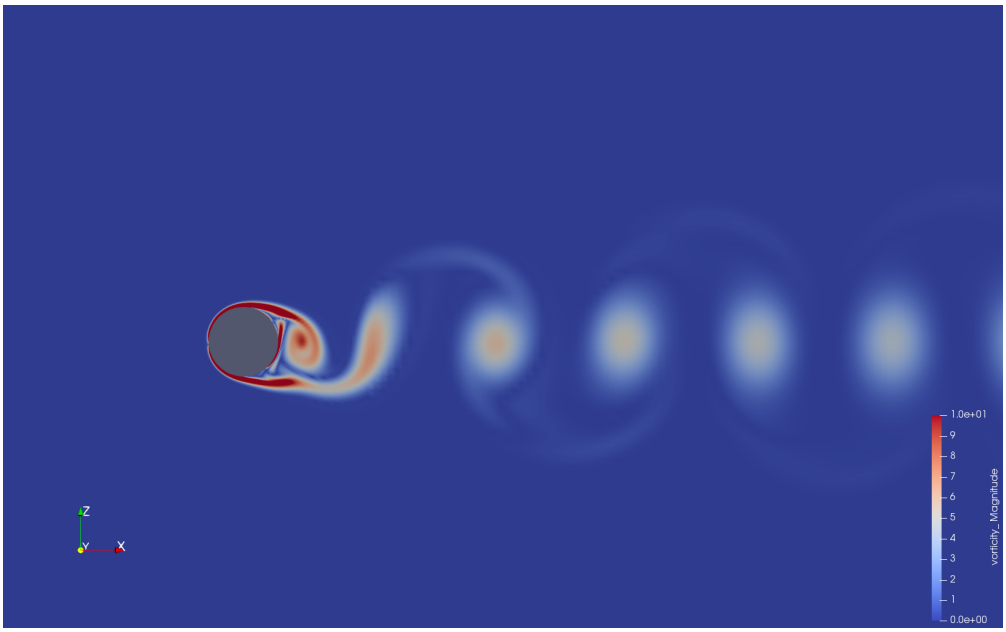

Fig. 3.18: Vorticity field for a flow past cylinder simulating using an overset mesh methodology with TIOGA mesh connectivity approach.

Property Evaluations

Property specification is provided in the material model section of the input file. Unity Lewis number assumptions for diffusive flux coefficients for mass fraction and enthalpy are assumed.

Density

At present, property evaluation for density is given by constant, single mixture fraction-based, HDF5 tables, or ideal gas. For ideal gas, we support a non-isothermal, non-uniform and even an acoustically compressible form.

Viscosity

Property evaluation for viscosity is given by constant, single mixture fraction-based, simple tables or Sutherland's three coefficient as a function of temperature. When mixtures are used, either by reference or species transport, only a mass fraction-weighed approach is used.

Specific Heat

Property evaluation for specific heat is either constant of two-band standard NASA polynomials; again species composition weighting are used (either transported or reference).

Lame Properties

Lame constants are either of type constant or for use in mesh motion/smoothing geometric whereby the values are inversely proportional to the dual volume.

Coupling Approach

The classic low Mach implementation uses an incremental approximate pressure projection scheme in which nonlinear convergence is obtained using outer Picard loops. Recently a full study on coupling approaches has been conducted using ASC Algorithm funds. In this project, coupling methods ranging from fully implicit, fully coupled equal order pressure/velocity interpolation with pressure stabilization to explicit advection/diffusion pressure projection schemes. A brief summary of the results follows.

Specifically, five algorithms were considered and are as follows:

1. A monolithic scheme in which advection and diffusion are implicit using full analytical sensitivities,
2. Monolithic momentum solve with implicit advection/diffusion in the context of a fourth order stabilized incremental pressure projection scheme,
3. Monolithic momentum solve with explicit advection; implicit diffusion in the context of a fourth order stabilized incremental pressure projection scheme,
4. Segregated momentum solve with implicit advection/diffusion in the context of a fourth order stabilized incremental pressure porjectin scheme, and
5. Explicit momementum advection/diffusion predictor/corrector scheme in the context of a second order stabilized pressure-free approximate projection scheme.

Each of the above algorithms has been run in the context of a transient uniform flow low Mach flow. The emphasis of this project is transient flows. As such, the numbers below are to be cast in this context. If steady flows are desired, conclusions may be different. The slowdown of each implementation is relative to the core low Mach algorith, i.e., algorithm (4) above. Numbers less than unity represent a speed-up whereas numbers greater than unity represent a slow down: 1) 3.4x, 2) 1.2x, 3) 0.6x, 4) 1.0x, 5) 0.7x.

The above runs were made using a time step that corresponded to a CFL of slightly less than unity. In this particlar flow, a transitionally turbulent open jet, the diffusion time scale stability limit was not a factor. In other words, there existed no detailed boundary layer at the wall bounded flow at the ground plane. Results for a Reynolds number of 45000 back step also are similar to the above jet results.

In general, although a mixture of implicit diffusion and explicit advection seem to be the winning combination, this scheme is very sensitive to time step and must be used by an educated user. In general, the conclusions are, thus far, that the standard segregated pressure projection scheme is preferred.

The algorithm implemented in Nalu is a fourth order approximate projection scheme with monolithic momentum coupling. Evaluation of a predictor/corrector approach for reating flow is anticipated in the late FY15 time frame.

Errors due to Splitting and Stabilization

As noted in many of our papers, the error in the above method can be written in block form (let's relax the variable density nuance - or simple fold these extra terms into our operators). Here we specifically partition error into both splitting (the pressure projection aspect of the alg that factorizes the fully coupled system) and pressure stabilization. Note that when we run fully coupled simulation with the same pressure stabilization algorithm, the answers converge to the same result.

Below, also forgive the specific definitions of τ. In general, they represent a choice of projection and stabilization time scales. Finally, the Laplace operator, e.g., $\mathbf{L_2}$, have the τ's built into them.

$$\begin{bmatrix} \mathbf{A} & \mathbf{G} \\ \mathbf{D} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{u}^{n+1} \\ p^{n+1} \end{bmatrix} = \begin{bmatrix} \mathbf{f} \\ 0 \end{bmatrix} + \begin{bmatrix} (\mathbf{I} - \tau\mathbf{A})\mathbf{G}(p^{n+1} - p^n) \\ \epsilon(\mathbf{L_i}, \tau_i, \mathbf{D}, \mathbf{G}) \end{bmatrix} \tag{3.138}$$

where the error term that appears for the discrete continuity solve is given by,

$$\begin{aligned} \epsilon(\mathbf{L_i}, \tau_i, \mathbf{D}, \mathbf{G}) = ((\mathbf{L_1} - \mathbf{D}\tau_3\mathbf{G}) \\ -(\mathbf{L_2} - \mathbf{D}\tau_2\mathbf{G}))(p^{n+1} - p^n) \\ +(\mathbf{L_2} - \mathbf{D}\tau_2\mathbf{G})p^{n+1} \end{aligned} \tag{3.139}$$

For the sake of this write-up, let $\mathbf{L_1} = \mathbf{L_2}$ and $\tau_2 = \tau_3$.

Time discretization

Time integrators range from simple backward Euler or a second order three state scheme, BDF2.

A general time discretization approach can be written as,

$$\int \frac{\partial \rho\phi}{\partial t} dV = \int \frac{(\gamma_1 \rho^{n+1}\phi^{n+1} + \gamma_2 \rho^n \phi^n + \gamma_3 \rho^{n-1}\phi^{n-1})}{\Delta t} dV$$

where γ_i represent the appropriate factors for either Backward Euler or a three-point BDF2 scheme. In both discretization approaches, the value for density and other dofs are evaluated at the node. As such, the time contribution is a lumped mass scheme with the volume simply the dual volume. The topology over one loops to assemble system is simply the node. Although CVFEM affords the use of a consistent mass matrix, this scheme is not used at present.

Multi-Physics

The equation set required to support the energy sector is already represented as a multiphysics application. However, in some common cases of coupling including conjugate heat transfer and coupling to participating media radiation, an operator split method may be preferred. The general concept is to define multiple Nalu Realms that each own the mesh on which the particular physics is solved. Surface- and volume-based couplings are supported through linear interpolation of the coupling parameters.

A typical CHT application involves the coupling of a thermal response and fluid transport. The coupling occurs between the surface that shares the thermal equation and static enthalpy equation. Moreover, coupling to a PMR solve is a volume-based coupling. Multiple Realms are supported with multiple transfers.

In Nalu, the method to achieve coupling in CHT or RTE coupled systems is through the usage of the STK Transfer module. This allows for linear interpolation between disparate meshes. Advanced conservative transfers are being evaluated, however, are not yet implemented in the code base. In general, the STK Transfer interface allows for this design point.

For FSI, the usage of the transfer module is also expected.

Actuator Wind Turbine Aerodynamics Modeling

Theory

Wind turbine rotor, tower, and nacelle aerodynamic effects can be modeled using actuator representations. Compared to resolving the geometry of the turbine, actuator modeling alleviates the need for a complex body-fitted meshes, can relax time step restrictions, and eliminates the need for turbulence modeling at the turbine surfaces. This comes at the expense of a loss of fine-scale detail, for example, the boundary layers of the wind turbine surfaces are not resolved. However, actuator methods well represent wind turbine wakes in the mid to far downstream regions where wake interactions are important.

Actuator methods usually fall within the classes of disks, lines, surface, or some blend between the disk and line (i.e., the swept actuator line). Most commonly, the force over the actuator is computed, and then applied as a body-force source term, f_i, to the Favre-filtered momentum equation

$$\int \frac{\partial \bar{\rho}\tilde{u}_i}{\partial t}\mathrm{d}V + \int \bar{\rho}\tilde{u}_i\tilde{u}_j n_j \mathrm{d}S + \int \bar{P} n_i \mathrm{d}S = \int \bar{\tau}_{ij} n_j \mathrm{d}S + \int \tau_{u_i u_j} n_j \mathrm{d}S + \int \left(\bar{\rho} - \rho_\circ\right) g_i \mathrm{d}V + \int f_i \mathrm{d}V, \tag{3.140}$$

The body-force term f_i is volumetric and is a force per unit volume. The actuator forces, F_i', are not volumetric. They exist along lines or on surfaces and are force per unit length or area. Therefore, a projection function, g, is used to project the actuator forces into the fluid volume as volumetric forces. A simple and commonly used projection function is a uniform Gaussian as proposed by S{o}rensen and Shen *[SrensenS02]*,

$$g(\vec{r}) = \frac{1}{\pi^{3/2}\epsilon^3} e^{-(|\vec{r}|/\epsilon)^2},$$

where $\vec{r}$ is the position vector between the fluid point of interest to a particular point on the actuator, and ϵ is the width of the Gaussian, which determines how diluted the body force become. As an example, for an actuator line extending from $l = 0$ to L, the body force at point (x, y, z) due to the line is given by

$$f_i(x, y, z) = \int_0^L g\left(\vec{r}\left(l\right)\right) F_i'\left(l\right)\ \mathrm{d}l. \tag{3.141}$$

Here, the projection function's position vector is a function of position on the actuator line. The part of the line nearest to the point in the fluid at (x, y, z) has most weight.

The force along an actuator line or over an actuator disk is often computed using blade element theory, where it is convenient to discretize the actuator into a set of elements. For example, with the actuator line, the line is broken into discrete line segments, and the force at the center of each element, F_i^k, is computed. Here, k is the actuator element index. These actuator points are independent of the fluid mesh. This set of point forces is then projected onto the fluid mesh using any desired projection function, $g(\vec{r})$, as described above. This is convenient because the integral given in Equation (3.141) can become the summation

$$f_i(x,y,z) = \sum_{k=0}^{N} g(\vec{r}^k) F_i^k. \tag{3.142}$$

This summation well approximates the integral given in Equation (3.141) so long as the ratio of actuator element size to projection function width ϵ does not exceed a certain threshold.

Design

The initial actuator capability implemented in Nalu is focused on the actuator line algorithm. However, the class hierarchy is designed with the potential to add other actuator source terms such as actuator disk, swept actuator line and actuator surface capability in the future. The `ActuatorLineFAST` class couples Nalu with the third party library OpenFAST for actuator line simulations of wind turbines. OpenFAST (https://nwtc.nrel.gov/FAST), available from https://github.com/OpenFAST/openfast, is a aero-hydro-servo-elastic tool to model wind turbine developed by the National Renewable Energy Laboratory (NREL). The `ActuatorLineFAST` class will help Nalu effectively act as an inflow module to OpenFAST by supplying the velocity field information.

We have tested actuator line implementation to be reasonably scalable. Actuators require searches and parallel communication of blade element velocities and forces, so our implementation should be scalable. Scalability is affected by the number of actuator turbines, the actuator element density, and the resolution of the mesh surrounding the actuators (i.e., the number of mesh elements that will receive body force). Further testing on scalability is underway with the demonstration of this capability to simulate the OWEZ wind farm.

The actuator line implementation allows for flexible blades that are not necessarily straight (prebend and sweep). The current implementation requires a fixed time step when coupled to OpenFAST, but allows the time step in Nalu to be an integral multiple of the OpenFAST time step. Initially, a simple time lagged FSI model is used to interface Nalu with the turbine model in OpenFAST:

- The velocity at time step at time step 'n' is sampled at the actuator points and sent to OpenFAST,
- OpenFAST advances the turbines upto the next Nalu time step 'n+1',
- The body forces at the actuator points are converted to the source terms of the momentum equation to advance Nalu to the next time step 'n+1'.

We are currently working on advanced FSI algorithms along with verification using an MMS approach.

The actuator implementation is flexible enough to incorporate a variety of future wind turbine technology capabilities. For example, it is possible that the nacelle may actively tilt for wake steering. The actuator capability is also able to handle a variety of turbines types within one simulation. The current capability allows the modeling of not only the rotor with actuators, but also the tower. However, an aerodynamic model still needs to be implemented for the nacelle.

Testing

We need a set of tests to make sure the actuator is working properly. Here are some of the proposed tests:

1. Momentum balance: set up a test that compares the change in fluid momentum to the momentum extracted by the actuator model.
2. Velocity/force/position transfer: set up a test that assures that the velocity, forces, and blade position being passed between Nalu and FAST is consistent.

3. Lifting line theory comparison: does it make sense to have a test in which a stationary actuator line wing with elliptic chord is placed in the flow and make sure that the results are consistent with theory? We won't get back the exact theoretical answer because lifting line theory is pretty idealized, but maybe a good check?

Implementation

1. During the load phase - the turbine data from the yaml file is read and stored in an object of the `fast::fastInputs` class
2. During the initialize phase - The processor containing the hub of each turbine is found through a search and assigned to be the one controlling OpenFAST for that turbine. All processors controlling > 0 turbines initialize FAST, populate the map of `ActuatorLinePointInfo` and initialize element searches for all the actuator points associated with the turbines. For every actuator point, the elements within a specified search radius are found and stored in the corresponding object of the `ActuatorLinePointInfo` class.
3. Elements are ghosted to the owning point rank. We tried the opposite approach of ghosting the actuator points to the processor owning the elements. The second approach was found to peform poorly compared to the first method.
4. During the execute phase called every time step, we sample the velocity at each actuator point and pass it to OpenFAST. All the OpenFAST turbine models are advanced upto Nalu's next time step to get the body forces at the actuator points. We then iterate over the `ActuatorLinePointInfoMap` to assemble source terms:
    - For each element *e* within the search radius of an actuator point *k*, the effective lumped body force is calculated at the center of the element by multiplying the actuator force with the Gaussian projection at the center of the element as $F_e^k = g(\vec{r}_e^k)\, F_i^k$.
    - The `assemble_source_to_nodes` function then distributes the force F_e at the center of an element to a node i surrounding it proportional to the subcontrol volume corresponding to that node as $F_e^i = F_e\ (V_{scv}^i/V_e)$, where V_e is the volume of the element.

Restart capability

While Nalu itself supports a full restart capability, OpenFAST may not support a full restart capability for specific use cases. To account for this, the OpenFAST - C++ API supports two kinds of restart capabilities. To restart a Nalu - OpenFAST coupled simulation one must set *t_start* in the line commands to a positive non-zero value and set *simStart* to either *trueRestart* or *restartDriverInitFAST*. Use *trueRestart* when OpenFAST supports a full restart capability for the specific use case. *restartDriverInitFAST* will start OpenFAST from *t=0* again for all turbines and run upto the restart time and then run the coupled Nalu + OpenFAST simulation normally. During the Nalu - OpenFAST he sampled velocity data at the actuator nodes is stored in a *hdf5* file at every OpenFAST time step and then read back in when using the *restart*.

The command line options for the actuator line with coupling to OpenFAST looks as follows for two turbines:

```
actuator:
type: ActLineFAST
search_method: boost_rtree
search_target_part: Unspecified-2-HEX

n_turbines_glob: 2
dry_run:  False
debug:    False
t_start: 0.0
simStart: init # init/trueRestart/restartDriverInitFAST
t_max:    5.0
n_every_checkpoint: 100
```

```
Turbine0:
  procNo: 0
  num_force_pts_blade: 50
  num_force_pts_tower: 20
  epsilon: [ 5.0, 5.0, 5.0 ]
  turbine_base_pos: [ 0.0, 0.0, -90.0 ]
  turbine_hub_pos: [ 0.0, 0.0, 0.0 ]
  restart_filename: "blah"
  FAST_input_filename: "Test01.fst"
  turb_id:  1
  turbine_name: machine_zero

Turbine1:
  procNo: 0
  num_force_pts_blade: 50
  num_force_pts_tower: 20
  epsilon: [ 5.0, 5.0, 5.0 ]
  turbine_base_pos: [ 250.0, 0.0, -90.0 ]
  turbine_hub_pos: [ 250.0, 0.0, 0.0 ]
  restart_filename: "blah"
  FAST_input_filename: "Test02.fst"
  turb_id:  2
  turbine_name: machine_one
```

Topological Support

The currently supported elements are as follows: hex, tet, pyramid, wedge, quad, and tri. In general, hybrid meshes are fully supported for the edge-based scheme. For CVFEM, hybrid meshes are also supported, however, wedge and pyramid elements are not permitted at exposed open or symmetry boundaries. The remedy to the CVFEM constraint is to simply implement the exposed face gradient operators.

Adaptivity

Adaptivity is supported through usage of the Percept module. However, this code base has not yet been deployed to the open sector. As such, ifdef guards are placed within the code base. A variety of choices exist for the manner by which hanging nodes are removed in a vertex-centered code base.

A typical h-adapted patch of elements is shown in Figure Fig. 3.19. The "hanging nodes" do not have control volumes associated with them. Rather, they are constrained to be a linear combination of the two parent edge nodes. There is no element assembly procedure to compute fluxes for the "hanging sub-faces" associated with the hanging nodes that occur along the parent-child element boundary.

In general, for a vertex-centered scheme, the h-adaptive scheme is driven at the element level. Refinement occurs within the element and the topology of refined elements is the same as the parent element.

Aftosmis *[Aft94]* describes a vertex-centered finite-volume scheme on unstructured Cartesian meshes. A transitional set of control volumes are formed about the hanging nodes, shown in Figure Fig. 3.20. on unstructured meshes. This approach would require a series of specialized master elements to deal with the different transition possibilities.

Kallinderis *[KB89]* describes a vertex-centered finite-volume scheme on unstructured quad meshes. Hanging nodes are treated with a constraint condition. The flux construction for a node on a refinement boundary is based on the unrefined parent elements, leading to a non-conservative scheme.

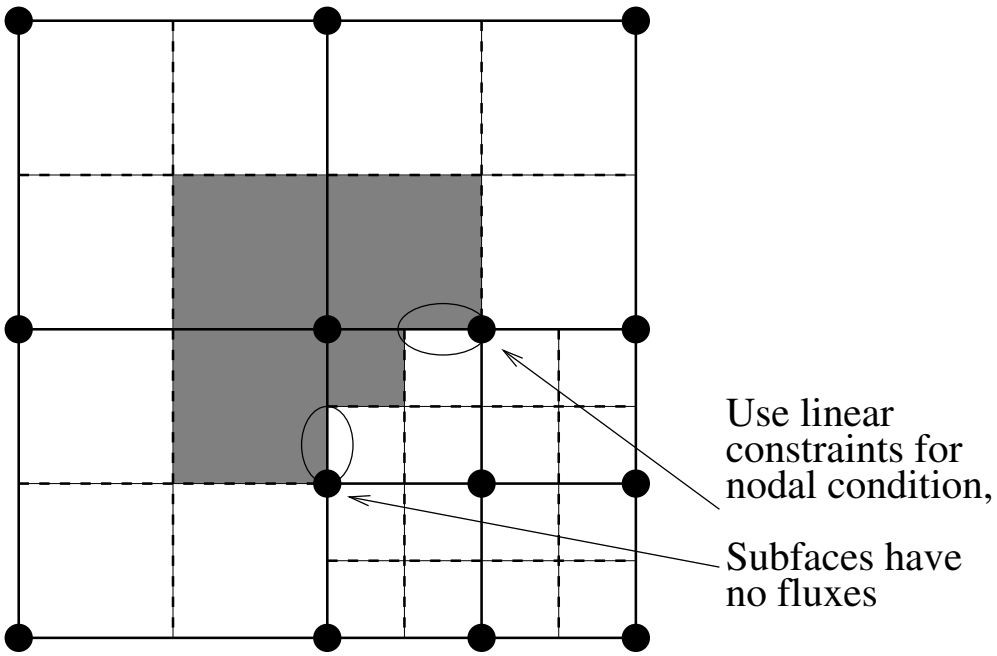

Fig. 3.19: Control volume definition on an h-adapted mesh with hanging nodes. (Four-patch of parent elements with refinement in bottom-right element.)

Kallinderis *[KV93]* also describes a vertex-centered finite-volume scheme on unstructured tetrahedral meshes. Hanging nodes are removed by splitting the elements on the "unrefined" side of the refinement boundary. Mavriplis *[Mav00]* uses a similar technique, however, extends it to a general set of heterogeneous elements, shown in Figure Fig. 3.21.

The future deployment of Percept will use the procedure of Mavriplis whereby hanging nodes are removed by neighbor topological changes. A variety of error indicators exists and a prototyped error transport equation appraoch for the one-equation k^{sgs} model has been tested for classic jet-in-crossflow configurations.

Prolongation and Restriction

Nodal variables are interpolated between levels of the h-adapted mesh hierarchy using the traditional prolongation and restriction operators defined over an element. The prolongation operation is used to compute values for new nodes that arise from element sub-division. The parent element shape functions are used to interpolate values from the parent nodes to the sub-divided nodes.

Prolongation and restriction operators for element variables and face variables are required to maintain mass flow rates that satisfy continuity. When adaptivity takes place, a code option to reconstruct the mass flow rates must be used. Whether or not a Poisson system must be created has been explored. More work is required to understand the nuances associated with prolongation, specifically with respect to possible dispersion errors.

Code Abstractions

The Nalu code base is a c++ code-base that significantly leverages the Sierra Toolkit and Trilinos infrastructure. This section is designed to provide a high level overview of the underlying abstractions that the code base exercises. For more detailed code information, the developer is referred to the Trilinos project (github.com). In the sections that follow, only a high level overview is provided.

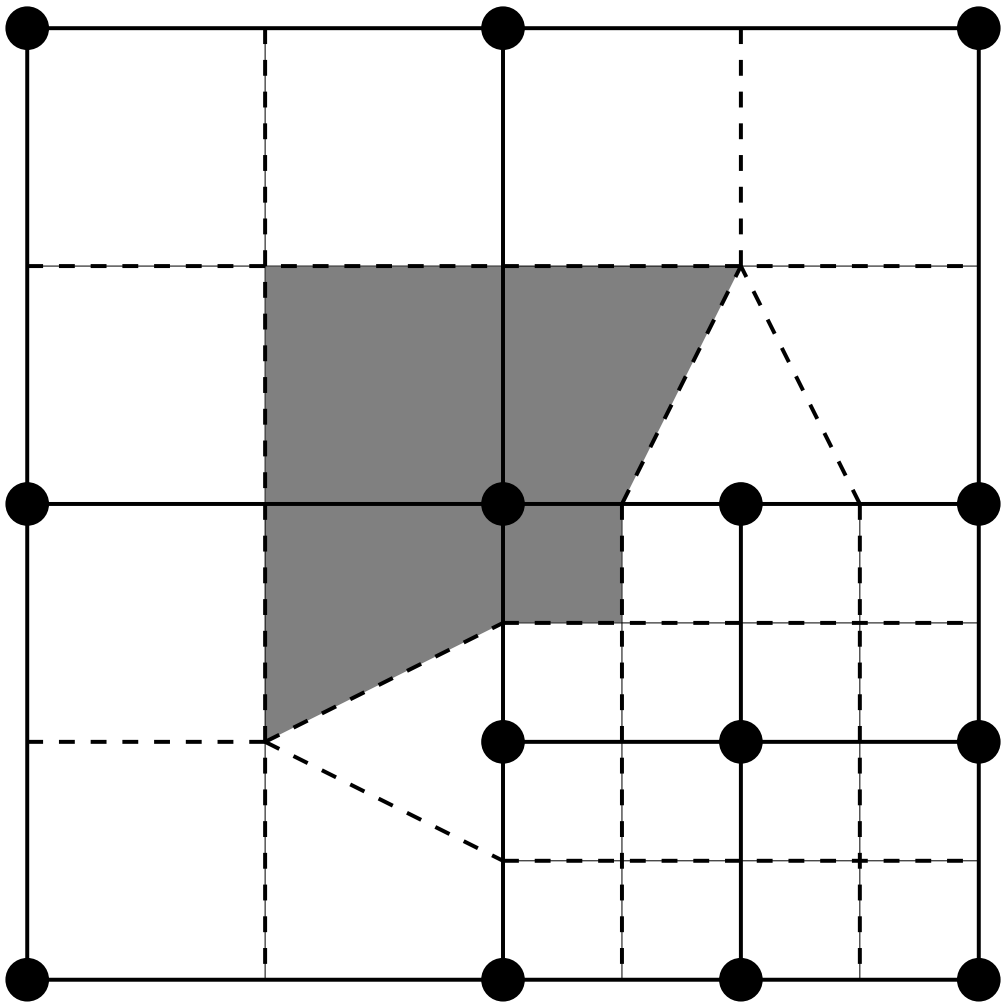

Fig. 3.20: Control volume definition on an h-adapted mesh with transition control volumes about the hanging nodes. (Four-patch of parent elements with refinement in bottom-right element.)

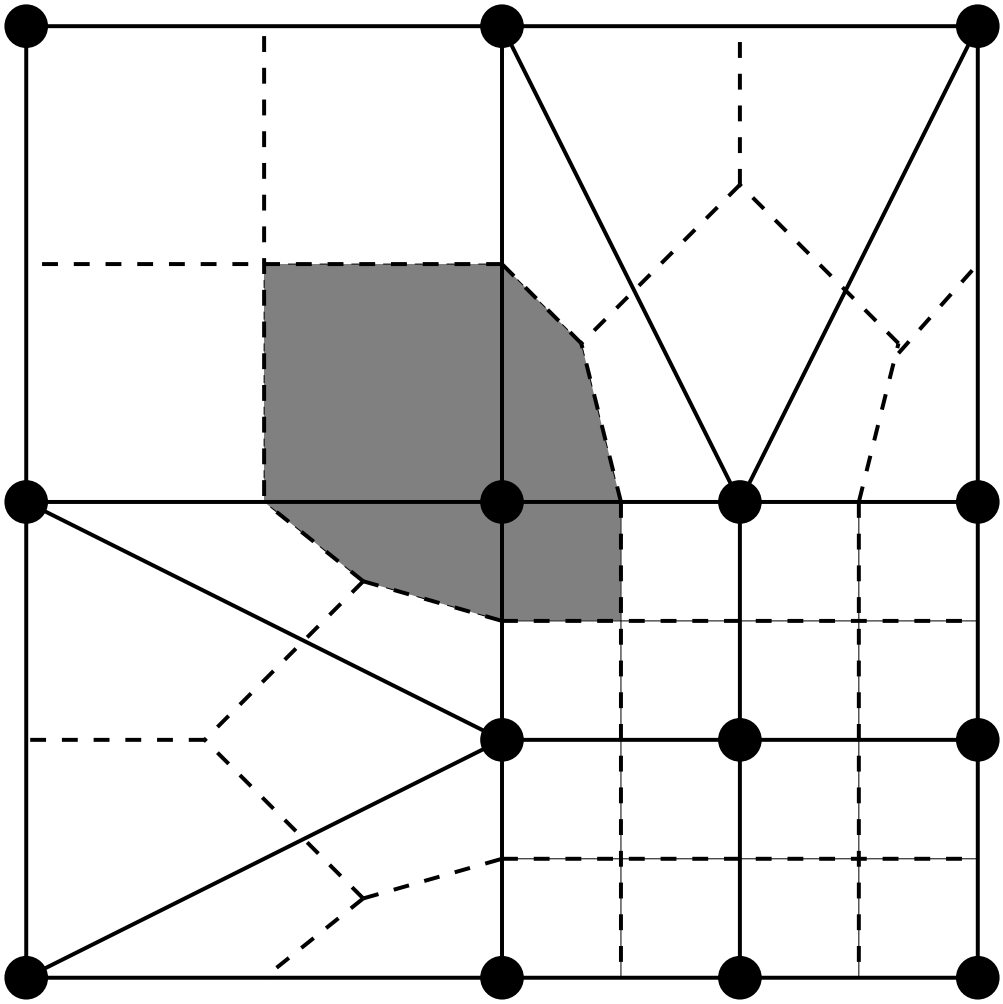

Fig. 3.21: Control volume definition on a heterogeneous h-adapted mesh with no hanging nodes. (Four-patch of parent elements with refinement in bottom-right element and splitting in adjacent parent elements.)

The developer might also find useful examples in the NaluUnit github repository as it contains a number of specialized implementations that are very small in nature. In fact, the Nalu code base emerged as a small testbed unit test to evaluate the STK infrastructure. Interestingly, the first "algorithm" implementation was a simple L_2 projected nodal gradient. This effort involved reading in a mesh, registering a nodal (vector) field, iterating elements and exposed surfaces to assemble the projected nodal gradient to the nodes of the mesh (in parallel). When evaluating kokkos, this algorithm was also used to learn about the parallel NGP abstraction provided.

Sierra Toolkit Abstractions

Consider a typical mesh that consists of nodes, sides of elements and elements. Such a mesh, when using the Exodus standard, will liekly be represented by a collection of "element blocks", "sidesets" and, possibly, "nodesets". The definition of the mesh (generated by the user through commercial meshing packages such as pointwise or ICM-CFD) will provide the required spatial definitions of the volume physics and the required boundary conditions.

An element block is a homegeneous collection of elements of the same underlying topology, e.g., HEXAHEDRAL-8. A sideset is a set of exposed element faces on which a boundary condition is to be applied. Finally, a nodeset is a collection of nodes. In general, nodesets are possibly output entities as the code does not exercise enforcing physics or boundary conditions on nodesets. Although Nalu supports an edge-based scheme, an edge, which is an entity connecting two nodes, is not part of the Exodus standard and must be generated within the STK infrastructure. Therefore, a particular discretization choice may require `stk::mesh::Entity` types of element, face (or side), edge and node.

Once the mesh is read in, a variety of routine operations are generally required. For example, a low-Mach physics equation set may want to be applied to `block_1` while inflow, open, symmetry, periodic and wall boundary conditions can be applied to a variety of sidesets. For example, `surface_1` might be of an "inflow" type. Therefore, the high level set of requirements on a mesh infrastructure might be to allow one to iterate parts of the mesh and, in the end, assemble a quantity to a nodal or elemental field.

Meta and Bulk Data

Meta and Bulk data are simply STK containers. MetaData is used to extract parts, extract ownership status, determine the side rank, field declaration, etc. BulkData is used to extract buckets, extract upward and downward connectivities and determine node count for a given entity.

Parallel Rules

In STK, elements are locally owned by a single rank. Elements may be ghosted to other parallel ranks through STK custom ghosting. Exposed faces are locally owned by the lower parallel rank. Nodes are also locally owned by the lower parallel rank and can also be shared by all parallel ranks touching them. Edges and internal faces (element:face:element connectivity) have the same rule of locally owned/shared and can also be ghosted. Again, edges and internal faces must be created by existing STK methods should the physics algorithm require them. In Nalu, the choice of element-based or edge-based is determined within the input file.

Connectivity

In an unstructured mesh, connectivity must be built from the mesh and can not be assumed to follow an assumed "i-j-k" data layout, i.e., structured. In general, one speaks of downward and upward relationships between the underlying entities. For example, if one has a particular element, one might like to extract all of the nodes connected to the element. Likewise, this represents a common opporation for faces and edges. Such examples are those in which downward relationships are required. However, one might also have a node and want to extract all of the connected elements to this node (consider some sort of patch recovery algorithm). STK provides the ability to extract such connectivities. In general, full downward and upward connectivities are created.

For example, consider an example in which one has a pointer to an element and wants to extract the nodes of this element. At this point, the developer has not been exposed to abstractions such as buckets, selectors, etc. As such, this is a very high level overview with more details to come in subsequent sections. Therefore, the scope below is to assume that from an element-k of a "bucket", b[k] (which is a collection of homogeneous RANK-ed entities) we will extract the nodes of this element using the STK bulk data.

```
// extract element from this bucket
stk::mesh::Entity elem = b[k];

// extract node relationship from bulk data
stk::mesh::Entity const * node_rels = bulkData_.begin_nodes(elem);
int num_nodes = bulkData_.num_nodes(elem);

// iterate nodes
for ( int ni = 0; ni < num_nodes; ++ni ) {
  stk::mesh::Entity node = node_rels[ni];

  // set connected nodes
  connected_nodes[ni] = node;

  // gather some data, e.g., density at state Np1,
  // into a local workset pointer to a std::vector
  p_density[ni] = *stk::mesh::field_data(densityNp1, node );
}
```

Parts

As noted before, a `stk::mesh::Part` is simply an abstraction that describes a set of mesh entities. If one has the name of the part from the mesh data base, one may extract the part. Once the part is in hand, one may iterate the underlying set of entities, walk relations, assemble data, etc.

The following example simply extracts a part for each vector of names that lives in the vector `targetNames` and provides this part to all of the underlying equations that have been created for purposes of nodal field registration. Parts of the mesh that are not included within the `targetNames` vector would not be included in the field registration and, as such, if this missing part was used to extract the data, an error would occur.

```
for ( size_t itarget = 0; itarget < targetNames.size(); ++itarget ) {
  stk::mesh::Part *targetPart = metaData_.get_part(targetNames[itarget]);

  // check for a good part
  if ( NULL == targetPart ) {
    throw std::runtime_error("Trouble with part " + targetNames[itarget]);
  }
  else {
    EquationSystemVector::iterator ii;
    for( ii=equationSystemVector_.begin(); ii!=equationSystemVector_.end(); ++ii )
      (*ii)->register_nodal_fields(targetPart);
  }
}
```

Selectors

In order to arrive at the precise parts of the mesh and entities on which one desires to operate, one needs to "select" what is useful. The STK selector infrastructure provides this.

In the following example, it is desired to obtain a selector that contains all of the parts of interest to a physics algorithm that are locally owned and active.

```
// define the selector; locally owned, the parts I have served up and active
stk::mesh::Selector s_locally_owned_union = metaData_.locally_owned_part()
  & stk::mesh::selectUnion(partVec_)
  & !(realm_.get_inactive_selector());
```

Buckets

Once a selector is defined (as above) an abstraction to provide access to the type of data can be defined. In STK, the mechanism to iterate entities on the mesh is through the `stk::mesh::bucket` interface. A bucket is a homogeneous collection of `stk::mesh::Entity`.

In the below example, the selector is used to define the bucket of entities that are provided to the developer.

```
// given the defined selector, extract the buckets of type ``element''
stk::mesh::BucketVector const& elem_buckets
  = bulkData_.get_buckets( stk::topology::ELEMENT_RANK,
                           s_locally_owned_union );

// loop over the vector of buckets
for ( stk::mesh::BucketVector::const_iterator ib = elem_buckets.begin();
      ib != elem_buckets.end() ; ++ib ) {
  stk::mesh::Bucket & b = **ib ;
  const stk::mesh::Bucket::size_type length   = b.size();

  // extract master element (homogeneous over buckets)
  MasterElement *meSCS = sierra::nalu::get_surface_master_element(b.topology());

  for ( stk::mesh::Bucket::size_type k = 0 ; k < length ; ++k ) {

    // extract element from this bucket
    stk::mesh::Entity elem = b[k];

    // etc...
  }
}
```

The look-and-feel for nodes, edges, face/sides is the same, e.g.,

• for nodes:

```
// given the defined selector, extract the buckets of type ``node''
stk::mesh::BucketVector const& node_buckets
  = bulkData_.get_buckets( stk::topology::NODE_RANK,
                           s_locally_owned_union );

// loop over the vector of buckets
```

• for edges:

```
// given the defined selector, extract the buckets of type ``edge''
stk::mesh::BucketVector const& edge_buckets
  = bulkData_.get_buckets( stk::topology::EDGE_RANK,
                           s_locally_owned_union );

// loop over the vector of buckets
```

• for faces/sides:

```
// given the defined selector, extract the buckets of type ``face/side''
stk::mesh::BucketVector const& face_buckets
  = bulkData_.get_buckets( metaData_.side_rank(),
                           s_locally_owned_union );

// loop over the vector of buckets
```

Field Data Registration

Given a part, we would like to declare the field and put the field on the part of interest. The developer can register fields of type elemental, nodal, face and edge of desired size.

• nodal field registration:

```
void
LowMachEquationSystem::register_nodal_fields(
  stk::mesh::Part *part)
{
  // how many states? BDF2 requires Np1, N and Nm1
  const int numStates = realm_.number_of_states();

  // declare it
  density_
    =  &(metaData_.declare_field<ScalarFieldType>(stk::topology::NODE_RANK,
                                                  "density", numStates));

  // put it on this part
  stk::mesh::put_field(*density_, *part);
}
```

• edge field registration:

```
void
LowMachEquationSystem::register_edge_fields(
  stk::mesh::Part *part)
{
  const int nDim = metaData_.spatial_dimension();
  edgeAreaVec_
    = &(metaData_.declare_field<VectorFieldType>(stk::topology::EDGE_RANK,
                                                 "edge_area_vector"));
  stk::mesh::put_field(*edgeAreaVec_, *part, nDim);
}
```

• side/face field registration:

```
void
MomentumEquationSystem::register_wall_bc(
  stk::mesh::Part *part,
  const stk::topology &theTopo,
  const WallBoundaryConditionData &wallBCData)
{
  // Dirichlet or wall function bc
  if ( wallFunctionApproach ) {
    stk::topology::rank_t sideRank
```

```
      = static_cast<stk::topology::rank_t>(metaData_.side_rank());
    GenericFieldType *wallFrictionVelocityBip
      =  &(metaData_.declare_field<GenericFieldType>
           (sideRank, "wall_friction_velocity_bip"));
    stk::mesh::put_field(*wallFrictionVelocityBip, *part, numIp);
  }
}
```

Field Data Access

Once we have the field registered and put on a part of the mesh, we can extract the field data anytime that we have the entity in hand. In the example below, we extract nodal field data and load a workset field.

To obtain a pointer for a field that was put on a node, edge face/side or element field, the string name used for declaration is used in addition to the field template type,

```
VectorFieldType *velocityRTM
  = metaData_.get_field<VectorFieldType>(stk::topology::NODE_RANK,
                                         "velocity");
ScalarFieldType *density
  = metaData_.get_field<ScalarFieldType>(stk::topology::NODE_RANK,
                                         "density");}

VectorFieldType *edgeAreaVec
  = metaData_.get_field<VectorFieldType>(stk::topology::EDGE_RANK,
                                         "edge_area_vector");

GenericFieldType  *massFlowRate
  = metaData_.get_field<GenericFieldType>(stk::topology::ELEMENT_RANK,
                                          "mass_flow_rate_scs");

GenericFieldType *wallFrictionVelocityBip_
  = metaData_.get_field<GenericFieldType>(metaData_.side_rank(),
                                          "wall_friction_velocity_bip");
```

State

For fields that require state, the field should have been declared with the proper number of states (see field declaration section). Once the field pointer is in hand, the specific field with state is easily extracted,

```
ScalarFieldType *density
  = metaData_.get_field<ScalarFieldType>(stk::topology::NODE_RANK,
                                         "density");
densityNm1_ = &(density->field_of_state(stk::mesh::StateNM1));
densityN_ = &(density->field_of_state(stk::mesh::StateN));
densityNp1_ = &(density->field_of_state(stk::mesh::StateNP1));
```

With the field pointer already in hand, obtaining the particular data is field the field data method.

• nodal field data access:

```
// gather some data (density at state Np1) into a local workset pointer
p_density[ni] = *stk::mesh::field_data(densityNp1, node );
```

• **edge field data access:** (from an edge bucket loop with the same selector as defined above)

```
stk::mesh::BucketVector const& edge_buckets
  = bulkData_.get_buckets( stk::topology::EDGE_RANK, s_locally_owned_union );
for ( stk::mesh::BucketVector::const_iterator ib = edge_buckets.begin();
      ib != edge_buckets.end() ; ++ib ) {
  stk::mesh::Bucket & b = **ib ;
  const stk::mesh::Bucket::size_type length   = b.size();

  // pointer to edge area vector and mdot (all of the buckets)
  const double * av = stk::mesh::field_data(*edgeAreaVec_, b);
  const double * mdot = stk::mesh::field_data(*massFlowRate_, b);

  for ( stk::mesh::Bucket::size_type k = 0 ; k < length ; ++k ) {
    // copy edge area vector to a pointer
    for ( int j = 0; j < nDim; ++j )
      p_areaVec[j] = av[k*nDim+j];

    // save off mass flow rate for this edge
    const double tmdot = mdot[k];
  }
}
```

High Level Nalu Abstractions

Realm

A realm holds a particular physics set, e.g., low-Mach fluids. Realms are coupled loosely through a transfer operation. For example, one might have a turbulent fluids realm, a thermal heat conduction realm and a PMR realm. The realm also holds a BulkData and MetaData since a realm requires fields and parts to solve the desired physics set.

EquationSystem

An equation system holds the set of PDEs of interest. As Nalu uses a pressure projection scheme with split PDE systems, the pre-defined systems are, LowMach, MixtureFraction, Enthalpy, TurbKineticEnergy, etc. New monolithic equation system can be easily created and plugged into the set of all equation systems.

In general, the creation of each equation system is of arbitrary order, however, in some cases fields required for MixtureFraction, e.g., `mass_flow_rate` might have only been registered on the low-Mach equation system. As such, if MixtureFraction is created before LowMachEOS, an error might be noted. This can be easily resolved by cleaning the code base such that each equation system is "autonomous".

Each equation system has a set of virtual methods expected to be implemented. These include, however, are not limited to registration of nodal fields, edge fields, boundary conditions of fixed type, e.g., wall, inflow, symmetry, etc.

CHAPTER 4

Sierra Low Mach Module: Nalu - Verification Manual

The SIERRA Low Mach Module: Nalu (henceforth referred to as Nalu, developed at Sandia, represents a generalized unstructured, massively parallel, variable density turbulent flow capability designed for energy applications. This code base began as an effort to prototype Sierra Toolkit, *[EWS+10]*, usage along with direct parallel matrix assembly to the Trilinos, *[HBH+03]*, Epetra and Tpetra data structure. However, the simulation tool has evolved as a tool to support a variety of research projects germane to the energy sector including wind aerodynamic prediction and traditional gas-phase combustion applications.

Introduction

The methodology used to evaluate the accuracy of each proposed scheme will be the method of manufactured solutions. The objective of code verification is to reveal coding mistakes that affect the order of accuracy and to determine if the governing discretized equations are being solved correctly. Quite often, the process of verification reveals algorithmic issues that would otherwise remain unknown.

In practice, a variety of comparison techniques exist for verification. For example, benchmark and code-to-code comparison are not considered rigorous due to the errors that exist in other code solutions, such as from discretization and iteration. Analytic solutions and the method of manufactured solutions remain the most powerful methods for code verification, since they provide a means to obtain quantitative error estimations in space and time.

Roache has made the distinction between code verification and calculation verification, where calculation verification involves grid refinement required for every problem solution to assess the magnitude, not order, of the discretization error. Discretization error, distinguished from modeling and iteration errors, is defined as the difference between the exact solution to the continuum governing equations and the solution to the algebraic systems representation due to discretization of the continuum equations. The order of accuracy can be determined by comparing the discretization error on successively refined grids. Thus, it is desirable to have an exact solution for comparision to determine the discretization errors.

2D Unsteady Uniform Property: Convecting Decaying Taylor Vortex

Verification of first-order and second-order temporal accuracy for the CVFEM and EBVC formulation in Nalu is performed using the method of manufactured solution (MMS) technique. For the unsteady isothermal, uniform laminar physics set, the exact solution of the convecting, decaying Taylor vortex is used.

$$u = u_o - cos(\pi(x - u_o t))sin(\pi(y - v_o t))e^{-2.0\omega t} \tag{4.1}$$

$$v = v_o + sin(\pi(x - u_o t))cos(\pi(y - v_o t))e^{-2.0\omega t} \tag{4.2}$$

$$p = -\frac{p_o}{4}(cos(2\pi(x - u_o t)) + cos(2\pi(y - v_o t)))e^{-4\omega t} \tag{4.3}$$

In this study, the constants u_o, v_o, and p_o are all assigned values of 1.0, and the viscosity μ is set to a constant value of 0.001. The value of ω is $\pi^2\mu$. This particular viscosity value results in a maximum cell reynolds number of twenty.

Temporal Order Of Accuracy Results

The temporal order of accuracy for the first order backward Euler and second order BDF2 are outlined in Figure Fig. 4.1 and Figure Fig. 4.2. Each of these simulations used a hybrid factor of zero to ensure pure second order central usage. A fixed Courant number of two was used for each of the three meshes (100x100, 200x200 and 400x400). The simulation was run out to 0.2 seconds and L_2 error norms were computed. The standard fourth order pressure stabilization scheme with time step scaling is used. This scheme is also known as the standard incremental pressure, approximate pressure projection scheme.

Two other pressure projection schemes have been evaluated in this study. Each represent a simplification of the standard pressure projection scheme. Figure Fig. 4.3 outlines three projection schemes: the first is when the projected nodal gradient appearing in the fourth-order pressure stabilization is lagged while the second is the classic pressure-free pressure approximate projection scheme with second order pressure stabilization. The third is the baseline fourth-order incremental pressure projection scheme. The error plots demonstrate that lagging the projected nodal gradient for pressure retains second order accuracy. However, as expected the pressure free pressure projection scheme is confirmed to be first order accurate given the first order splitting error noted in this fully implicit momentum solve.

The Steady Taylor Vortex will be used to verify the spatial accuracy for the full set of advection operators supported in Nalu.

Higher Order 2D Steady Uniform Property: Taylor Vortex

A higher order unstructured CVFEM method has been developed by Domino *[Dom14]*. A 2D structured mesh study demonstrating second order time and third order in space scheme has been demonstrated. The below work has emphasis on unstructured meshes.

Source Term Quadrature

Higher order accuracy is only demonstrated on solutions with source terms when a fully integrated approach is used. Lumping the source term evaluation is a second order error and is fully noted in the MMS study (not shown).

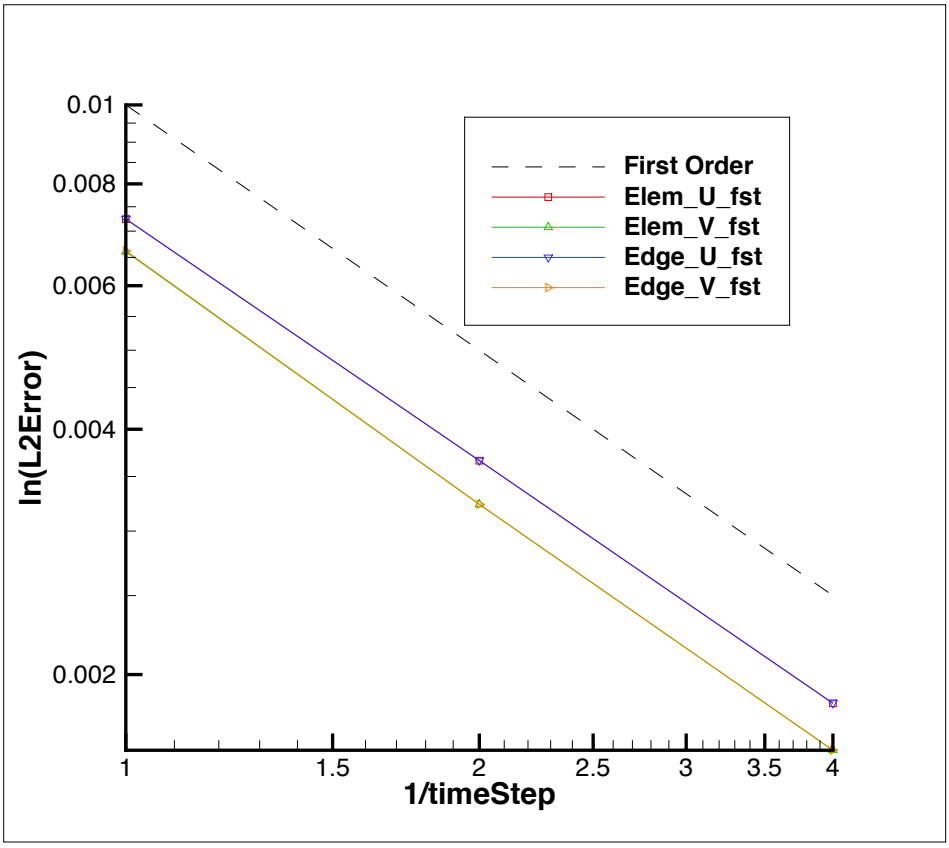

Fig. 4.1: Error norms as a function of timestep size for the u and v component of velocity using fourth order pressure stabilization with timestep scaling, backward Euler

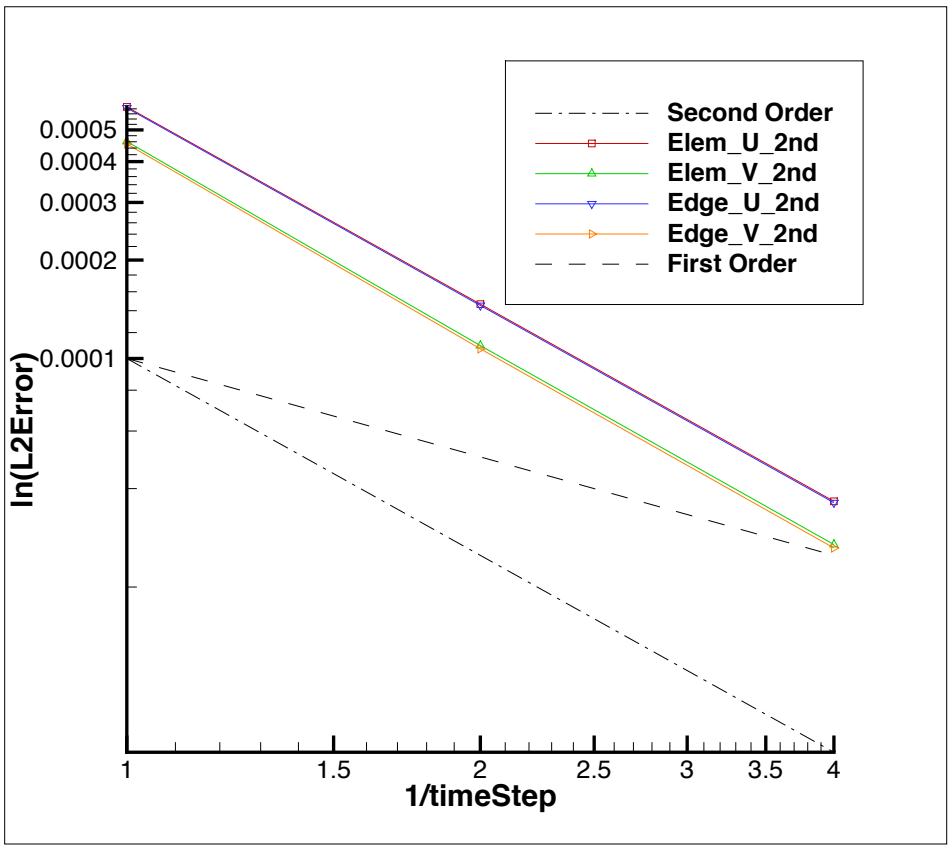

Fig. 4.2: Error norms as a function of timestep size for the u and v component of velocity using fourth order pressure stabilization with timestep scaling, BDF2

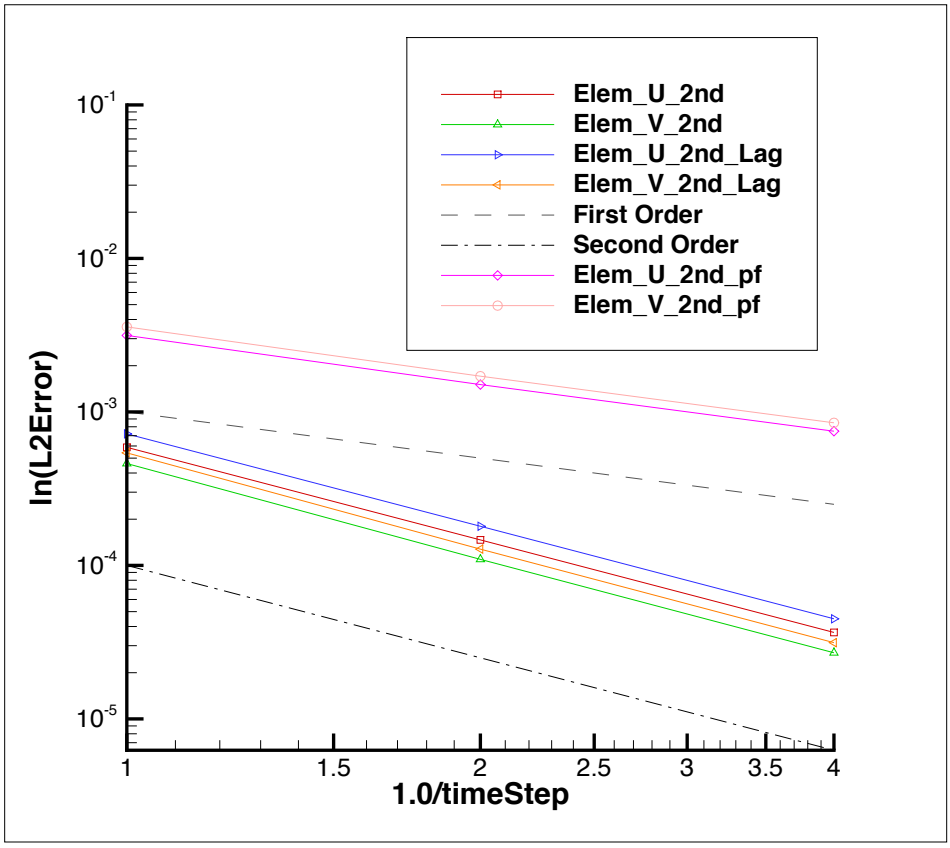

Fig. 4.3: Error norms as a function of timestep size for the u and v component of velocity using the lagged projected nodal pressure gradient and pressure-free pressure projection scheme; all with with timestep scaling, BDF2

Projected nodal gradients

Results show that one must use design order projected nodal gradients. Figure Fig. 4.4 demonstrates a code verification result for a steady thermal manufactured solution comparing lumped and consistent mass matrix approaches for the projected nodal gradient on a quadratic tquad mesh. In the lumped approach, a simple explicit algorithm is processed while for the consistent approach, a simple mass matrix inversion equation must be solved. The lumped approach is first order while the consistent approach retains the expected second order as the projected nodal gradient is expected to be order P. Both Dirichlet and periodic domains display the same order of convergence.

Momentum and Pressure

The steady taylor vortex exact solution was run on a quadratic tquad mesh. Figure Fig. 4.5 demonstrates the order of accuracy for projected nodal gradients (pressure) and the velocity field (x-component). Second order accuracy for the projected nodal gradient (pressure) and third order for the velocity field is realized when the consistent mass matrix approach is used for the projected nodal pressure gradient. Note that this term is used in the pressure stabilization approach. However, order of convergence for the projected nodal pressure gradient and velocity field is compromised when the lumped mass matrix approach is used for the pressure stabilization term. Note that both approaches use the fully integrated pressure gradient term in the momentum equation (i.e., $\int pn_i dS$). Therefore, the reduced order of integration for the projected nodal pressure gradient has consequence on the velocity field order of convergence.

Again, dirichlet (inflow) and periodic domains display the same order of convergence.

3D Steady Non-isothermal with Buoyancy

Building from the basic functional form of the Taylor Vortex, a non-isothermal solution (momentum, pressure and static enthalpy) is manufactured as follows:

$$\begin{aligned} u &= -u_o cos(a\pi x) sin(a\pi y) sin(a\pi z) \\ v &= +v_o sin(a\pi x) cos(a\pi y) sin(a\pi z) \\ w &= -w_o sin(a\pi x) sin(a\pi y) cos(a\pi z) \\ p &= -\frac{p_o}{4}(cos(2a\pi x) + cos(2a\pi y) + cos(2a\pi z)) \\ h &= +h_o cos(a_h\pi x) cos(a_h\pi y) cos(a_h\pi z) \end{aligned} \tag{4.4}$$

The equation of state is simply the ideal gas law,

$$\rho = \frac{P^{ref} M}{RT} \tag{4.5}$$

The simulation is run on a three-dimensional domain ranging from -0.05:+0.05 with constants $a, a_h, M, R, C_p, P^{ref}, T_{ref}, Pr, \mu$ equal to (20, 10, 30, 10, 0.01, 100, 300, 0.8, 0.00125), respectively.

At reference conditions, the density is unity. The effects of buoyancy are also provided by an arbitrary gravity vector of magnitude of approximately ten, $g_i = (-5, 6, 7)^T$. On this domain, the enthalpy ranges from zero to unity. Given the reference values, the temperature field ranges from 300K to 400K which is designed to mimic a current LES non-isothermal validation suite.

Edge- and element-based discretization (P=1) demonstrate second order convergence in the L_2 norm for u, v, w and temperature. This test is captured within the variableDensityMMS regression test suite.

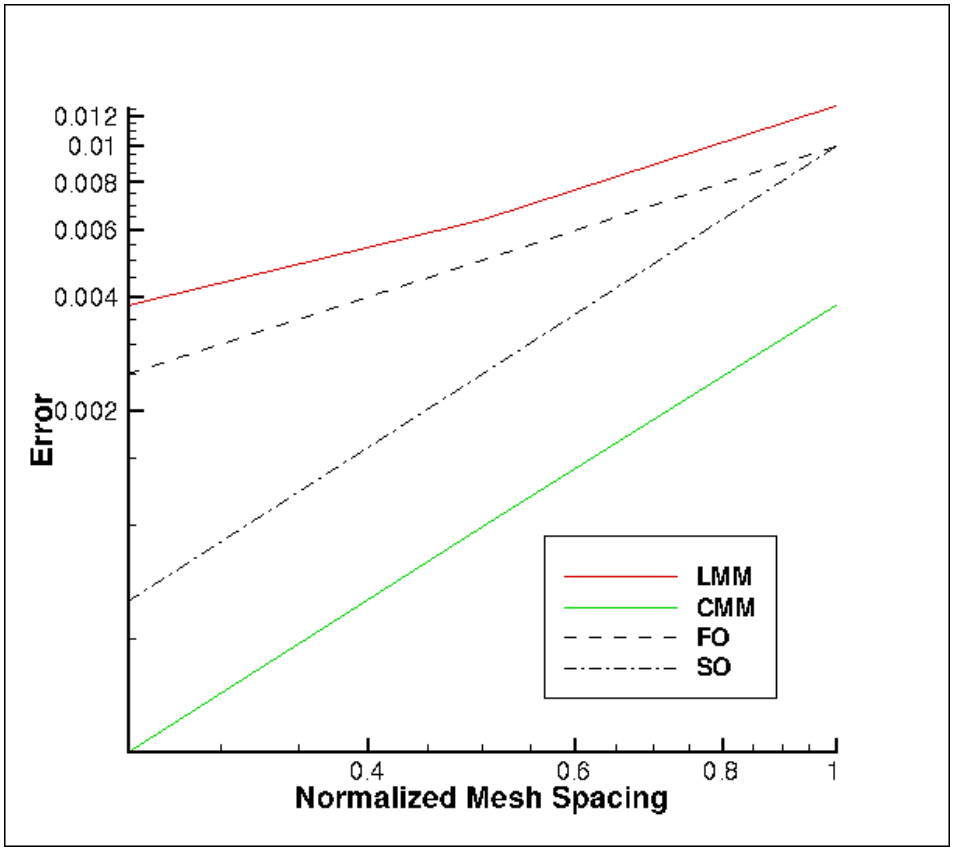

Fig. 4.4: Error norms as a function of mesh size for a CMM and LMM projected nodal gradient on a quadratic tquad mesh.

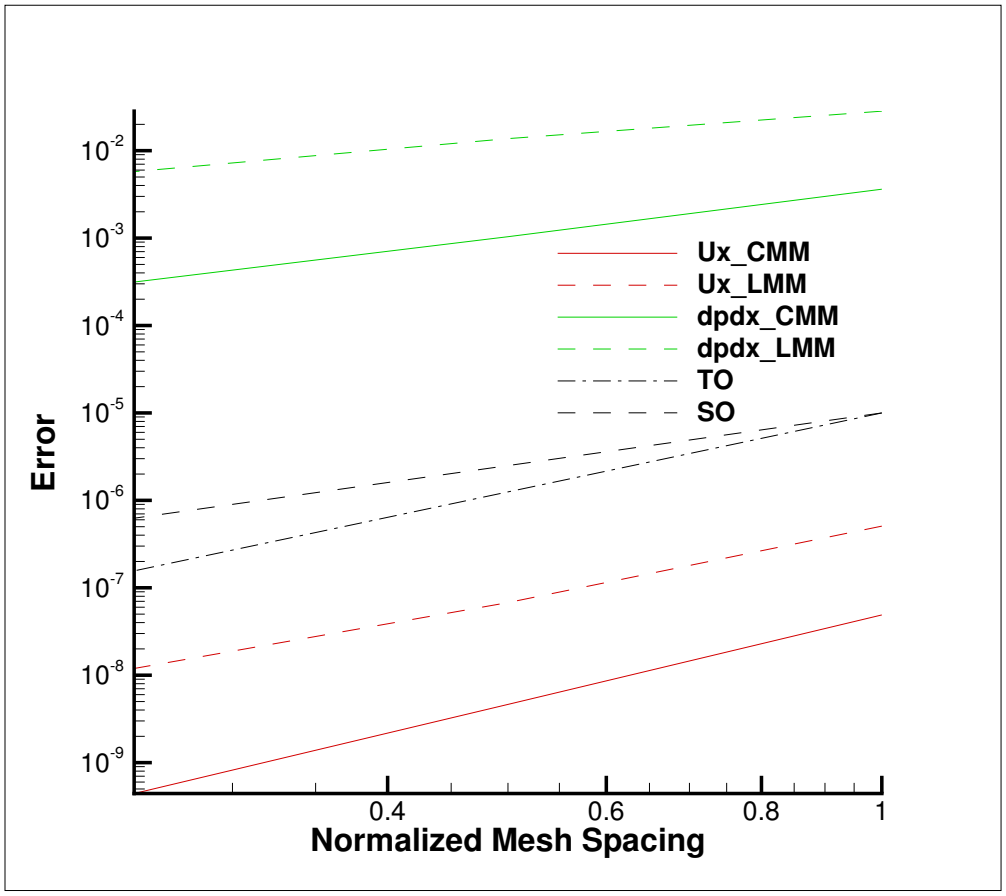

Fig. 4.5: Error norms as a function of mesh size for the Steady Taylor Vortex momentum and pressure gradient field.

3D Steady Non-uniform with Buoyancy

Building from the basic functional form of the Taylor Vortex, a non-uniform solution (momentum, pressure and mixture fraction) is manufactured as follows:

$$\begin{aligned} u &= -u_o cos(a\pi x) sin(a\pi y) sin(a\pi z) \\ v &= +v_o sin(a\pi x) cos(a\pi y) sin(a\pi z) \\ w &= -w_o sin(a\pi x) sin(a\pi y) cos(a\pi z) \\ p &= -\frac{p_o}{4}(cos(2a\pi x) + cos(2a\pi y) + cos(2a\pi z)) \\ z &= +z_o cos(a_z\pi x) cos(a_z\pi y) cos(a_z\pi z) \end{aligned} \tag{4.6}$$

The equation of state is simply the standard inverse mixture fraction property expression for density,

$$\rho = \frac{1}{\frac{z}{rho^P} + \frac{1-z}{rho^S}} \tag{4.7}$$

The simulation is run on a three-dimensional domain ranging from -0.05:+0.05 with constants $a, a_z, \rho^p, \rho^s, Sc, \mu$ equal to (20, 10, 0.1, 1.0, 0.8, 0.001), respectively.

At reference conditions, the density is that of the primary condition (0.1). The effects of buoyancy are also provided by an arbitrary gravity vector of magnitude of approximately ten, $g_i = (-5, 6, 7)^T$. On this domain, the mixture fraction ranges from zero to unity. This test case is designed to support the helium plume DNS study with primary and secondary density values of helium and air, respectively.

Edge- and element-based discretization (P=1) demonstrate second order convergence in the L_2 norm for u, v, w and mixture fraction. This test is captured within the variableDensityMMS regression test suite.

2D Steady Laplace Operator

The evaluation of the low-Mach Laplace (or diffusion operator) is of great interest to the core supported application space. Although the application space for Nalu is characterized by a highly turbulent flow, the usage of an approximate pressure projection scheme always makes the chosen Laplace form important. Although the element-based scheme is expected to be accurate, it can be problematic on high aspect ratio meshes as element-based schemes are not gauranteed to be monotonic for aspect ratios as low as $\sqrt{2}$ for FEM-based schemes and $\sqrt{3}$ for CVEM-based approaches (both when using standard Gauss point locations). Conversely, while the edge-based operator is accurate on high aspect ratio meshes, it suffers on skewed meshes due to both quadrature error and the inclusion of a non orthogonal correction (NOC).

In order to assess the accuracy of the Laplace operator, a the two-dimensional MMS temperature solution is used. The functional temperature field takes on the following form:

$$T = \frac{\lambda}{4}(cos(2a\pi x) + cos(2a\pi y)). \tag{4.8}$$

The above manufactured solution is run on three meshes of domain size of 1x1. The domain was first meshed as a triangular mesh and then converted to a tquad4 mesh. Therefore, non orthogonal correction (NOC) effects are expected for the edge-based scheme. In this study, both λ and a are unity. Either periodic or Dirichlet conditions are used for boundary conditions.

A brief overview of the diffusion operator tested is now provided. For more details, consult the theory manual. The general diffusion kernel is as follows:

$$-\int \Gamma \frac{\partial \phi}{\partial x_j} A_j. \tag{4.9}$$

The choice of the gradient operator at the integration point is a functin of the underlying method. For CVFEM, the gradient operator is provided by the standard shape function derivatives,

$$\frac{\partial \phi_{ip}}{\partial x_j} = \sum \frac{\partial N^{ip}_{j,k}}{\partial x_j} \phi_k. \tag{4.10}$$

For the edge-based scheme, a blending of an orthogonal gradient along the edge and a NOC is employed,

$$\frac{\partial \phi_{ip}}{\partial x_j} = \bar{G_j \phi} + \left[(\phi_R - \phi_L) - \bar{G_l \phi} dx_l \right] \frac{A_j}{A_k dx_k}. \tag{4.11}$$

In the above equation, $G_j \phi$ is a projected nodal gradient. The general equation for this quantity is

$$\int w_I G_j \phi dV = \int w_i \frac{\partial \phi}{\partial x_j} dV. \tag{4.12}$$

Possible forms of this include either lumped or consistent mass (the later requires a global equation solve) with either the full CVFEM stencil or the edge-based (reduced) stencil. The above equation can even be applied within the element itself for a simple, local integration step that provides a piecewise constant gradient over the element.

The simulation study is run with the following diffusion operators: 1) the standard CVFEM operator, 2) the edge-based operator with CVFEM projected nodal gradients (NOC), 3) the edge-based operator with edge-based projected nodal gradients (NOC), 4) the edge-based operator without NOC correction, 5) the CVFEM operator with shifted integration points to the edge, and, lastly, 6) a mixed edge/element scheme in which the orthogonal diffuion operator is edge-based while the NOC terms are based on the elemental CVFEM gradient (either evaluated at the given integration point or integrated over the element for a piecewise constant form).

The last operator is interesting in that it represents a candidate operator for the CVFEM pressure Poisson system when high aspect ratio meshes are used. Figure Fig. 4.6 outlines the convergence of the five above operators; shown are all of the standard norms (∞, 1 and 2) for the R0, R1 and R2 mesh refinements. The results in the left side of the figure indicate that the edge-based scheme with NOC retains second-order convergence for all norms when the more accurate CVFEM projected nodal gradient is used (lumped only tested given its good results). Convergence is degraded with the edge-based scheme when NOC terms are either neglected or use the reduced edge-based projected nodal gradient. The CVFEM-based methods are second order accurate in the L_1 and L_2 norms, however, questionable results are noted in the L_∞ norm for all methods that include any shape function derivative for local or elemental piecwise constant gradient operators. Shifting the Gauss points from the standard subcontrol surface to the edges of the element (while still using shape function derivatives) is only problematic in the L_∞ norm (just as the standard CVFEM approach). The use of the mixed-approach with a piecewise constant gradient over the element demonstrates the same behavior as when using the integration point CVFEM gradient operator. Figure Fig. 4.7 outlines two more refinement meshes for the CVFEM operator (R3 and R4). Results indicate that the L_∞ norm is approaching second order accuracy.

An inspection of the magnitude of error between the exact and computed temperature for the R3 mesh is shown in Figure Fig. 4.8. Results show that the CVFEM error is highest at the corner mesh nodes that form a reduced stencil. The edge-based scheme shows increased error at the higher aspect ratio dual mesh.

3D Steady Laplace Operator with Nonconformal Interface

A three dimensional element-based verification study is provided to evaluate the DG-based CVFEM approach.

$$T = \frac{\lambda}{4}(cos(2a\pi x) + cos(2a\pi y) + cos(2a\pi z)). \tag{4.13}$$

Figure Fig. 4.9 represents the MMS field for temperature. The simulation study includes uniform refinement of a first- and second-order CVFEM basis. Both temperature field and projected nodal gradient norms are of interest.

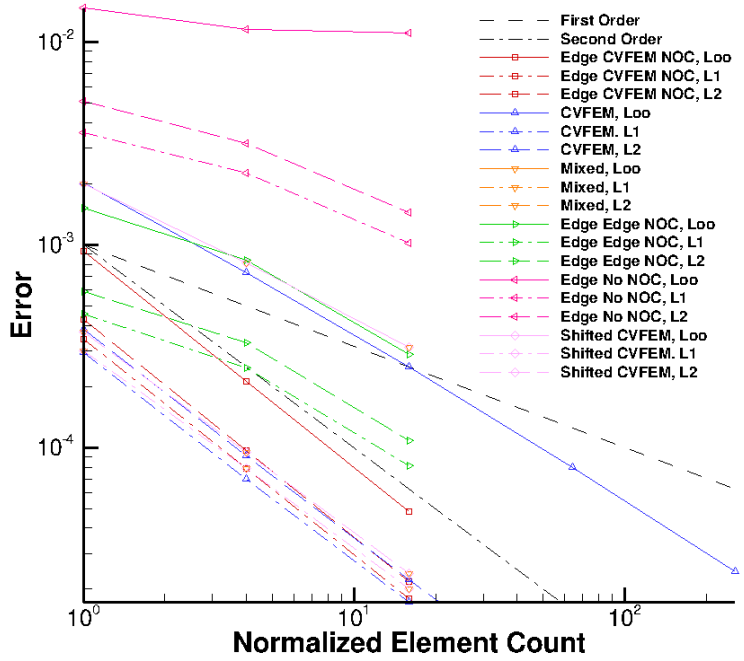

Fig. 4.6: Error norms for tquad4 refinement study. R0, R1, and R2 refinement.

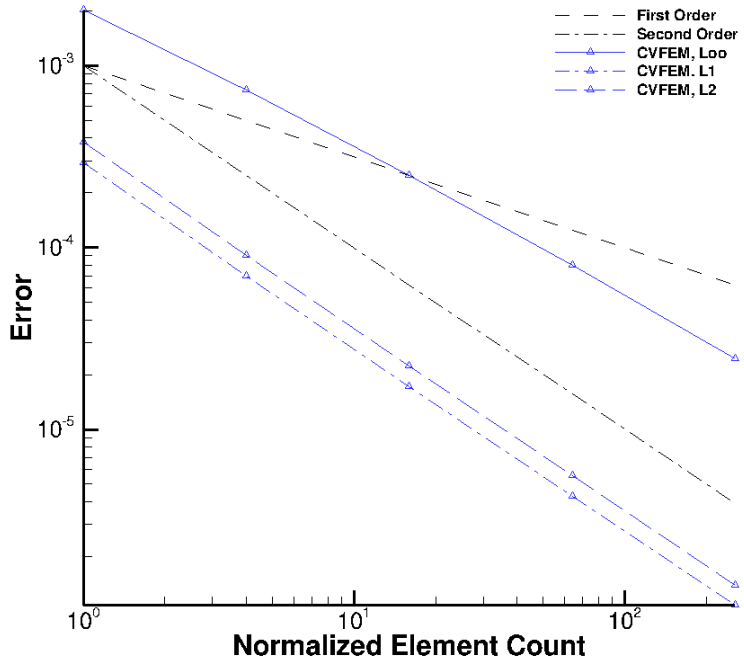

Fig. 4.7: Error norms for tquad4 refinement study. R0, R1, R2, R3, R4, and R4 refinementError for CVFEM.

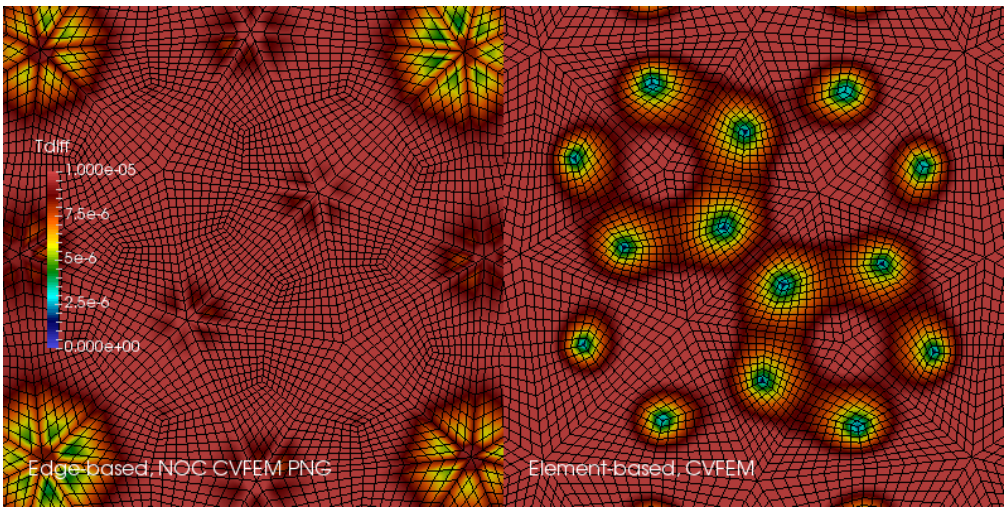

Fig. 4.8: Magnitude of the L_∞ temperature norm comparing the edge-based CVFEM (NOC) and standard CVFEM operators on the R3 mesh.

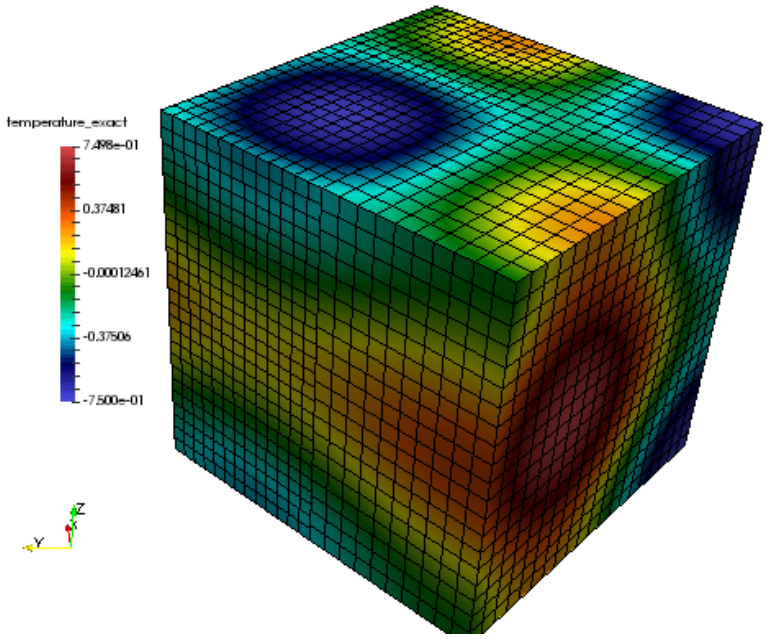

Fig. 4.9: MMS temperature field for nonconformal algorithm.

Figure Fig. 4.10 outlines the linear and quadratic basis. For P1, the CVFEM temperature field predicts between second and first order while for P2, third order is recovered. When using a consistent mass matrix for the projected nodal gradient, second order is noted, see Figure Fig. 4.11.

dof	L_∞	L1	L2
temperature	3.33067e-16	2.30077e-17	4.68103e-17
dTdx	4.13225e-13	9.06848e-15	1.98249e-14
dTdy	4.15668e-13	1.11256e-14	2.15065e-14
dTdz	4.31211e-13	9.60785e-15	1.97517e-14

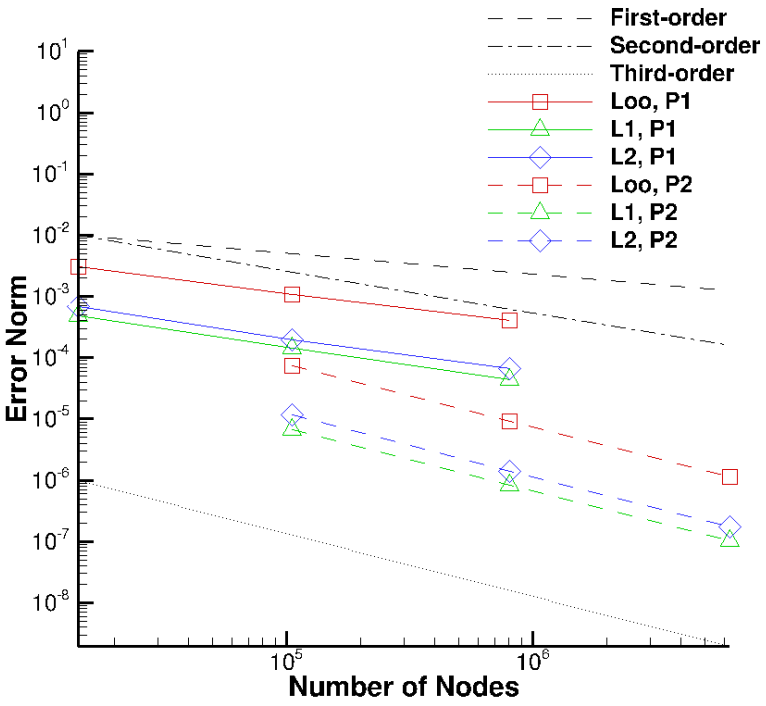

Fig. 4.10: MMS order of accuracy for nonconformal algorithm. Temperature norms for P1 and P2 elements.

Given the order of accuracy results for the P1 implementation, a linear patch test was run. The temperature solution was simply, $T(x, y, z) = x + y + z$; all analytical temperature gradients are unity. Table Table 4.7 demonstrates the successful patch test results for a P1 CVFEM implementation.

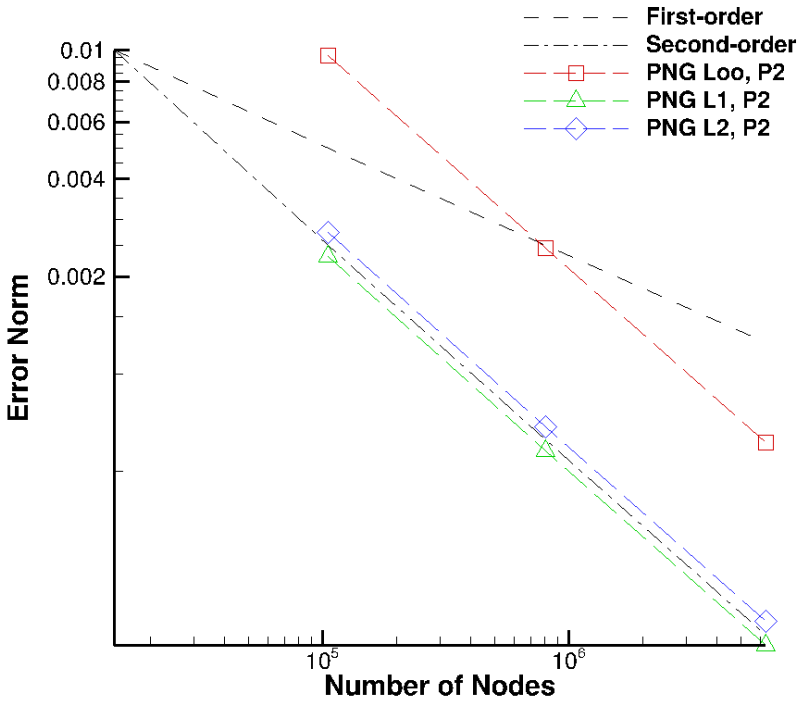

Fig. 4.11: MMS order of accuracy for nonconformal algorithm. Projected nodal gradient norms for P1 and P2 elements.

Bibliography

[Aft94] M. Aftosmis. Upwind method for simulation of viscous flow on adaptively refined meshes. *AIAA Journal*, 32(2):268–277, 1994.

[Dav97] L. Davidson. Large-eddy simulations: a note on the derivation of the equations for the subgrid turbulent kintic energies. Technical Report, Chalmers University of Technology, Department of Thermo and Fluid Dynamics, 1997.

[Dom06] S. Domino. Towards verification of formal time accuracy for a family of approximate projection methods using the method of manufactured solutions. In *Center for Turbulence Research Summer Proceedings*. 2006.

[Dom08] S. Domino. A comparison of various equal-order interpolation methodologies using the method of manufactured solutions. In *Center for Turbulence Research Summer Proceedings*. 2008.

[Dom10] S. Domino. Towards verification of sliding mesh algorithms for complex applications using mms. In *Center for Turbulence Research Summer Proceedings*. 2010.

[Dom14] S. Domino. A comparison between low order and higher order low mach discretization approaches. In *Center for Turbulence Research Summer Proceedings*. 2014.

[DNP98] F. Ducors, F. Nicoud, and T. Poinsot. Wall-adapting local eddy-viscosity models for simulations in complex geometries. In *International Conference on Computational Conference*, volume 50. 1998.

[Dye74] A. J. Dyer. A review of flux-profile relationships. *Boundary-Layer Meteorology*, 7:363–372, 1974.

[EWS+10] H. Edwards, A. Williams, G. Sjaardema, D. Baur, and W. Cochran. Sierra toolkit computational mesh computational model. Technical Report SAND-20101192, Sandia National Laboratories, Albuquerque, NM, 2010.

[HBH+03] M. Heroux, R. Bartlett, V. Howle, R. Hoekstra, J. Hu, T. Kolda, R. Lehoucq, K. Long, R. Pawlowski, E. Phipps, A. Salinger, J. Thornquist, R. Tuminaro, J. Willenbring, and A. Williams. An overview of trilinos. Technical Report SAND-20032927, Sandia National Laboratories, Albuquerque, NM, 2003.

[Jas96] H. Jasek. Error analysis and estimation for the finite volume menthod with applications to fluid flow. In *Ph.D. Thesis, Imperial College*. 1996.

[KV93] Y. Kallinderis and P. Vijayan. Adaptive refinement-coarsening scheme for three-dimensional unstructured meshes. *AIAA Journal*, 31(8):1440–1447, 1993.

[KB89] Y. G. Kallinderis and J. R. Baron. Adaptive methods for a new navier-stokes algorithm. *AIAA Journal*, 27(1):37–43, 1989.

[Mar05] M. Martinez. Comparison of galerkin and control volume finite element for advection-diffusion problems. *Int. J. Num. Meth. Fluids*, 50(3):347–376, 2005.

[Mav00] D. J. Mavriplis. Adaptive meshing techniques for viscous flow calculations on mixed element unstructured meshes. *International Journal for Numerical Methods in Fluids*, 34(2):93–111, 2000.

[MKL03] F. R. Menter, M. Kuntz, and R. Langtry. Ten years of industrial experience with the sst turbulence model. *Turb, Heat and Mass Trans*, 2003.

[Moe84] C.-H. Moeng. A large-eddy-simulation model for the study of planetary boundary-layer turbulence. *J. Atmos. Sci.*, 41(13):2052–2062, 1984.

[Pao82] S. Paolucci. On the filtering of sound waves from the navier-stokes equations. Technical Report SAND-828257, Sandia National Laboratories, Livermore, CA, December 1982.

[RB78] R. G. Rehm and H. R. Baum. The equations of motion for thermally driven buoyant flows. *Journal of Research of the National Bureau of Standards*, 83:279, 1978.

[RM84] R. Rogallo and P. Moin. Numerical simulation of turbulent flows. *Annual Review of Fluid Mechanics*, 16:99–137, 1984.

[SR87] G. Schneider and M. Raw. Control volume finite element method for heat transfer and fluid flow using colocated variables - 1. computational procedure. *Numerial Heat Transfer*, 11(4):363–390, 1987.

[SHZ91] F. Shakib, T. J. R. Hughes, and J. Zdenek. A new finite element formulation for computational fluids dynamics: the compressible euler and navier stokes equations. *Comp. Meth. in App. Mech and Engr.*, 89:141–219, 1991.

[SrensenS02] Jens Nørkær Sørensen and Wen Zhong Shen. Numerical modeling of wind turbine wakes. *Journal of Fluids Engineering*, 124(2):393–399, 05 2002. URL: http://dx.doi.org/10.1115/1.1471361.

[TDB05] S. Tieszen, S. Domino, and A. Black. Validation of a simple turbulence model suitable for closure of temporally-filtered navier-stokes equations using a helium plume. Technical Report SAND-20053210, Sandia National Laboratories, Albuquerque, NM, June 2005.

Index

Symbols

A

B

D

E

I

L

M

N

O

P

R

S

T

U