NaluWindUtils Documentation

Release v0.1.0

Shreyas Ananthan, Marc Henry de Frahan

Sep 17, 2019

CONTENTS

Nalu wind-utils is a companion software library to Nalu-Wind — a generalized, unstructured, massively parallel, low-Mach flow solver for wind energy applications. As the name indicates, this software repository provides various meshing, pre- and post-processing utilities for use with the Nalu CFD code to aid setup and analysis of wind energy LES problems. This software is licensed under Apache License Version 2.0 open-source license.

The source code is hosted and all development is coordinated through the Github repository under the Exawind organization umbrella. The official documentation for all released and development versions are hosted on ReadTheDocs. Users are welcome to submit issues, bugs, or questions via the issues page. Users are also encouraged to contribute to the source code and documentation using pull requests using the normal Github fork and pull request workflow.

This documentation is divided into two parts:

User Manual

> Directed towards end-users, this part provides detailed documentation regarding installation and usage of the various utilities available within this library. Here you will find a comprehensive listing of all available utilties, and information regarding their usage and current limitations that the users must be aware of.

Developer Manual

> The developer guide is targeted towards users wishing to extend the functionality provided within this library. Here you will find details regarding the code structure, API supported by various classes, and links to source code documentation extracted using Doxygen.

Acknowledgements

This software is developed by researchers at NREL and Sandia National Laboratories with funding from DOE's Exascale Computing Project and DOE WETO Atmosphere to electrons (A2e) research initiative.

Part I

User Manual

CHAPTER

ONE

INTRODUCTION

This section provides a general overview of NaluWindUtils and describes features common to all utilities available within this package.

1.1 Installing NaluWindUtils

NaluWindUtils is written using C++ and Fortran and depends on several packages for compilation. Every effort is made to keep the list of third party libraries (TPLs) similar to the Nalu dependencies. Therefore, users who have successfully built Nalu on their systems should be able to build NaluWindUtils without any additional software. The main dependencies are listed below:

1. Operating system — NaluWindUtils has been tested on Linux and Mac OS X operating systems.
2. `C++` compiler — Like Nalu, this software package requires a recent version of the `C++` compiler that supports the C++11 standard. The build system has been tested with GNU GCC, LLVM/Clang, and Intel suite of compilers.
3. Trilinos Project — Particularly the Sierra ToolKit (STK) and Seacas packages for interacting with Exodus-II mesh and solution database formats used by Nalu.
4. YAML C++ – YAML C++ parsing library to process input files.

Users are strongly encouraged to use the Spack package manager to fetch and install Trilinos along with all its dependencies. Spack greatly simplifies the process of fetching, configuring, and installing packages without the frustrating guesswork. Users unfamiliar with Spack are referred to the installation section in the official Nalu documentation that describes the steps necessary to install Trilinos using Spack. Users unable to use Spack for whatever reason are referred to Nalu manual that details steps necessary to install all the necessary dependencies for Nalu without using Spack.

While not a direct build dependency for NaluWindUtils, the users might want to have Paraview or VisIt installed to visualize the outputs generated by this package.

1.1.1 Compiling from Source

1. If you are on an HPC system that provides Modules Environment, load the necessary compiler modules as well as any other package modules that are necessary for Trilinos.
2. Clone the latest release of NaluWindUtils from the git repository.

```
cd ${HOME}/nalu/
git clone https://github.com/NaluCFD/NaluWindUtils.git
cd NaluWindUtils
```

(continues on next page)

(continued from previous page)

```
# Create a build directory
mkdir build
cd build
```

3. Run CMake configure. The `examples` directory provides two sample configuration scripts for spack and non-spack builds. Copy the appropriate script into the `build` directory and edit as necessary for your particular system. In particular, the users will want to update the paths to the various software libraries that CMake will search for during the configuration process. Please see *CMake Configuration Options* for information regarding the different options available.

   The code snippet below shows the steps with the Spack configuration script, replace the file name `doConfigSpack.sh` with `doconfig.sh` for a non-spack environment.

```
# Ensure that `build` is the working directory
cp ../examples/doConfigSpack.sh .
# Edit the script with the correct paths, versions, etc.

# Run CMake configure
./doConfigSpack.sh -DCMAKE_INSTALL_PREFIX=${HOME}/nalu/install/
```

4. Run *make* to build and install the executables.

```
make         # Use -j N if you want to build in parallel
make install # Install the software to a common location
```

5. Test installation

```
bash$ ${HOME}/nalu/install/bin/nalu_preprocess -h
Nalu preprocessor utility. Valid options are:
  -h [ --help ]                         Show this help message
  -i [ --input-file ] arg (=nalu_preprocess.yaml)
                                        Input file with preprocessor options
```

If you see the *help message* as shown above, then proceed to *General Usage* section to learn how to use the compiled executables. If you see errors during either the CMake or the build phase, please capture *verbose* outputs from both steps and submit an issue on Github.

Note:

1. The WRF to Nalu inflow conversion utility is not built by default. Users must explicitly enable compilation of this utility using the *ENABLE_WRFTONALU* flag. The default behavior is chose to eliminate the extra depenency on NetCDF-Fortran package required build this utility. The `examples/doConfigSpack.sh` provides an example of how to build the this utility if desired.
2. See *Building Documentation* for instructions on building a local copy of this user manual as well as API documentation generated using Doxygen.
3. Run `make help` to see all available targets that CMake understands to quickly build only the executable you are looking for.

1.1.2 Building Documentation

Official documentation is available online on ReadTheDocs site. However, users can generate their own copy of the documentation using the RestructuredText files available within the `docs` directory. NaluWindUtils uses the Sphinx

documentation generation package to generate HTML or PDF files from the `rst` files. Therefore, the documentation building process will require Python and Sphinx packages to be installed on your system.

The easiest way to get Sphinx and all its dependencies is to install the Anaconda Python Distribution for the operating system of your choice. Expert users can use Miniconda to install basic packages and install additional packages like Sphinx manually within a *conda environment*.

Doc Generation Using CMake

1. Enable documentation genration via CMake by turning on the *`ENABLE_SPHINX_DOCS`* flag.
2. Run `make docs` to generate the generate the documentation in HTML form.
3. Run `make sphinx-pdf` to generate the documentation using `latexpdf`. Note: requires Latex packages installed in your system.

The resulting documentation will be available in `doc/manual/html` and `doc/manual/latex` directories respectively for HTML and PDF builds within the CMake build directory. See also *Building API Documentation*.

Doc Generation Without CMake

Since CMake will require users to have Trilinos installed, an alternate path is provided to bypass CMake and generate documentation using `Makefile` on Linux/OS X systems and `make.bat` file on Windows systems provided in the `docs/manual` directory.

```
cd docs/manual
# To generate HTML documentation
make html
open build/html/index.html

# To generate PDF documentation
make latexpdf
open build/latex/NaluWindUtils.pdf

# To generate help message
make help
```

Note: Users can also use `pipenv` or `virtualenv` as documented here to manage their python packages without Anaconda.

1.1.3 CMake Configuration Options

Users can use the following variables to control the CMake behavior during configuration phase. These variables can be added directly to the configuration script or passed as arguments to the script via command line as shown in the previous section.

`CMAKE_INSTALL_PREFIX`
: The directory where the compiled executables and libraries as well as headers are installed. For example, passing `-DCMAKE_INSTALL_PREFIX=${HOME}/software` will install the executables in `${HOME}/software/bin` when the user executes the `make install` command.

`CMAKE_BUILD_TYPE`
: Controls the optimization levels for compilation. This variable can take the following values:

Value	Typical flags
RELEASE	`-O2 -DNDEBUG`
DEBUG	`-g`
RelWithDebInfo	`-O2 -g`

Example: `-DCMAKE_BUILD_TYPE:STRING=RELEASE`

Trilinos_DIR
Absolute path to the directory where Trilinos is installed.

YAML_ROOT
Absolute path to the directory where the YAML C++ library is installed.

ENABLE_WRFTONALU
A boolean flag indicating whether the WRF to Nalu conversion utility is to be built along with the C++ utilities. By default, this utility is not built as it requires the NetCDF-Fortran library support that is not part of the standard Nalu build dependency. Users wishing to enable this library must make sure that the NetCDF-Fortran library has been installed and configure the *NETCDF_F77_ROOT* and *NETCDF_DIR* appropriately.

NETCDF_F77_ROOT
Absolute path to the location of the NETCDF Fortran 77 library.

NETCDF_DIR
Absolute path to the location of the NETCDF C library.

ENABLE_SPHINX_DOCS
Boolean flag to enable building Sphinx-based documentation via CMake. Default: OFF.

ENABLE_DOXYGEN_DOCS
Boolean flag to enable extract source code documentation using Doxygen. Default: OFF.

ENABLE_SPHINX_API_DOCS
Enable embedding API documentation generated from Doxygen within user and developer manuals. Default: OFF.

Further fine-grained control of the build environment can be achieved by using standard CMake flags, please see CMake documentation for details regarding these variables.

CMAKE_VERBOSE_MAKEFILE
A boolean flag indicating whether the build process should output verbose commands when compiling the files. By default, this flag is `OFF` and `make` only shows the file being processed. Turn this flag `ON` if you want to see the exact command issued when compiling the source code. Alternately, users can also invoke this flag during the `make` invocation as shown below:

```
bash$ make VERBOSE=1
```

CMAKE_CXX_COMPILER
Set the C++ compiler used for compiling the code

CMAKE_C_COMPILER
Set the C compiler used for compiling the code

CMAKE_Fortran_COMPILER
Set the Fortran compiler used for compiling the code

CMAKE_CXX_FLAGS
Additional flags to be passed to the C++ compiler during compilation. For example, to enable OpenMP support during compilation pass `-DCMAKE_CXX_FLAGS=" -fopenmp"` when using the GNU GCC compiler.

CMAKE_C_FLAGS
Additional flags to be passed to the C compiler during compilation.

CMAKE_Fortran_FLAGS
Additional flags to be passed to the Fortran compiler during compilation.

1.2 General Usage

Most utilities require a YAML input file containing all the information necessary to run the utility. The executables have been configured to look for a default input file name within the run directory, this default filename can be overridden by providing a custom filename using the `-i` option flag. Users can use the `-h` or the `--help` flag with any executable to look at various command line options available as well as the name of the default input file as shown in the following example:

```
bash$ src/preprocessing/nalu_preprocess -h
Nalu preprocessor utility. Valid options are:
  -h [ --help ]                         Show this help message
  -i [ --input-file ] arg (=nalu_preprocess.yaml)
                                        Input file with preprocessor options
```

The output above shows the default input file name as `nalu_preprocess.yaml` for the **nalu_preprocess** utility.

Note: It is assumed that the `bin` directory where the utilities were installed are accessible via the user's `PATH` variable. Please refer to *Installing NaluWindUtils* for more details.

CHAPTER

TWO

TUTORIALS

2.1 Pre-processing for ABL precursor runs

This tutorial walks through the steps required to create an ABL mesh and initialize the fields for an ABL precursor run. In this tutorial, you will use the *abl_mesh* and certain capabilities of *nalu_preprocess*. The steps covered in this tutorial are

1. Generate a $1 \times 1 \times 1$ km HEX block mesh with uniform resolution of 10m in all three directions.
2. Generate a *sampling plane* at hub-height (90m) where the velocity field will be sampled to force it to a desired wind speed and direction using a driving pressure gradient source term.
3. Initialize the velocity and temperature field to the desired profile as a function of height, and add perturbations to the fields to kick-off turbulence generation.

2.1.1 Prerequisites

Before attempting this tutorial, you should have a compiled version of NaluWindUtils. Please consult the *Installing NaluWindUtils* section to fetch, configure, and compile the latest version of the source code. You can also download the input file (`abl_setup.yaml`) that will be used with **`abl_mesh`** and **`nalu_preprocess`** executables.

2.1.2 Generate ABL precursor mesh

In this step, we will use the **`abl_mesh`** utility to generate $1 \times 1 \times 1$ km with a uniform resolution of 10m in all three directions. The domain will span $[0, 1000]$ m in each direction. The relevant section in the input file is shown below

```
#
# 1. Generate ABL mesh
#
nalu_abl_mesh:
  output_db: abl_1x1x1_10_mesh.exo    # output filename

  spec_type: bounding_box             # Vertex input type

  vertices:
    - [    0.0,    0.0,    0.0 ]     # min corner
    - [ 1000.0, 1000.0, 1000.0 ]     # max corner

  mesh_dimensions: [ 100, 100, 100]  # number of elements in each direction


```

With this section saved in the input file `abl_setup.yaml`, the sample interaction is shown below

```
$ abl_mesh -i abl_setup.yaml

Nalu ABL Mesh Generation Utility
Input file: abl_setup.yaml
HexBlockBase: Registering parts to meta data
      Mesh block: fluid
Num. nodes = 1030301; Num elements = 1000000
      Generating node IDs...
      Creating nodes... 10% 20% 30% 40% 50% 60% 70% 80% 90%
      Generating element IDs...
      Creating elements... 10% 20% 30% 40% 50% 60% 70% 80% 90%
      Finalizing bulk modifications...
      Generating X Sideset: west
      Generating X Sideset: east
      Generating Y Sideset: south
      Generating Y Sideset: north
      Generating Z Sideset: terrain
      Generating Z Sideset: top
      Generating coordinates...
       Generating x spacing: constant_spacing
       Generating y spacing: constant_spacing
       Generating z spacing: constant_spacing
Writing mesh to file: abl_1x1x1_10_mesh.exo

Memory usage: Avg:  553.148 MB; Min:  553.148 MB; Max:  553.148 MB
```

2.1.3 Initializing fields and sampling planes

In the next step we will use **`nalu_preprocess`** to setup the fields necessary for a precursor simulation. The relevant section of the input file is shown below

```
#
# 2. Preprocessing
#
nalu_preprocess:
  input_db: abl_1x1x1_10_mesh.exo
  output_db: abl_1x1x1_10.exo

  tasks:
    - init_abl_fields

  init_abl_fields:
    fluid_parts: [ fluid ]

    velocity:
      heights: [ 0.0, 1000.0]
      values:
        - [7.250462296293199, 3.380946093925596, 0.0]
        - [7.250462296293199, 3.380946093925596, 0.0]
      perturbations:
        reference_height: 50.0
        amplitude: [1.0, 1.0]
        periods: [4.0, 4.0]

```

(continues on next page)

(continued from previous page)

```
24      temperature:
25        heights: [    0, 650.0, 750.0, 1000.0 ]
26        values:  [300.0, 300.0, 308.0,  308.75]
27        perturbations:
28          amplitude: 0.8
29          cutoff_height: 600.0
30          skip_periodic_parts: [ west, east, north, south]
```

The following actions are performed

1. Lines 14–18: Initialize a constant velocity field such that the wind speed is 8.0 m/s along 245° compass direction.
2. Lines 24–26: A constant temperature field of 300K till 650m and then a capping inversion between 650m to 750m and a temperature gradient of 0.003 K/m above the capping inversion zone.
3. Pertubations to the velocity (lines 19–22) and temperature field (lines 27–30) to kick off turbulence generation during the precursor run. The velocity field perturbations are similar to those generated in SOWFA for ABL precursor runs.
4. The mesh generated in the previous step is used as input (line 5), and a new file is written out with the new fields and the sampling plane (line 6).

Output from the execution of **`nalu_preprocess`** with this input file is shown below

```
$ nalu_preprocess -i abl_setup.yaml

Nalu Preprocessing Utility
Input file: abl_setup.yaml
Found 1 tasks
    - init_abl_fields

Performing metadata updates...
Metadata update completed
Reading mesh bulk data... done.

-------------------------------------------------
Begin task: init_abl_fields
Generating ABL fields
End task: init_abl_fields

All tasks completed; writing mesh...
Exodus results file: abl_1x1x1_10.exo

Memory usage: Avg:  786.082 MB; Min:  786.082 MB; Max:  786.082 MB
```

2.1.4 Using `ncdump` to examine mesh metadata

`ncdump` is a NetCDF utility that is built and installed as a depedency of Trilinos. Since Trilinos is a dependency of NaluWindUtils, you should have **`ncdump`** available in your path if Trilinos and its dependencies were loaded properly (either via `spack` or `module load`). **`ncdump`** is useful to quickly examine the Exodus file metadata from the command line. Invoke the command with `-h` option to quickly see the number of nodes and elements in a mesh

```
$ ncdump -h abl_1x1x1_10.exo
netcdf abl_1x1x1_10 {
dimensions:
    len_string = 33 ;
```

(continues on next page)

(continued from previous page)

```
    len_line = 81 ;
    four = 4 ;
    num_qa_rec = 1 ;
    num_info = 2 ;
    len_name = 33 ;
    num_dim = 3 ;
    time_step = UNLIMITED ; // (1 currently)
    num_nodes = 1040502 ;
    num_elem = 1000000 ;
    num_el_blk = 1 ;
    num_node_sets = 1 ;
    num_side_sets = 6 ;
    num_el_in_blk1 = 1000000 ;
    num_nod_per_el1 = 8 ;
    num_nod_ns1 = 10201 ;
    num_side_ss1 = 10000 ;
    num_df_ss1 = 40000 ;
    num_side_ss2 = 10000 ;
    num_df_ss2 = 40000 ;
    num_side_ss3 = 10000 ;
    num_df_ss3 = 40000 ;
    num_side_ss4 = 10000 ;
    num_df_ss4 = 40000 ;
    num_side_ss5 = 10000 ;
    num_df_ss5 = 40000 ;
    num_side_ss6 = 10000 ;
    num_df_ss6 = 40000 ;
    num_nod_var = 4 ;
```

For the ABL precursor mesh generated using **`abl_mesh`** we have 1 mesh block (`num_el_blk`) that has one million elements (`num_el_in_blk1`) composed of Hexahedral elements with 8 nodes per element (`num_nod_per_el1`). There are 4 nodal field variables (`num_nod_var`) stored in this database that were created by **`nalu_preprocess`** utility. Finally, there are 6 sidesets (`num_side_sets`) each with 10,000 faces, and one node set (`num_node_sets`) that contains 10201 nodes that were created as a sampling plane at hub height of 90m during the pre-processing step.

Use the `-v` flag with the desired variable names (separated by commas) to examine the contents of those variables. For example, to output the mesh blocks (`eb_names`), sidesets or boundaries (`ss_names`), nodal (`name_nod_var`) and element fields (`name_elem_var`) present in an Exodus database:

```
$ ncdump -v eb_names,ss_names,name_nod_var abl_1x1x1_10.exo
#
# OUTPUT TRUNCATED !!!
#
data:

 eb_names =
  "fluid" ;

 ss_names =
  "west",
  "east",
  "south",
  "north",
  "terrain",
  "top" ;
```

(continues on next page)

(continued from previous page)

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

As seen in the output for `name_nod_var`, Exodus file contains `temperature`, a scalar field, and `velocity`, a vector field. Internally, exodus stores each component of a vector or tensor field as a separate variable. The mesh block is called `fluid` can should be referred as such in the pre-processing tasks or within the Nalu input file. As indicated in the `dimensions`, this file contains one `time_step`, you can use `-v time_whole` to determine the timesteps that are currently stored in the Exodus database.

2.2 Wind-farm mesh refinement for Actuator Line simulation using Percept

This tutorial demonstrates the workflow for refining ABL meshes for use with actuator line simulations using the Percept mesh adaptivity tool. We will start with the precursor mesh and add nested zones of refinement around turbines of interest so that the wakes are captured with adequate resolution necessary to predict the impact on downstream turbine performance. We will perform the following steps

1. Use **`nalu_preprocess`** to *tag* elements within the mesh that must be refined. In this exercise, we will perform two levels of refinement where the second level is nested within the first refinement zone. This step creates a `turbine_refinement_field`, an element field, in the Exodus database. The refinement field is a scalar with a value ranging between 0 and 1. We will use this field as a threshold to control the regions where the refinement is applied by the **`mesh_adapt`** utility in Percept.
2. Invoke Percept's **`mesh_adapt`** utility twice to perform two levels of refinement. Each invocation will use the `turbine_refinement_field`, created in the previous step, to determine the region where refinement is applied, the threshold is changed using YAML-formatted input files to **`mesh_adapt`** during each call.

2.2.1 Prerequisites

To complete this tutorial you will need the Exodus mesh (`abl_1x1x1_10_mesh.exo`) generated in the *the previous tutorial*. You will also need the input file for **`nalu_preprocess`** (`abl_refine.yaml`)

2.2.2 Tag mesh regions for refinement

In this step we will use **`nalu_preprocess`** to create a refinement field that will be used by **`mesh_adapt`** to determine which elements are selected for refinement. The input file that performs this action is shown below

```
mesh_local_refinement:
  fluid_parts: [ fluid ]
  write_percept_files: true
  percept_file_prefix: adapt
  search_tolerance: 11.0

  turbine_diameters:  80.0
  turbine_heights: 70.0
  turbine_locations:
    - [ 550.0, 350.0, 0.0 ]
```

(continues on next page)

(continued from previous page)

```
11        - [ 400.0, 500.0, 0.0 ]
12      orientation:
13        type: wind_direction
14        wind_direction: 245.0
15      refinement_levels:
16        - [ 4.0, 4.0, 2.0, 2.0 ]
17        - [ 3.0, 3.0, 1.2, 1.2 ]
```

The mesh blocks targeted for refinement is provided as a list to the `fluid_parts` parameter (line 2), `turbine_locations` list the base locations of the turbines in the wind farm that are being simulated, `refinement_levels` contain a list of length equal to the number of nested refinement levels. Each entry in this list contains an array of four non-dimensional lengths: the upstream, downstream, lateral, and vertical extent of the refinement zones (as a multiple of rotor diameters) with respect to the rotation center of the turbine. The orientation of the refinement boxes is determined by the parameters provided within the `orientation` sub-dictionary. In the current example, the boxes will be oriented along the wind direction (245°) to match the ABL wind direction at hub-height used in the previous tutorial.

Note: It is recommended that the `search_tolerance` parameter in `mesh_local_refinement` section be set slightly larger than the coarset mesh resolution in the base ABL mesh chosen for refinement. This prevents jagged boundaries around the refinement zones as a result of roundoff and truncation errors. In our current example, this parameter was set to 11m based on the fact that the base mesh has a uniform resolution of 10m.

The output of **`nalu_preprocess`** is shown below

```
$ nalu_preprocess -i abl_refine.yaml

Nalu Preprocessing Utility
Input file: abl_refine.yaml
Found 1 tasks
    - mesh_local_refinement

Performing metadata updates...
Metadata update completed
Reading mesh bulk data... done.

-------------------------------------------------
Begin task: mesh_local_refinement
Processing percept field: turbine_refinement_field
Writing percept input files...
      adapt1.yaml
      adapt2.yaml
Sample percept command line:
mesh_adapt --refine=DEFAULT --input_mesh=mesh0.e --output_mesh=mesh1.e --RAR_
↪info=adapt1.yaml
End task: mesh_local_refinement

All tasks completed; writing mesh...
Exodus results file: mesh0.e

Memory usage: Avg:  723.312 MB; Min:  723.312 MB; Max:  723.312 MB
```

2.2.3 Refine using Percept

After executing **nalu_preprocess** we should have `mesh0.e`, the Exodus database used as input for **mesh_adapt** and two YAML files `adapt1.yaml` and `adapt2.yaml` that contain the thresholds for each level of refinement. To invoke Percept in serial mode, execute the following command

```
# Refine the first level
mesh_adapt --refine=DEFAULT --input_mesh=mesh0.e --output_mesh=mesh1.e --RAR_
↪info=adapt1.yaml --progress_meter=1
# Refine the second level
mesh_adapt --refine=DEFAULT --input_mesh=mesh1.e --output_mesh=mesh2.e --RAR_
↪info=adapt2.yaml --progress_meter=1
```

After successful execution of the two invocations of **mesh_adapt**, the refined mesh for use with actuator line wind farm simulations is saved in `mesh2.e`. Percept-based refinement creates pyramid and tetrahedral elements at the refinement interfaces. These additional elements are added to new mesh blocks (parts in STK parlance) that must be included in the Nalu input file for simulation. Use **ncdump** (see *previous tutorial*) to examine the names of the new mesh blocks created by Percept.

```
$ ncdump -v eb_names mesh2.e
#
# OUTPUT TRUNCATED !!!
#
data:

 eb_names =
  "fluid",
  "fluid.pyramid_5._urpconv",
  "fluid.tetrahedron_4._urpconv",
  "fluid.pyramid_5._urpconv.Tetrahedron_4._urpconv" ;
```

For large meshes, parallel execution of Percept's **mesh_adapt** utility is recommended. A sample command line is shown below

```
# Example mesh_adapt invocation in parallel.
mpiexec -np ${NPROCS} mesh_adapt \
             --refine=DEFAULT \
             --RAR_info=adapt1.yaml \
             --progress_meter=1 \
             --input_mesh=mesh0.e \
             --output_mesh=mesh1.e \
             --ioss_read_options="auto-decomp:yes" \
             --ioss_write_options="large,auto-join:yes"
```

We pass `auto-join:yes` to IOSS write options so that the final mesh is combined for subsequent use with a different number of MPI ranks with Nalu.

2.2.4 Troubleshooting tips

- Percept **mesh_adapt** will hang if it runs out of memory without any error message. The user must ensure that enough memory is available to perform the refinements. Parallel execution on a larger number of nodes is the best solution to this problem.
- Percept creates long part names for the new mesh blocks it generates. These names are sometimes longer than the 32 characters allowed by SEACAS utilities for Exodus strings. Exodus mesh reading process will

automatically truncate these names during read, but STK will throw an error if the full name is used to refer to the part. The user must take care to truncate the names to 32 characters in the Nalu input file.

- Percept declares additional parts of form `<BASE_PART>.pyramid_5._urpconv.Tetrahedron_4._urpconv` in anticipation of possible refinement of pyramid elements into pyramids and tetrahedrons. However, the nested refinement strategy does not result in pyramids being refined and, therefore, this part remains empty. Currently, SEACAS and STK will throw an error if the user attempts to include this part in the Nalu input file during simulations.
- When using **`mesh_adapt`** in parallel, appropriate IOSS read/write options must be specified to allow automatic decomposition of an undecomposed mesh and subsequent rejoin after parallel exection. Failure to provide appropriate options will lead to error during execution of **`mesh_adapt`**.

CHAPTER

THREE

NALU_PREPROCESS – NALU PREPROCESSING UTILITIES

This utility loads an input mesh and performs various pre-processing *tasks* so that the resulting output database can be used in a wind LES simulation. Currently, the following *tasks* have been implemented within this utility.

Task type	Description
`init_abl_fields`	Initialize ABL velocity and temperature fields
`init_channel_fields`	Initialize channel velocity fields
`create_bdy_io_mesh`	Create an I/O transfer mesh for sampling inflow planes
`mesh_local_refinement`	Local refinement around turbines for wind farm simulations
`rotate_mesh`	Rotate mesh
`move_mesh`	Translate mesh by a given offset vector

Warning: Not all tasks are capable of running in parallel. Please consult documentation of individual tasks to determine if it is safe to run it in parallel using MPI. It might be necessary to set *`automatic_decomposition_type`* when running in parallel.

The input file (`download`) must contain a **nalu_preprocess** section as shown below. Input options for the individual tasks are provided as sub-sections within **nalu_preprocess** with the corresponding task names provided under `tasks`. For example, in the sample shown below, the program will expect to see two sub-sections, namely `init_abl_fields` and `generate_planes` based on the list of tasks shown in lines 22-23.

```
1  # -*- mode: yaml -*-
2  #
3  # Nalu Preprocessing Utility - Example input file
4  #
5
6  # Mandatory section for Nalu preprocessing
7  nalu_preprocess:
8    # Name of the input exodus database
9    input_db: abl_mesh.g
10   # Name of the output exodus database
11   output_db: abl_mesh_precursor.g
12
13   # Flag indicating whether the database contains 8-bit integers
14   ioss_8bit_ints: false
15
16   # Flag indicating mesh decomposition type (for parallel runs)
17   # automatic_decomposition_type: rcb
18
19   # Nalu preprocessor expects a list of tasks to be performed on the mesh and
```

(continues on next page)

(continued from previous page)

```
  # field data structures
  tasks:
    - init_abl_fields
    - generate_planes
```

3.1 Command line invocation

```
mpiexec -np <N> nalu_preprocess -i [YAML_INPUT_FILE]
```

-i, --input-file
Name of the YAML input file to be used. Default: `nalu_preprocess.yaml`.

3.2 Common input file options

input_db
Path to an existing Exodus-II mesh database file, e.g.., `ablNeutralMesh.g`

output_db
Filename where the pre-processed results database is output, e.g., `ablNeutralPrecursor.g`

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

tasks
A list of task names that define the various pre-processing tasks that will be performed on the input mesh database by this utility. The program expects to find additional sections with headings matching the task names that provide additional inputs for individual tasks. By default, the task names found within the list should correspond to one of the **task types** discussed earlier in this section. If the user desires to use custom names, then the exact task type should be provided with a `type` within the task section. A specific use-case where this is useful is when the user desires to rotate the mesh, perform additional operations, and, finally, rotate it back to the original orientation.

```
tasks:
  - rotate_mesh_ccw  # Rotate mesh such that sides align with XYZ axes
  - generate_planes  # Generate sampling planes using bounding box
  - rotate_mesh_cw   # Rotate mesh back to the original orientation

rotate_mesh_ccw:
  task_type: rotate_mesh
  mesh_parts:
    - unspecified-2-hex

```

(continues on next page)

(continued from previous page)

```
    angle: 30.0
    origin: [500.0, 0.0, 0.0]
    axis: [0.0, 0.0, 1.0]

rotate_mesh_cw:
    task_type: rotate_mesh
    mesh_parts:
      - unspecified-2-hex
      - zplane_0080.0          # Rotate auto generated parts also

    angle: -30.0
    origin: [500.0, 0.0, 0.0]
    axis: [0.0, 0.0, 1.0]
```

transfer_fields
: A Boolean flag indicating whether the time histories of the fields available in the input mesh database must be transferred to the output database. Default: `false`.

ioss_8bit_ints
: A Boolean flag indicating whether the output database must be written out with 8-bit integer support. Default: `false`.

3.3 init_abl_fields

This task initializes the vertical velocity and temperature profiles for use with an ABL precursor simulations based on the parameters provided by the user and writes it out to the *output_db*. It is safe to run `init_abl_fields` in parallel. A sample invocation is shown below

```
init_abl_fields:
  fluid_parts: [fluid]

  temperature:
    heights: [    0, 650.0, 750.0, 10750.0]
    values:  [280.0, 280.0, 288.0,   318.0]

    # Optional section to add random perturbations to temperature field
    perturbations:
      amplitude: 0.8 # in Kelvin
      cutoff_height: 600.0 # Perturbations below capping inversion
      skip_periodic_parts: [east, west, north, south]

  velocity:
    heights: [0.0, 10.0, 30.0, 70.0, 100.0, 650.0, 10000.0]
    values:
      - [ 0.0, 0.0, 0.0]
      - [4.81947, -4.81947, 0.0]
      - [5.63845, -5.63845, 0.0]
      - [6.36396, -6.36396, 0.0]
      - [6.69663, -6.69663, 0.0]
      - [8.74957, -8.74957, 0.0]
      - [8.74957, -8.74957, 0.0]

    # Optional section to add sinusoidal streaks to the velocity field
    perturbations:
```

(continues on next page)

(continued from previous page)

```
27        reference_height: 50.0    # Reference height for damping
28        amplitude: [1.0, 1.0]     # Perturbation amplitudes in Ux and Uy
29        periods: [4.0, 4.0]       # Num. periods in x and y directions
```

fluid_parts
: A list of element block names where the velocity and/or temperature fields are to be initialized.

temperature
: A YAML dictionary containing two arrays: `heights` and the corresponding `values` at those heights. The data must be provided in SI units. No conversion is performed within the code.

  The temperature section can contain an optional section `perturbations` (lines 8-12) that will add fluctuations to the temperature field. It requires three parameters: 1. the amplitude of oscillations (in degrees Kelvin), 2. the cutoff height above which perturbations are not added, and a list of sidesets where the perturbations should not be added. It is important that the perturbations are not added to the periodic sidesets, otherwise the Nalu simulations will show spurious flow structures.

velocity
: A YAML dictionary containing two arrays: `heights` and the corresponding `values` at those heights. The data must be provided in SI units. No conversion is performed within the code. The values in this case are two dimensional lists of shape `[nheights, 3]` where `nheights` is the length of the *heights* array provided.

  Like temperature, the user can add sinusoidal streaks to the velocity field to trigger the turbulence generation – see lines 25-29. The implementation follows the method used in SOWFA.

Note: Only one of the entries `velocity` or `temperature` needs to be present. The program will skip initialization of a particular field if it cannot find an entry in the input file. This can be used to speed up the execution process if the user intends to initialize uniform velocity throughout the domain within Nalu.

3.4 mesh_local_refinement

This task creates an *error indicator field* that can be used to locally refine the mesh using Percept. This is used to refine the wind farm simulation mesh around the turbines to capture the wakes with the desired resolution while performing the ABL simulations with a coarser mesh resolution.

Example Percept invocation

```
# Load necessary percept modules ...
mpiexec -np ${NPROCS} mesh_adapt \
             --refine=DEFAULT \
             --RAR_info=adapt1.yaml \
             --progress_meter=1 \
             --input_mesh=mesh0.e \
             --output_mesh=mesh1.e \
             --ioss_read_options="auto-decomp:yes" \
             --ioss_write_options="large,auto-join:yes"
```

Note:

1. This utility just creates a field that will be used by *percept* to perform the refinement. The user must execute percept to actually refine the mesh.

2. The `mesh_adapt` utility from Percept must be called once for each level of refinement desired. Each step will use the input file created by the pre-processing utility. However, the mesh files created by percept during the intermediate levels are temporary files used for the next invocation of percept and can be discarded. Only the final mesh file is used with Nalu for wind farm simulations. In the above example increment `adaptN.yaml` and `meshN.e` for input and output appropriately.
3. Currently, `mesh_adapt` utility requires the meshes to be numbered serially. So it is recommended that the user start with `mesh0.e` and then name the output files `mesh1.e` and so on for each level of refinement.
4. For the final refinement level the `auto-join` option is useful to obtain a single mesh file instead of decomposed files for the number of MPI ranks Percept was invoked on. If you leave out the `auto-join` option for intermediate levels, make sure you don't provide `auto-decomp` option for the next level of refinement.
5. Percept uses a lot of memory, so make sure that `mesh_adapt` is invoked in parallel over a large number of MPI ranks, preferably under subscribing cores on a node.
6. Always use `progress_meter` to see if the job is progressing as expected. `mesh_adapt` can hang without warning if it runs out of memory.
7. Mesh refinement process will create new blocks especially containing tets and pyramids. Make sure these are added to the Nalu-Wind input file. Use `ncdump -v eb_names` to see the new parts that were created by the refinement process.

```
mesh_local_refinement:
  fluid_parts: [fluid]
  write_percept_files: true
  percept_file_prefix: adapt
  search_tolerance: 11.0

  turbine_locations:
    - [ 200.0, 200.0, 0.0 ]
    - [ 230.0, 300.0, 0.0 ]

  turbine_diameters: 15.0         # Provide a list for variable diameters
  turbine_heights: 50.0           # Provide a list for variable tower heights
  orientation:
    type: wind_direction
    wind_direction: 225.0
  refinement_levels:
    - [ 7.0, 12.0, 7.0, 7.0 ]
    - [ 5.0, 10.0, 5.0, 5.0 ]
    - [ 3.0, 6.0, 3.0, 3.0 ]
    - [ 1.5, 3.0, 1.2, 1.2 ]
```

turbine_diameters
: A list of turbine diameters for the turbines in the wind farm. If all the turbines in the wind farm have the same rotor, then the input can be a single scalar entry as shown in the example. Otherwise, the list passed must have the same size as the number of entries in `turbine_locations`.

turbine_heights
: The list of tower heights for the turbines in the wind farm. If all the turbines in the wind farm have the same tower height, then the input can be a single scalar entry as shown in the example. Otherwise, the list passed must have the same size as the number of entries in `turbine_locations`.

turbine_locations
: The `(x, y, z)` coordinates of the turbine base in the wind farm.

orientation
: The orientation of the refinement boxes. Currently there is only one option available indicated by `type` param-

eter: `wind_direction`. For this option, it expects the `wind_direction` variable to contain the compass direction in degrees.

refinement_levels
A list of 4 parameters for each nested refinement zone. The three parameters are the distance upstream, distance downstream, the lateral and vertical extents of the refinement zone. These parameters are non-dimensional and are internally scaled by the turbine diameters by the utility. The nested boxes must be specified with the largest box first and the subsequent sizes in descending order.

search_tolerance
The tolerance parameter added when searching for elements enclosed by the refinement box. A value slightly larger than the coarsest mesh size is recommended.

refine_field_name
The name of the `error_indicator_field` used when creating STK fields. Default is `turbine_refinement_field`.

write_percept_files
Boolean flag indicating whether input files for use with Percept is written out by this utility as part of the run. Default: `true`.

percept_file_prefix
The prefix used for the Percept input file name. The default value is `adapt`. With the default file name and three levels of refinement, it will create three input files: `adapt1.yaml`, `adapt2.yaml`, and `adapt3.yaml`.

3.5 init_channel_fields

This task initializes the velocity fields for channel flow simulations based on the parameters provided by the user and writes it out to the *output_db*. It is safe to run `init_channel_fields` in parallel. A sample invocation is shown below

```
1 init_channel_fields:
2   fluid_parts: [Unspecified-2-HEX]
3
4   velocity:
5     Re_tau : 550
6     viscosity : 0.0000157
```

fluid_parts
A list of element block names where the velocity fields are to be initialized.

velocity
A YAML dictionary containing two values: the friction Reynolds number, `Re_tau`, and the kinematic `viscosity` (m^2/s).

3.6 create_bdy_io_mesh

Create an I/O transfer mesh containing the boundaries of a given ABL precursor mesh. The I/O transfer mesh can be used with Nalu during the precursor runs to dump inflow planes for use with a later wind farm LES simulation with inflow/outflow boundaries. Unlike other utilities described in this section, this utility creates a new mesh instead of adding to the database written out by the **nalu_preprocess** executable. It is safe to invoke this task in a parallel MPI run.

output_db
Name of the I/O transfer mesh where the boundary planes are written out. This argument is mandatory.

boundary_parts
A list of boundary parts that are saved in the I/O mesh. The names in the list must correspond to the names of the sidesets in the given ABL mesh.

3.7 move_mesh

Translates a mesh in space by a given offset vector.

mesh_parts
List of element block names that must be translated

offset_vector
A 3-D vector that specifies the translation in space.

```
nalu_preprocess:
  input_db: abl_1x1x1_10.exo
  output_db: move_mesh.g

  tasks:
    - move_mesh

  move_mesh:
    mesh_parts:
      - fluid

    offset_vector: [10.0, 10.0, 0.0]
```

3.8 rotate_mesh

Rotates the mesh given angle, origin, and axis using quaternion rotations.

mesh_parts
A list of element block names that must be rotated.

angle
The rotation angle in degrees.

origin
An (x, y, z) coordinate for mesh rotation.

axis
A unit vector about which the mesh is rotated.

```
1 rotate_mesh:
2   mesh_parts:
3     - unspecified-2-hex
4
5   angle: 30.0
6   origin: [500.0, 0.0, 0.0]
7   axis: [0.0, 0.0, 1.0]
```

3.9 generate_planes

Deprecated since version Since: 2018-09-01

Generates horizontal planes of nodesets at given heights that are used for sampling velocity and temperature fields during an ABL simulation. The resulting spatial average at given heights is used within Nalu to determine the driving pressure gradient necessary to achieve the desired ABL profile during the simulation. This task is capable of running in parallel.

The horizontal extent of the sampling plane can be either prescribed manually, or the program will use the bounding box of the input mesh. Note that the latter approach only works if the mesh boundaries are oriented along the major axes. The extent and orientation of the sampling plane is controlled using the `boundary_type` option in the input file.

boundary_type
Flag indicating how the program should estimate the horizontal extents of the sampling plane when generating nodesets. Currently, two options are supported:

Type	Description
bounding_box	Automatically estimate based on bounding box of the mesh
quad_vertices	Use user-provided `vertices`

This flag is optional, and if it is not provided the program defaults to using the `bounding_box` approach to estimate horizontal extents.

fluid_part
A list of element block names used to compute the extent using bounding box approach.

heights
A list of vertical heights where the nodesets are generated.

part_name_format
A `printf` style string that takes one floating point argument `%f` representing the height of the plane. For example, if the user desires to generate nodesets at 70m and 90m respectively and desires to name the plane `zh_070` and `zh_090` respectively, this can be achieved by setting `part_name_format: zh_%03.0f`.

dx, dy
Uniform resolutions in the x- and y-directions when generating nodesets. Used only when *`boundary_type`* is set to `bounding_box`.

nx, ny
Number of subdivisions of along the two axes of the quadrilateral provided. Given 4 points, `nx` will divide segments `1-2` and `3-4`, and `ny` will divide segments `2-3` and `4-1`. Used only when *`boundary_type`* is set to `quad_vertices`.

vertices
Used to provide the horizontal extents of the sampling plane to the utility. For example

```
vertices:
  - [250.0, 0.0]       # Vertex 1 (S-W corner)
  - [500.0, -250.0]    # Vertex 2 (S-E corner)
  - [750.0, 0.0]       # Vertex 3 (N-E corner)
  - [500.0, 250.0]     # Vertex 4 (N-W corner)
```

3.9.1 Example using custom vertices

```yaml
generate_planes:
  boundary_type: quad_vertices        # Override default behavior
  fluid_part: Unspecified-2-hex       # Fluid part

  heights: [ 70.0 ]                   # Heights were sampling planes are generated
  part_name_format: "zplane_%06.1f"   # Name format for new nodesets
  nx: 25                              # X resolution
  ny: 25                              # Y resolution
  vertices:                           # Vertices of the quadrilateral
    - [250.0, 0.0]
    - [500.0, -250.0]
    - [750.0, 0.0]
    - [500.0, 250.0]
```

CHAPTER

FOUR

NALU_POSTPROCESS – NALU POST-PROCESSING UTILITIES

This utility loads an Exodus-II solution file and performs various post-processing *tasks* on the database. Currently, the following *tasks* have been implemented within this utility.

Task type	Description
abl_statistics	Calculate various ABL statistics of interest

The input file (`download`) must contain a **nalu_postprocess** section a shown below. Input options for various *tasks* are provided as sub-sections within **nalu_postprocess** with the corresponding task names under tasks.

```
# Example input file for Nalu Post-processing utility

nalu_postprocess:

  # Name of the solution results or restart database
  input_db: rst/precursor.e

  # List of post-processing tasks to be performed
  tasks:
    - abl_statistics

  # Input parameters for the post-processing tasks
  abl_statistics:
    fluid_parts:
      - Unspecified-2-HEX

    field_map:
      velocity: velocity_raone
      temperature: temperature_raone
      sfs_stress: sfs_stress_raone

    height_info:
      min_height: 0.0
      max_height: 1000.0
      delta_height: 10.0
```

4.1 Command line invocation

```
mpiexec -np <N> nalu_postprocess -i [YAML_INPUT_FILE]
```

-i, --input-file
Name of the YAML input file to be used. Default: `nalu_postprocess.yaml`.

4.2 Common input file options

input_db
: Path to an existing Exodus-II mesh database file, e.g.., `ablPrecursor.e`

tasks
: A list of task names that define the various pre-processing tasks that will be performed on the input mesh database by this utility. The program expects to find additional sections with headings matching the task names that provide additional inputs for individual tasks.

4.3 abl_statistics

This task computes various various statistics relevant for ABL simulations and outputs vertical profiles of various quantities of interest.

```
# Input parameters for the post-processing tasks
abl_statistics:
  fluid_parts:
    - Unspecified-2-HEX

  field_map:
    velocity: velocity_raone
    temperature: temperature_raone
    sfs_stress: sfs_stress_raone

  height_info:
    min_height: 0.0
    max_height: 1000.0
    delta_height: 10.0
```

CHAPTER

FIVE

WRFTONALU – WRF TO NALU CONVERTOR

This program converts WRF data to the Nalu (Exodus II) data format. Exodus II is part of SEACAS and one can find other utilities to work with Exodus II files there. The objective is to provide Nalu with input WRF data as boundary conditions (and, optionally, initial conditions).

This program was started as `WRFTOOF`, a WRF to OpenFoam converter, which was written by J. Michalakes and M. Churchfield. It was adapted for converting to Nalu data by M. T. Henry de Frahan.

Note: This utility is not built by default. The user must set `ENABLE_WRFTONALU` to `ON` during the *CMake configure phase*.

5.1 Command line invocation

```
bash$ wrftonalu [options] wrfout
```

where `wrfout` is the WRF data file used to generate inflow conditions for the Nalu simulations. The user must provide the relevant boundary files in the run directory named `west.g`, `east.g`, `south.g`, `north.g`, `lower.g`, and `upper.g`. Only the boundaries where inflow data is required need to exist. The interpolated WRF data is written out to files with extension `*.nc` for the corresponding grid files for use with Nalu. The following optional parameters can be supplied to customize the behavior of **`wrftonalu`**.

`-startdate`
: Date string of the form `YYYY-mm-dd_hh_mm_ss` or `YYYY-mm-dd_hh:mm:ss`

`-offset`
: Number of seconds to start Exodus directory naming (default: 0)

`-coord_offset` `lat lon`
: Latitude and longitude of origin for Exodus mesh. Default: center of WRF data.

`-ic`
: Populate initial conditions as well as boundary conditions.

`-qwall`
: Generate temperature flux for the terrain (lower) BC file.

CHAPTER

SIX

ABL_MESH – BLOCK HEX MESH GENERATION

The abl_mesh executable can be used to generate structured mesh with HEX-8 elements in Exodus-II format. It can generate meshes from scratch or convert from other formats to Exodus-II format.

6.1 Command line invocation

```
bash$ abl_mesh -i abl_mesh.yaml

Nalu ABL Mesh Generation Utility
Input file: abl_mesh.yaml
HexBlockMesh: Registering parts to meta data
     Mesh block: fluid_part
Num. nodes = 1331; Num elements = 1000
     Generating node IDs...
     Creating nodes... 10% 20% 30% 40% 50% 60% 70% 80% 90%
     Generating element IDs...
     Creating elements... 10% 20% 30% 40% 50% 60% 70% 80% 90%
     Finalizing bulk modifications...
     Generating X Sideset: west
     Generating X Sideset: east
     Generating Y Sideset: south
     Generating Y Sideset: north
     Generating Z Sideset: terrain
     Generating Z Sideset: top
     Generating coordinates...
Writing mesh to file: ablmesh.exo
```

-i, --input-file
YAML input file to be processed for mesh generation details. Default: nalu_abl_mesh.yaml.

6.2 Common Input File Parameters

The input file must contain a nalu_abl_mesh section that contains the input parameters.

mesh_type
This variable can take the following options:

- generate_ablmesh - Will generate a structured HEX mesh, and is the default for mesh_type if not present in the input file. See *Structured Mesh Generation* for more details.
- convert_plot3d - Converts a Plot3D binary file to Exodus-II format for use with Nalu. See *Converting Plot3D to Exodus-II* for more details.

output_db [nalu_abl_mesh]
The Exodus-II filename where the mesh is output. No default, must be provided by the user.

fluid_part_name
Name of the element block created with HEX-8 elements. Default value: `fluid_part`.

ioss_8bit_ints
Boolean flag that enables output of 8-bit ints when writing Exodus mesh. Default value: false.

6.2.1 Boundary names

The user has the option to provide custom boundary names through the input file. Use the boundary name input parameters to change the default parameters. If these are not provided the default boundary names are described below:

Boundary	Default sideset name
xmin_boundary_name	west
xmax_boundary_name	east
ymin_boundary_name	south
ymax_boundary_name	north
zmin_boundary_name	terrain
zmax_boundary_name	top

6.3 Structured Mesh Generation

The interface is similar to OpenFOAM's `blockMesh` utility and can be used to generate simple meshes for ABL simulations on flat terrain without resorting to commercial mesh generation software, e.g., Pointwise.

A sample input file is shown below

```
nalu_abl_mesh:
  mesh_type: generate_ablmesh
  output_db: ablmesh.exo

  spec_type: bounding_box

  vertices:
    - [0.0, 0.0, 0.0]
    - [10.0, 10.0, 10.0]

  mesh_dimensions: [10, 10, 10]
```

spec_type
Specification type used to define the extents of the structured HEX mesh. This option is used to interpret the *vertices* read from the input file. Currently, two options are supported:

Type	Description
bounding_box	Use axis aligned bounding box as domain boundaries
vertices	Use user provided vertices to define extents

vertices
The coordinates specifying the extents of the computational domain. This entry is interpreted differently depending on the *spec_type*. If type is set to `bounding_box` then the code expects a list of two 3-D coordinate

points describing bounding box to generate an axis aligned mesh. Otherwise, the code expects a list of 8 points describing the vertices of the trapezoidal prism.

mesh_dimensions
Mesh resolution for the resulting structured HEX mesh along each direction. For a trapezoidal prism, the code will interpret the major axis along `1-2`, `1-4`, and `1-5` edges respectively.

6.3.1 Mesh spacing

Users can specify the mesh spacing to be applied in each direction by adding additional sections (`x_spacing`, `y_spacing`, and `z_spacing` respectively) to the input file. If no option is specified then a constant mesh spacing is used in that direction.

Available options	Implementation
`constant_spacing`	*ConstantSpacing*
`geometric_stretching`	*GeometricStretching*

Example input file

```
# Specifiy constant spacing in x direction (this is the default)
x_spacing:
  spacing_type: constant_spacing

# y direction has a mesh stretching factor
y_spacing:
  spacing_type: geometric_stretching
  stretching_factor: 1.1

# z direction has a mesh stretching factor in both directions
z_spacing:
  spacing_type: geometric_stretching
  stretching_factor: 1.1
  bidirectional: true
```

6.4 Limitations

1. Does not support the ability to generate multiple blocks
2. Must be run on a single processor, running with multiple MPI ranks is currently unsupported.

6.5 Converting Plot3D to Exodus-II

An example input block is shown below:

```
nalu_abl_mesh:
  mesh_type: convert_plot3d
  plot3d_file: grid.p3d
  output_db: p3d_grid.exo
```

plot3d_file
Path to the Plot3D grid file in binary format.

CHAPTER

SEVEN

`SLICE_MESH` – SAMPLING PLANE GENERATION

The `slice_mesh` executable can be used to generate sampling planes that can be used with I/O transfer interface of Nalu-Wind for extract subsets of data from wind farm simulations.

7.1 Command line invocation

```
bash$ slice_mesh -i slice_mesh.yaml

Slice Mesh Generation Utility
Input file: slice_mesh.yaml
Loading slice inputs...
Initializing slices...
Slice: Registering parts to meta data:
  -  turbine1_1
  -  turbine1_2
Slice: Registering parts to meta data:
  -  turbine2_1
  -  turbine2_2
Generating slices for: turbine1
Creating nodes... 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Creating elements... 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Generating coordinate field
 - turbine1_1
 - turbine1_2
Generating slices for: turbine2
Creating nodes... 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Creating elements... 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Generating coordinate field
 - turbine2_1
 - turbine2_2
Writing mesh to file: sampling_planes.exo

Memory usage: Avg:   10.957 MB; Min:   10.957 MB; Max:   10.957 MB
```

-i, --input-file
YAML input file to be processed. Default: `slice_mesh.yaml`

CHAPTER

EIGHT

BOXTURB – TURBULENCE BOX UTILITY

The `boxturb` executable is used to convert binary turbulence files into NetCDF format that can be read during Nalu-Wind simulations. In addition to conversion, it allows the user to apply divergence correction and scaling the different components through the input file.

8.1 Command line invocation

```
bash$ boxturb -i boxturb.yaml

Nalu Turbulent File Processing Utility
Input file: boxturb.yaml
Begin loading WindSim turbulence data
     Loading file: sim1u.bin
     Loading file: sim1v.bin
     Loading file: sim1w.bin
Begin output in NetCDF format: turbulence.nc
NetCDF file written successfully: turbulence.nc
```

-i, --input-file
YAML inout file that contains inputs for the executable. Default: *boxturb.yaml*

8.2 Sample input file

```yaml
boxturb:
  data_format: windsim
  output: turbulence.nc

  box_dims: [1024, 128, 128]
  box_len: [2400.0, 160.0, 160.0]

  bin_filenames:
    - sim1u.bin
    - sim1v.bin
    - sim1w.bin

  correct_divergence: yes

  solver_settings:
    method: pfmg
    preconditioner: none
```

(continues on next page)

(continued from previous page)

```
    max_iterations: 200
    tolerance: 1.0e-8
    print_level: 1
    log_level: 1

  # Scaling factor
  apply_scaling: yes
  scale_type: default
  scaling_factors: [1.0, 0.7, 0.3]
```

Part II

Developer Manual

CHAPTER

NINE

INTRODUCTION

This part of the documentation is intended for users who wish to extend or add new functionality to the NaluWindUtilities toolsuite. End users who want to use existing utilities should consult the *User Manual* for documentation on standalone utilities.

9.1 Version Control System

Like Nalu, NaluWindUtils uses Git SCM to track all development activity. All development is coordinated through the Github repository. Pro Git, a book that covers all aspects of Git is a good resource for users unfamiliar with Git SCM. Github Desktop and Git Kraken are two options for users who prefer a GUI based interaction with Git source code.

9.2 Building API Documentation

In-source comments can be compiled and viewed as HTML files using Doxygen. If you want to generate class inheritance and other collaboration diagrams, then you will need to install Graphviz in addition to Doxygen.

1. API Documentation generation is disabled by default in CMake. Users will have to enable this by turning on the *ENABLE_DOXYGEN_DOCS* flag.
2. Run `make api-docs` to generate the documentation in HTML form.

The resulting documentation will be available in `doc/doxygen/html/` within the CMake build directory.

9.3 Contributing

The project welcomes contributions from the wind research community. Users can contribute to the source code using the normal Github fork and pull request workflow. Please follow these general guidelines when submitting pull requests to this project

- All C++ code must conform to the C++11 standard. Consult C++ Core Guidelines on best-practices to writing idiomatic C++ code.
- Check and fix all compiler warnings before submitting pull requests. Use `-Wall -Wextra -pedantic` options with GNU GCC or LLVM/Clang to check for warnings.
- New feature pull-requests must include doxygen-compatible *in source* documentation, additions to user manual describing the enchancements and their usage, as well as the necessary updates to CMake files to enable configuration and build of these capabilities.

- Prefer Markdown format when documenting code using Doxgen-compatible comments.
- Avoid incurring additional third-party library (TPL) dependencies beyond what is required for building Nalu. In cases where this is unavoidable, please discuss this with the development team by creating an issue on issues page before submitting the pull request.

CHAPTER

TEN

NALU PRE-PROCESSING UTILITIES

NaluWindUtils provides several pre-processing utilities that are built as subclasses of *PreProcessingTask*. These utilities are configured using a YAML input file and driven through the *PreProcessDriver* class – see *nalu_preprocess – Nalu Preprocessing Utilities* for documentation on the available input file options. All pre-processing utilities share a common interface and workflow through the *PreProcessingTask* API, and there are three distinct phases for each utility namely: construction, initialization, and execution. The function of each of the three phases as well as the various actions that can be performed during these phases are described below.

10.1 Task Construction Phase

The driver initializes each *task* through a constructor that takes two arguments:

- *CFDMesh* – a mesh instance that contains the MPI communicator, STK MetaData and BulkData instances as well as other mesh related utilities.
- `YAML::Node` – a yaml-cpp node instance containing the user defined inputs for this particular task.

The driver class initializes the instances in the order that was specified in the YAML input file. However, the classes must not assume existence or dependency on other task instances.

The base class *PreProcessingTask* already stores a reference to the `CFDMesh` instance in `mesh_`, that is accessible to subclasses via protected access. It is the responsibility of the individual task instances to process the YAML node during construction phase. Currently, this is typically done via the `load()`, a private method in the concrete task specialization class.

No actions on STK MetaData or BulkData instances should be performed during the construction phase. The computational mesh may not be loaded at this point. The construction should only initialize the class member variables that will be used in subsequent phases. The instance may store a reference to the YAML Node if necessary, but it is better to process and validate YAML data during this phase and store them as class member variables of correct types.

It is recommended that all tasks created support execution in parallel and, if possible, handle both 2-D and 3-D meshes. However, where this is not possible, the implementation much check for the necessary conditions via asserts and throw errors appropriately.

10.2 Task Initialization Phase

Once all the task instances have been created and each instance has checked the validity of the user provided input files, the driver instance calls the `initialize` method on all the available task instances. All `stk::mesh::MetaData` updates, e.g., part or field creation and registration, must be performed during this phase. No `stk::mesh::BulkData` modifications should be performed during this stage. Some tips for proper initialization of parts and fields:

- Access to `stk::mesh::MetaData` and `stk::mesh::BulkData` is through *`meta()`* and *`bulk()`* respectively. They return non-const references to the instances stored in the mesh object.
- Use `MetaData::get_part()` to check for the existence of a part in the mesh database, `MetaData::declare_part()` will automatically create a part if none exists in the database.
- As with parts, use `MetaData::declare_field()` or `MetaData::get_field()` to create or perform checks for existing fields as appropriate.
- New fields created by pre-processing tasks must be registered as an output field if it should be saved in the result output ExodusII database. The default option is to not output all fields, this is to allow creation of temporary fields that might not be necessary for subsequent Nalu simulations. Field registration for output is achieved by calling *`add_output_field()`* from within the `initialize()` method.

```
// Register velocity and temperature fields for output
mesh_.add_output_field("velocity");
mesh_.add_output_field("temperature");
```

- The *coordinates* field is registered on the universal part, so it is not strictly necessary to register this field on newly created parts.

Once all tasks have been initialized, the driver will **commit** the STK MetaData object and populate the BulkData object. At this point, the mesh is fully loaded and BulkData modifications can begin and the driver moves to the execution phase.

10.3 Task Execution Phase

The driver initiates execution phase of individual tasks by calling the `run()` method, which performs the core pre-processing task of the instance. Since STK MetaData has been committed, no further MetaData modifications (i.e., part/field creation) can occur during this phase. All actions at this point are performed on the BulkData instance. Typical examples include populating new fields, creating new entities (nodes, elements, sidesets), or moving mesh by manipulating coordinates. If the mesh does not explicitly create any new fields, the *task* instance can still force a write of the output database by calling the *`set_write_flag()`* to indicate that the database modifications must be written out. By default, no output database is created if no actions were performed.

10.4 Task Destruction Phase

All *task* implementations must provide proper cleanup procedures via destructors. No explicit clean up task methods are called by the driver utility. The preprocessing utility depends on C++ destructor actions to free resources etc.

10.5 Registering New Utility

The *`sierra::nalu::PreProcessingTask`* class uses a runtime selection mechanism to discover and initialize available utilities. To achieve this, new utilities must be registered by invoking a pre-defined macro (`REGISTER_DERIVED_CLASS`) that wrap the logic necessary to register classes with the base class. For example, to register a new utility `MyNewUtility` the developer must add the following line

```
REGISTER_DERIVED_CLASS(PreProcessingTask, MyNewUtility, "my_new_utility");
```

in the C++ implementation file (i.e., the `.cpp` file and not the `.h` header file). In the above example, `my_new_utility` is the lookup *type* (see *`tasks`*) used by the driver when processing the YAML input file. Note that this macro must be invoked from within the `sierra::nalu` namespace.

CHAPTER

ELEVEN

NALUWINDUTILS API DOCUMENTATION

11.1 Core Utilities

11.1.1 CFDMesh

class CFDMesh

STK Mesh interface.

This class provides a thin wrapper around the STK mesh objects (MetaData, BulkData, and StkMeshIoBroker) for use with various preprocessing utilities.

Public Functions

CFDMesh(stk::ParallelMachine &*comm*, **const** std::string *filename*)

Create a CFD mesh instance from an existing mesh database.

Parameters

- `comm`: MPI Communicator object
- `filename`: Exodus database filename

CFDMesh(stk::ParallelMachine &*comm*, **const** int *ndim*)

Create a CFD mesh instance from scratch.

Parameters

- `comm`: MPI Communicator object
- `ndim`: Dimensionality of mesh

~CFDMesh()

void **init**(stk::io::DatabasePurpose *db_purpose* = stk::io::READ_MESH)

Initialize the mesh database.

If an input DB is provided, the mesh is read from the file. The MetaData is committed and the BulkData is ready for use/manipulation.

stk::ParallelMachine &**comm**()

Reference to the MPI communicator object.

stk::mesh::MetaData &**meta**()
Reference to the stk::mesh::MetaData instance.

stk::mesh::BulkData &**bulk**()
Reference to the stk::mesh::BulkData instance.

stk::io::StkMeshIoBroker &**stkio**()
Reference to the STK mesh I/O instance.

void **add_output_field**(**const** std::string *field*)
Register a field for output during write.

Parameters

- `field`: Name of the field to be output

size_t **open_database**(std::string *output_db*)
Open a database for writing time series data.

Return A valid file handle for use with write_database

See *write_database*, *write_timesteps*

Parameters

- `output_db`: Pathname to the output ExodusII database

void **write_database**(size_t *fh*, double *time*)
Write time series data to an open database.

See *open_database*, *write_timesteps*

Parameters

- `fh`: Valid file handle
- `time`: Time to write

void **write_database**(std::string *output_db*, double *time* = 0.0)
Write the Exodus results database with modifications.

See *write_database_with_fields*

Parameters

- `output_db`: Pathname to the output ExodusII database
- `time`: Timestep to write

Parameters

- `output_db`: Filename for the output Exodus database
- `time`: (Optional) time to write (default = 0.0)

void **write_database_with_fields**(std::string *output_db*)
Write database with restart fields.

Copies the restart data fields from the input Exodus database to the output database.

Parameters

- `output_db`: Pathname to the output ExodusII database

template<typename **Functor**>
void **write_timesteps**(std::string *output_db*, int *num_steps*, *Functor* *lambdaFunc*)
Write time-history to database.

This method accepts a functor that takes one integer argument (timestep) and returns the time (double) that must be written to the database. The functor should update the fields that are being written to the database. An example would be to simulate mesh motion by updating the mesh_displacement field at every timestep.

The following example shows the use with a C++ lambda function:

```
double deltaT = 0.01;  // Timestep size

write_timesteps("inflow_history.exo", 100,
  [&](int tstep) {
      double time = tstep * deltaT;

      // Update velocity and coordinates

      return time;
});
```

BoxType **calc_bounding_box**(**const** stk::mesh::Selector *selector*, bool *verbose* = true)
Calculate the bounding box of the mesh.

The selector can pick parts that are not contiguous. However, the bounding box returned will be the biggest box that encloses all parts selected.

Return An stk::search::Box instance containing the min and max points (3-D).

Parameters

- `selector`: An instance of stk::mesh::Selector to filter parts of the mesh where bounding box is calculated.
- `verbose`: If true, then print out the bounding box to standard output.

void **set_decomposition_type**(std::string *decompType*)
Set automatic mesh decomposition property.

Valid decomposition types are: rcb, rib, block, linear

Parameters

- `decompType`: The decomposition type

void **set_64bit_flags**()
Force output database to use 8-bit integers.

bool **db_modified**()
Flag indicating whether the DB has been modified.

void **set_write_flag**(bool *flag* = true)
Force output of the results DB.

const std::unordered_set<std::string> &**output_fields**()
Return a reference to the registered output fields.

11.1.2 Interpolation utilities

struct OutOfBounds
Flags and actions for out-of-bounds operation.

Public Types

enum boundLimits
Out of bounds limit types.

Values:

LOWLIM = -2
xtgt < xarray[0]

UPLIM = -1
xtgt > xarray[N]

VALID = 0
xarray[0] <= xtgt <= xarray[N]

enum OobAction
Flags indicating action to perform on Out of Bounds situation.

Values:

ERROR = 0
Raise runtime error.

WARN
Warn and then CLAMP.

CLAMP
Clamp values to the end points.

EXTRAPOLATE
Extrapolate linearly based on end point.

template<typename **T**>
InterpTraits<*T*>::index_type `sierra::nalu::utils::`**check_bounds**(**const** Array1D<*T*> &*xinp*, **const** *T* &*x*)
Determine whether the given value is within the limits of the interpolation table.

Return A std::pair containing the *OutOfBounds* flag and the index (0 or MAX)

Parameters

- `xinp`: 1-D array of monotonically increasing values
- `x`: The value to check for

template<typename **T**>
InterpTraits<*T*>::index_type `sierra::nalu::utils::`**find_index**(**const** Array1D<*T*> &*xinp*, **const** *T* &*x*)
Return an index object corresponding to the x-value based on interpolation table.

Return The `std::pair` returned contains two values: the bounds indicator and the index of the element in the interpolation table such that `xarray[i] <= x < xarray[i+1]`

Parameters

- `xinp`: 1-D array of monotonically increasing values

- `x`: The value to check for

template<typename **T**>
void `sierra::nalu::utils::`**linear_interp**(**const** Array1D<*T*> &*xinp*, **const** Array1D<*T*> &*yinp*, **const** *T* &*xout*, *T* &*yout*, *OutOfBounds*::*OobAction* *oob* = *OutOfBounds*::*CLAMP*)

Perform a 1-D linear interpolation.

Parameters

- `xinp`: A 1-d vector of monotonically increasing x-values
- `yinp`: Corresponding 1-d vector of y-values
- `xout`: Target x-value for interpolation
- `yout`: Interpolated value at `xout`
- `oob`: (Optional) Out-of-bounds handling (default: CLAMP)

11.1.3 YAML utilities

Miscellaneous utilities for working with YAML C++ library.

namespace sierra

namespace nalu

namespace wind_utils

Functions

template<typename **T**>
bool **get_optional**(**const** YAML::Node &*node*, **const** std::string &*key*, *T* &*result*)

Fetch an optional entry from the YAML dictionary if it exists.

The result parameter is unchanged if the entry is not found in the YAML dictionary.

Parameters

- `node`: The YAML::Node instance to be examined
- `key`: The name of the variable to be extracted
- `result`: The variable that is updated with the value if it exists

template<typename **T**>
bool **get_optional**(**const** YAML::Node &*node*, **const** std::string &*key*, *T* &*result*, **const** *T* &*default_value*)

Fetch an optional entry from the YAML dictionary if it exists.

The result parameter is updated with the value from the dictionary if it exists, otherwise it is initialized with the default value provided.

Parameters

- `node`: The YAML::Node instance to be examined
- `key`: The name of the variable to be extracted
- `result`: The variable that is updated with the value if it exists

- `default_value`: The default value to be used if the parameter is not found in the dictionary.

11.1.4 Performance Monitoring Utilities

namespace sierra

namespace nalu

Functions

Teuchos::RCP<Teuchos::Time> **get_timer**(**const** std::string &*name*)
Return a timer identified by name.

If an existing timer is found, then the timer is returned. Otherwise a new timer is created. The user will have to manually start/stop the timer. For most use cases, it might be preferable to use `get_stopwatch` function instead.

Teuchos::TimeMonitor **get_stopwatch**(**const** std::string &*name*)
Return a stopwatch identified by name.

The clock starts automatically upon invocation and will be stopped once the `Teuchos::Timemonitor` instance returned by this function goes out of scope.

11.2 Pre-processing Utilities

11.2.1 PreProcessDriver

class PreProcessDriver
A driver that runs all preprocessor tasks.

This class is responsible for reading the input file, parsing the user-requested list of tasks, initializing the task instances, executing them, and finally writing out the updated Exodus database with changed inputs.

Public Functions

PreProcessDriver(stk::ParallelMachine &*comm*, **const** std::string *filename*)

Parameters

- `comm`: MPI Communicator reference
- `filename`: Name of the YAML input file

void **run**()
Run all tasks and output the updated Exodus database.

11.2.2 PreProcessingTask

class PreProcessingTask

An abstract implementation of a *PreProcessingTask*.

This class defines the interface for a pre-processing task and contains the infrastructure to allow concrete implementations of pre-processing tasks to register themselves for automatic runtime discovery. Derived classes must implement two methods:

- `initialize` - Perform actions on STK MetaData before processing BulkData
- `run` - All actions on BulkData and other operations on mesh after it has been loaded from the disk.

For automatic class registration, the derived classes must implement a constructor that takes two arguments: a *CFDMesh* reference, and a `const` reference to YAML::Node that contains the inputs necessary for the concrete task implementation. It is the derived class' responsibility to process the input dictionary and perform error checking. No STK mesh manipulations must occur in the constructor.

Subclassed by *sierra::nalu::ABLFields*, *sierra::nalu::BdyIOPlanes*, *sierra::nalu::ChannelFields*, sierra::nalu::HITFields, sierra::nalu::InflowHistory, *sierra::nalu::NDTW2D*, *sierra::nalu::NestedRefinement*, *sierra::nalu::RotateMesh*, *sierra::nalu::SamplingPlanes*, sierra::nalu::TranslateMesh

Public Functions

PreProcessingTask(*CFDMesh* &*mesh*)

Parameters

- `mesh`: A *sierra::nalu::CFDMesh* instance

virtual void **initialize**() = 0

Initialize the STK MetaData instance.

This method handles the registration and creation of new parts and fields. All subclasses must implement this method.

virtual void **run**() = 0

Process the STK BulkData instance.

This method handles the creating of new entities, manipulating coordinates, and populating fields.

Public Static Functions

PreProcessingTask ***create**(*CFDMesh* &*mesh*, **const** YAML::Node &*node*, std::string *lookup*)

Runtime creation of concrete task instance.

Protected Attributes

CFDMesh &**mesh_**

Reference to the *CFDMesh* instance.

11.2.3 ABLFields

class ABLFields : **public** *sierra*::*nalu*::PreProcessingTask

Initialize velocity and temperature fields for ABL simulations.

This task is activated by using the `init_abl_fields` task in the preprocessing input file. It requires a section `init_abl_fields` in the `nalu_preprocess` section with the following parameters:

```
init_abl_fields:
  fluid_parts: [Unspecified-2-HEX]

  temperature:
    heights: [    0, 650.0, 750.0, 10750.0]
    values:  [280.0, 280.0, 288.0,   318.0]

  velocity:
    heights: [0.0, 10.0, 30.0, 70.0, 100.0, 650.0, 10000.0]
    values:
      - [ 0.0, 0.0, 0.0]
      - [4.81947, -4.81947, 0.0]
      - [5.63845, -5.63845, 0.0]
      - [6.36396, -6.36396, 0.0]
      - [6.69663, -6.69663, 0.0]
      - [8.74957, -8.74957, 0.0]
      - [8.74957, -8.74957, 0.0]
```

The sections `temperature` and `velocity` are optional, allowing the user to initialize only the temperature or the velocity as desired. The heights are in meters, the temperature is the potential temperature in Kelvin, and the velocity is the actual vector in m/s. Currently, the code does not include the ability to automatically convert (mangitude, direction) to velocity vectors.

Public Functions

ABLFields(*CFDMesh* &*mesh*, **const** YAML::Node &*node*)

Parameters

- `mesh`: A *sierra::nalu::CFDMesh* instance
- `node`: The YAML::Node containing inputs for this task

void **initialize**()

Declare velocity and temperature fields and register them for output.

void **run**()

Initialize the velocity and/or temperature fields by linear interpolation.

Private Functions

void **load**(**const** YAML::Node &*abl*)

Parse the YAML file and initialize parameters.

void **load_velocity_info**(**const** YAML::Node &*abl*)

Helper function to parse and initialize velocity inputs.

void **load_temperature_info**(**const** YAML::Node &*abl*)

Helper function to parse and initialize temperature inputs.

void **init_velocity_field**()
Initialize the velocity field through linear interpolation.

void **init_temperature_field**()
Intialize the temperature field through linear interpolation.

void **perturb_velocity_field**()
Add perturbations to velocity field.

void **perturb_temperature_field**()
Add perturbations to temperature field.

Private Members

stk::mesh::MetaData &**meta_**
STK Metadata object.

stk::mesh::BulkData &**bulk_**
STK Bulkdata object.

stk::mesh::PartVector **fluid_parts_**
Parts of the fluid mesh where velocity/temperature is initialized.

std::vector<double> **vHeights_**
List of heights where velocity is defined.

Array2D<double> **velocity_**
List of velocity (3-d components) at the user-defined heights.

std::vector<double> **THeights_**
List of heights where temperature is defined.

std::vector<double> **TValues_**
List of temperatures (K) at user-defined heights (THeights_)

std::vector<std::string> **periodicParts_**
List of periodic parts.

double **deltaU_** = {1.0}
Velocity perturbation amplitude for Ux.

double **deltaV_** = {1.0}
Velocity perturbation amplitude for Uy.

double **Uperiods_** = {4.0}
Number of periods for Ux.

double **Vperiods_** = {4.0}
Number of periods for Uy.

double **zRefHeight_** = {50.0}
Reference height for velocity perturbations.

double **thetaAmplitude_**
Amplitude of temperature perturbations.

double **thetaGaussMean_** = {0.0}
Mean for the Gaussian random number generator.

double **thetaGaussVar_** = {1.0}
Variance of the Gaussian random number generator.

double **thetaCutoffHt_**
Cutoff height for temperature fluctuations.

int **ndim_**
Dimensionality of the mesh.

bool **doVelocity_**
Flag indicating whether velocity is initialized.

bool **doTemperature_**
Flag indicating whether temperature is initialized.

bool **perturbU_** = {false}
Flag indicating whether velocity perturbations are added during initialization.

bool **perturbT_** = {false}
Flag indicating whether temperature perturbations are added.

11.2.4 BdyIOPlanes

class BdyIOPlanes : **public** *sierra*::*nalu*::PreProcessingTask
Extract boundary planes for I/O mesh.

Given an ABL precursor mesh, this utility extracts the specified boundaries and creates a new IO Transfer mesh for use with ABL precursor simulations.

Public Functions

BdyIOPlanes(*CFDMesh* &*mesh*, **const** YAML::Node &*node*)

Parameters

- `mesh`: A *sierra::nalu::CFDMesh* instance
- `node`: The YAML::Node containing inputs for this task

void **initialize**()
Register boundary parts and attach coordinates to the parts.

The parts are created as SHELL elements to as needed by the Nalu Transfer class.

void **run**()
Copy user specified boundaries and save the IO Transfer mesh.

Private Functions

void **load**(**const** YAML::Node &*node*)
Parse user inputs from the YAML file.

void **create_boundary**(**const** std::string *bdyName*)
Copy the boundary from Fluid mesh to the IO Xfer mesh.

Private Members

CFDMesh &**mesh_**
Original mesh DB information.

CFDMesh **iomesh_**
IO Mesh db STK meta and bulk data.

std::vector<std::string> **bdyNames_**
User specified list of boundaries to be extracted.

std::string **output_db_** = {""}
Name of the I/O db where the boundaries are written out.

11.2.5 SamplingPlanes

class SamplingPlanes : **public** *sierra*::*nalu*::PreProcessingTask
Generate 2-D grids/planes for data sampling.

Currently only generates horizontal planes at user-defined heights.

Requires a section `generate_planes` in the input file within the `nalu_preprocess` section:

```
generate_planes:
  fluid_part: Unspecified-2-hex

  heights: [ 70.0 ]
  part_name_format: "zplane_%06.1f"

  dx: 12.0
  dy: 12.0
```

With the above input definition, it will use the bounding box of the `fluid_part` to determine the bounding box of the plane to be generated. This will provide coordinate axis aligned sapling planes in x and y directions. Alternately, the user can specify `boundary_type` to be `quad_vertices` and provide the vertices of the quadrilateral that is used to generate the sampling plane as shown below:

```
generate_planes:
  boundary_type: quad_vertices
  fluid_part: Unspecified-2-hex

  heights: [ 50.0, 70.0, 90.0 ]
  part_name_format: "zplane_%06.1f"

  nx: 25  # Number of divisions along (1-2) and (4-3) vertices
  ny: 25  # Number of divisions along (1-4) and (2-3) vertices
  vertices:
    - [250.0, 0.0]
    - [500.0, -250.0]
    - [750.0, 0.0]
    - [500.0, 250.0]
```

`part_name_format` is a printf-like format specification that takes one argument - the height as a floating point number. The user can use this to tailor how the nodesets or the shell parts are named in the output Exodus file.

Public Types

enum PlaneBoundaryType
Sampling Plane boundary type.

Values:

BOUND_BOX = 0
Use bounding box of the fluid mesh defined.

QUAD_VERTICES
Use user-defined vertex list for plane boundary.

Public Functions

void **initialize**()
Initialize the STK MetaData instance.

This method handles the registration and creation of new parts and fields. All subclasses must implement this method.

void **run**()
Process the STK BulkData instance.

This method handles the creating of new entities, manipulating coordinates, and populating fields.

Private Functions

void **calc_bounding_box**()
Use fluid Realm mesh to estimate the x-y bounding box for the sampling planes.

void **generate_zplane**(**const** double *zh*)
Generate entities and update coordinates for a given sampling plane.

Private Members

stk::mesh::MetaData &**meta_**
STK Metadata object.

stk::mesh::BulkData &**bulk_**
STK Bulkdata object.

std::vector<double> **heights_**
Heights where the averaging planes are generated.

std::array<std::array<double, 3>, 2> **bBox_**
Bounding box of the original mesh.

std::string **name_format_**
Format specification for the part name.

std::vector<std::string> **fluidPartNames_**
Fluid realm parts (to determine mesh bounding box)

stk::mesh::PartVector **fluidParts_**
Parts of the fluid mesh (to determine mesh bounding box)

double **dx_**
Spatial resolution in x and y directions.

double **dy_**
Spatial resolution in x and y directions.

size_t **nx_**
Number of nodes in x and y directions.

size_t **mx_**
Number of elements in x and y directions.

int **ndim_**
Dimensionality of the mesh.

PlaneBoundaryType **bdyType_** = {*BOUND_BOX*}
User defined selection of plane boundary type.

11.2.6 NestedRefinement

class NestedRefinement : **public** *sierra*::*nalu*::PreProcessingTask
Tag regions in mesh for refinement with Percept mesh_adapt utility.

This utility creates a field turbine_refinement_field that is populated with an indicator value between [0, 1] that can be used with the Percept mesh_adapt utility to locally refine regions of interest.

A typical use of this utility is to refine an ABL mesh around turbines, especially for use with actuator line wind farm simulations.

Public Functions

void **initialize**()
Initialize the refinement field and register to parts.

void **run**()
Perform search and tag elements with appropriate values for subsequent refinement.

Private Functions

void **load**(**const** YAML::Node &*node*)
Parse the YAML file and initialize the necessary parameters.

void **process_inputs**()
Process input data and populate necessary data structures for subsequent use.

double **compute_refine_fraction**(Vec3D &*point*)
Estimate the refinement fraction [0,1] for a given element, indicated by the element mid point.

void **write_percept_inputs**()
Write out the input files that can be used with Percept.

Private Members

std::vector<std::string> **fluidPartNames_**
Partnames for the ABL mesh.

stk::mesh::PartVector **fluidParts_**
Parts of the ABL mesh where refinement is performed.

std::vector<double> **turbineDia_**
List of turbine diameters for the turbines in the wind farm [numTurbines].

std::vector<double> **turbineHt_**
List of turbine tower heights for the turbines in wind farm [numTurbines].

std::vector<Vec3D> **turbineLocs_**
List of turbine pad locations [numTurbines, 3].

std::vector<std::vector<double>> **refineLevels_**
List of refinement levels [numLevels, 3].

std::vector<TrMat> **boxAxes_**
Transformation matrices for each turbine [numTurbines].

std::vector<Vec3D> **corners_**
The minimum corners for each refinement box [numTurbines * numLevels].

std::vector<Vec3D> **boxLengths_**
The dimensions of each box [numTurbines * numLevels].

std::string **refineFieldName_** = {"turbine_refinement_field"}
Field name used in the Exodus mesh for the error indicator field.

std::string **perceptFilePrefix_** = {"adapt"}
Prefix for the input file name.

double **searchTol_** = {10.0}
Search tolerance used when searching for box inclusion.

double **windAngle_** = {270.0}
Compass direction of the wind (in degrees)

size_t **numTurbines_**
The number of turbines in the wind farm.

size_t **numLevels_**
The number of refinement levels.

bool **writePercept_** = {true}
Write input files for use with subsequent percept run.

11.2.7 ChannelFields

class ChannelFields : **public** *sierra*::*nalu*::PreProcessingTask
Initialize velocity fields for channel flow simulations.

This task is activated by using the `init_channel_fields` task in the preprocessing input file. It requires a section `init_channel_fields` in the `nalu_preprocess` section with the following parameters:

```
init_channel_fields:
  fluid_parts: [Unspecified-2-HEX]
```

(continues on next page)

(continued from previous page)

```
  velocity:
    Re_tau : 550
    viscosity : 0.0000157
```

The user specified the friction Reynolds number, `Re_tau`, and the kinematic `viscosity` (in m^2/s). The velocity field is initialized to a Reichardt function, with an imposed sinusoidal perturbation and random perturbation in the wall parallel directions.

Public Functions

void **initialize**()
Declare velocity fields and register them for output.

void **run**()
Initialize the velocity fields by linear interpolation.

11.2.8 RotateMesh

class RotateMesh : **public** *sierra*::*nalu*::PreProcessingTask
Rotate a mesh.

```
rotate_mesh:
  mesh_parts:
    - unspecified-2-hex

  angle: 45.0
  origin: [500.0, 0.0, 0.0]
  axis: [0.0, 0.0, 1.0]
```

Public Functions

void **initialize**()
Initialize the STK MetaData instance.

This method handles the registration and creation of new parts and fields. All subclasses must implement this method.

void **run**()
Process the STK BulkData instance.

This method handles the creating of new entities, manipulating coordinates, and populating fields.

Private Members

stk::mesh::MetaData &**meta_**
STK Metadata object.

stk::mesh::BulkData &**bulk_**
STK Bulkdata object.

std::vector<std::string> **meshPartNames_**
Part names of the mesh that needs to be rotated.

stk::mesh::PartVector **meshParts_**
Parts of the mesh that need to be rotated.

double **angle_**
Angle of rotation.

std::vector<double> **origin_**
Point about which rotation is performed.

std::vector<double> **axis_**
Axis around which the rotation is performed.

int **ndim_**
Dimensionality of the mesh.

11.2.9 NDTW2D

class NDTW2D : **public** *sierra*::*nalu*::PreProcessingTask
2-D Nearest distance to wall calculator

Calculates a new field NDTW containing the wall distance for 2-D airfoil-like applications used in RANS wall models.

Public Functions

void **initialize** ( )
Initialize the NDTW field and register for output.

void **run** ( )
Calculate wall distance and update NDTW field.

11.3 Meshing Utilities

11.3.1 Mesh Generation and Conversion

class HexBlockBase
Base class representation of a structured hex mesh.

Subclassed by *sierra::nalu::HexBlockMesh*, *sierra::nalu::Plot3DMesh*

Public Types

enum SideIDType
Sideset definition type.

Values:

XMIN = 0

YMIN

ZMIN

XMAX

YMAX

ZMAX

Public Functions

void **initialize**()
Registers the element block and the sidesets to the STK MetaData instance.

void **run**()
Creates the nodes and elements within the mesh block, processes sidesets, and initializes the coordinates of the mesh structure.

Public Static Functions

HexBlockBase ***create**(*CFDMesh* &*mesh*, **const** YAML::Node &*node*, std::string *lookup*)
Runtime creation of mesh generator instance.

class HexBlockMesh : **public** *sierra*::*nalu*::HexBlockBase
Create a structured block mesh with HEX-8 elements.

Public Types

enum DomainExtentsType
Computational domain definition type.

Values:

BOUND_BOX = 0
Use bounding box to define mesh extents.

VERTICES
Provide vertices for the cuboidal domain.

Public Functions

HexBlockMesh(*CFDMesh* &*mesh*, **const** YAML::Node &*node*)

Parameters

- `mesh`: A *sierra::nalu::CFDMesh* instance
- `node`: The YAML::Node containing inputs for this task

class Plot3DMesh : **public** *sierra*::*nalu*::HexBlockBase

Mesh Spacing Options

class MeshSpacing
Abstract base class that defines the notion of mesh spacing.

This class provides an interface where mesh spacing for a structured mesh can be represented as a 1-D array of values (`0.0 <= ratio[i] <= 1.0`) in a particular direction, that represents the location of the i-th node on the mesh on a unit cube.

See *sierra::nalu::HexBlockMesh*

Subclassed by *sierra::nalu::ConstantSpacing*, *sierra::nalu::GeometricStretching*

Public Functions

virtual void **init_spacings** () = 0
Initialize spacings based on user inputs.

const std::vector<double> &**ratios** () **const**
A 1-D array of fractions that represents the distance from the origin for a unit cube.

Public Static Functions

MeshSpacing ***create** (int *npts*, **const** YAML::Node &*node*, std::string *lookup*)
Runtime creation of the concrete spacing instance.

class ConstantSpacing : **public** *sierra*::*nalu*::MeshSpacing
Constant mesh spacing distribution.

Specialization of *MeshSpacing* to allow for constant mesh spacing which is the default implementation if no user option is specified in the input file. This class requires no additional input arguments in the YAML file.

Public Functions

void **init_spacings** ()
Initialize a constant spacing 1-D mesh.

class GeometricStretching : **public** *sierra*::*nalu*::MeshSpacing
Create a mesh spacing distribution with a constant stretching factor.

Requires user to specify a constant stretching factor that is used, along with the number of elements, to determine the first cell height and the resulting spacing distribution on a one-dimensional mesh of unit length. Given a stretching factor s, the first cell height is calculated as

$$h_0 = L\left(\frac{s-1}{s^n-1}\right)$$

By default, the stretching factor is applied in one direction. The user can set the `bidirectional` flag to true to apply the stretching factors and spacings at both ends.

Public Functions

void **init_spacings** ()
Initialize spacings based on user inputs.

Part III

Indices and Tables

- genindex

INDEX

Symbols

A

B

C

D

E

F

H

I

M

N

O

P

R