OOPS 1.1 Changes

May 2020

1 What's New?

OOPS 1.1 adds some new functionality in terms of the code itself, but it mostly features substantial improvements to the workflow and improves how `Parameters` are used. Here is a brief description of the most prominent changes.

- The `CMAKE_BUILD_TYPE` option now allows the user to select from a set of preset CMake defaults rather than typing them in.

- A new `BUILD_TESTS` option has been added. By default, this is set to `OFF`, which substantially improves compilation times by not compiling the OOPS unit tests.

- The `CMakeLists.txt` file in the OOPS project directory has been completely rewritten. Not only are target naming schemes more consistent, but the OOPS framework is now compiled to a library which is linked to new project targets through subdirectories. The OOPS build process now automatically searches all subdirectories in OOPS for additional `CMakeLists.txt` files and adds them to the build procedure. This means that all new projects now require their own `CMakeLists.txt` files in their root directory, which is *slightly* more involved than simply adding a new executable to the main `CMakeLists.txt` file. However, this makes maintaining OOPS much easier, and hopefully the other improvements to the build process mean that adding new projects is actually more straightforward than before.

- The `genParams.py` script has been slightly tweaked to make it easier to add parameter generation as part of the build process. The `genParams.py` script previously generated new files in the `scripts` directory with the script itself. Now, the script looks for `include` and `src` directories in the project directory where the JSON setup file is located and automatically places them there.

- The `ODE` class no longer maintains a pointer to a `Parameters` object, so the `ODE.getParameters()` and `ODE.setParameters()` functions have been removed. Due to changes in the OOPS architecture, it no longer makes sense to assume every `ODE` has a `Parameters` object. Every `ODE` subclass requiring `Parameters` objects should therefore explicitly declare and define a pointer, a getter, and a setter for the specific type of `Parameters` object required. While requiring slightly more work on the part of the user, this will hopefully make working with custom `Parameters` objects much more straightforward.

- The `SolverData` class is now a subclass of a new class, `ODEData`. `ODEData` is a simpler data structure which contains a reference to a grid and a single 2d array to solution data.

- The `ODE` class and the `Solver` class have been modified to support evolving only a subset of variables through the `evolutionIndices` member in the `ODE` class.

- The `ODE` class has added limited support for auxiliary fields via the `...AuxiliaryField()` methods. The multigrid infrastructure of OOPS makes it very difficult (although not impossible) to use these fields in concert with the `data` object, so users needing their auxiliary fields to behave somewhat like the evolved fields should consider increasing the number of variables in their `ODE` object and adjusting `evolutionIndices` instead. Some good examples for where auxiliary fields might be useful in their current form is maintaining an analytic solution for comparison, saving an old solution, and similar situations where the auxiliary fields are not needed directly in the `ODE.rhs()` method.

2 Migrating from OOPS 1.0 to 1.1

2.1 Updating the Build Process

The first mandatory change is that all projects now need to maintain their own `CMakeLists.txt` file. In other words, a basic project now needs an independent `CMakeLists.txt` file, a `include` directory, and a `src` directory. Projects using custom `Parameters` objects should also include a `scripts` directory containing the relevant JSON setup file(s). This new `CMakeLists.txt` file should look something like this:

```
cmake_minimum_required(VERSION 3.0)
project(Example)

set(EXAMPLE_INCLUDE_FILES
    include/example.h
  )
set(EXAMPLE_SOURCE_FILES
    src/example.cpp
    src/main.cpp
  )

set(SOURCE_FILES ${EXAMPLE_INCLUDE_FILES} ${EXAMPLE_SOURCE_FILES})
add_executable(Example ${SOURCE_FILES})
target_include_directories(Example PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/include)
target_include_directories(Example PRIVATE ${CMAKE_SOURCE_DIR}/include)
target_link_libraries(Example oops ${EXTRA_LIBS})
```

If the user is not working directy on the OOPS framework, no modifications should be necessary to the `CMakeLists.txt` file in the OOPS base directory. Consequently, changes to the framework should no longer create merge conflicts when pulling repository changes.

2.2 Updating `ODE` Objects using `Parameters`

The `ODE` class no longer has the `Parameters *params` member or the `Parameters* getParameters()` and `Result setParameters(Parameters*)` methods. We instead recommend that any `ODE` subclass dependent on these fields implement them explicitly. Therefore, if `CustomODE` depends on `CustomParameters`, it should have a `private` field `CustomParameters* params` and `public` functions `CustomParameters* getParameters()` and `void setParameters(CustomParameters *p)`. Therefore, accessing parameter data should no longer require an inelegant cast from `Parameters*` to `CustomParameters*`.

2.3 Running `genParams.py` at Build Time

One of the other beauties of the new build system is that individual projects can have special build instructions without cluttering up the main build file. As an example, consider the following CMake file from the `MultiGridTest` program:

```
cmake_minimum_required(VERSION 3.0)
project(MultiGrid)

# We only need to do any of this if test building is enabled.

if(NOT BUILD_TESTS)
  return()
endif()

# Generate the Parameters and ParamParser files.
set(PARAM_SRC ${CMAKE_CURRENT_SOURCE_DIR}/src/waveparser.cpp)
set(PARAM_INC
    ${CMAKE_CURRENT_SOURCE_DIR}/include/waveparameters.h
    ${CMAKE_CURRENT_SOURCE_DIR}/include/waveparser.h
   )
set(SETUP_SRC ${CMAKE_CURRENT_SOURCE_DIR}/scripts/wave.json)

add_custom_command(
  OUTPUT ${PARAM_INC}
         ${PARAM_SRC}
  DEPENDS ${SETUP_SRC}
  COMMAND ${PYTHON_EXECUTABLE} ${CMAKE_SOURCE_DIR}/scripts/↩
      genParams.py ${SETUP_SRC}
  COMMENT ”Generating custom Parameters files”
  WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
  VERBATIM USES_TERMINAL
)

set(MULTIGRID_INCLUDE_FILES
    include/firstorderwave.h
   )
set(MULTIGRID_SOURCE_FILES
    src/multiGridTest.cpp
    src/firstorderwave.cpp
   )

```

```
set(SOURCE_FILES ${MULTIGRID_INCLUDE_FILES} ${MULTIGRID_SOURCE_FILES} ${PARAM_INC} ${PARAM_SRC})
add_executable(MultiGridTest ${SOURCE_FILES})
target_include_directories(MultiGridTest PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/include)
target_include_directories(MultiGridTest PRIVATE ${CMAKE_SOURCE_DIR}/include)
target_link_libraries(MultiGridTest oops ${EXTRA_LIBS})
```

The `add_custom_command` section tells CMake that we want to run `genParams.py` on any files contained in `${SETUP_SRC}` (which is `wave.json` in this case) and expect the output files described by `${PARAM_SRC}` and `${PARAM_INC}`. These directories are then linked to `MultiGrid` in the `add_executable` command. The actual code in `add_custom_command` is completely generic, so it can be dropped into any `CMakeLists.txt` file which defines the expected input and output files. Best of all, because `${SETUP_SRC}` is declared as a dependency for `genParams.py`, it only runs if the output source files are missing or the input JSON setup scripts are changed.