NaluWindUtils Documentation

Release v0.1.0

Shreyas Ananthan, Marc Henry de Frahan

Sep 30, 2017

Contents

NaluWindUtils is a companion software library to Nalu — 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 NaluCFD 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 1

Introduction

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

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.

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

# 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.

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.

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* appropriately.

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

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.

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 2

`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
`generate_planes`	Generate horizontal sampling planes for `dp/dx` forcing
`create_bdy_io_mesh`	Create an I/O transfer mesh for sampling inflow planes
`rotate_mesh`	Rotate mesh

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
```

```
  # Flag indicating mesh decomposition type (for parallel runs)
  # automatic_decomposition_type: rcb

  # Nalu preprocessor expects a list of tasks to be performed on the mesh and
  # field data structures
  tasks:
    - init_abl_fields
    - generate_planes
```

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`.

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

  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`.

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: [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]
```

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.

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.

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.

---

generate_planes

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)
```

Example using custom vertices

```
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]
```

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.

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.

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

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

calc_ndtw2d

Deprecated since version 0.1.0: The implementation uses a brute-force method to compute the nearest wall distance and as such is unsitable for production use. Use only on small two-dimensional meshes.

Calculate the nearest distance to wall (NDTW) for 2-D airfoil meshes.

```
calc_ndtw2d:
  fluid_parts:
    - Unspecified-2-QUAD
    - Unspecified-3-QUAD

  wall_parts:
    - airfoil
```

CHAPTER 3

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*.

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 4

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. 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.

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.

Input File Parameters

The input file must contain a `nalu_abl_mesh` section that contains the input parameters.A sample input file is shown below

```
nalu_abl_mesh:
  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]
```

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

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.

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

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`

Limitations

1. Currently the code is setup to only generate constant size grids in each direction.
2. Does not support the ability to generate multiple blocks
3. Must be run on a single processor, running with multiple MPI ranks is currently unsupported.

Part II

Developer Manual

CHAPTER 5

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.

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.

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.

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 6

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.

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.

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.

Task Execution Phase

The driver initiates execution phase of individual tasks by calling the `run()` method, which performs the core preprocessing 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.

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.

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 7

NaluWindUtils API Documentation

Core Utilities

CFDMesh

class `sierra::nalu::`**`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*)

Parameters

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

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

`~CFDMesh`()

void **`init`**()

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`**()

stk::mesh::MetaData &**`meta`**()

stk::mesh::BulkData &**bulk**()

stk::io::StkMeshIoBroker &**stkio**()

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

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

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.

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.

void **set_64bit_flags**()

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

void **set_write_flag**()
Force output of the results DB.

Interpolation utilities

struct `sierra::nalu::utils::`**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.

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.

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]`

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 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)

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*)

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

Pre-processing Utilities

PreProcessDriver

class sierra::nalu::**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.

PreProcessingTask

class sierra::nalu::**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::NDTW2D*, *sierra::nalu::RotateMesh*, *sierra::nalu::SamplingPlanes*

Public Functions

PreProcessingTask(*CFDMesh* &*mesh*)

Parameters

- `mesh`: A *CFDMesh* instance

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.

ABLFields

class `sierra::nalu::`**ABLFields**
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.

Inherits from *sierra::nalu::PreProcessingTask*

Public Functions

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.

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_)

int **ndim_**
Dimensionality of the mesh.

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

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

BdyIOPlanes

class `sierra::nalu::`**`BdyIOPlanes`**
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.

Inherits from *sierra::nalu::PreProcessingTask*

Public Functions

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.

SamplingPlanes

class `sierra::nalu::`**`SamplingPlanes`**
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.

Inherits from *sierra::nalu::PreProcessingTask*

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.

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.

RotateMesh

class `sierra::nalu::`**RotateMesh**
: 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]
```

Inherits from *sierra::nalu::PreProcessingTask*

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.

NDTW2D

class `sierra::nalu::`**NDTW2D**
: 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.

  Inherits from *sierra::nalu::PreProcessingTask*

Public Functions

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

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

Meshing Utilities

HexBlockMesh

class `sierra::nalu::`**HexBlockMesh**
: 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.

Part III

Indices and Tables

- genindex

Index

Symbols

A

B

C

D

E

T

V

W

Y