CGNS-based mesh reading library for multi-block structured and unstructured grids

This is a C++ library that is capable of reading 2D structured and unstructured grids, which may or may not be distributed into different blocks (i.e. the grids may be represented as multi-block structured and unstructured grids).

Introduction

When it comes to reading a computational mesh, there are several file formats that offer a varying degree of support. Most are segregated into either structured or unstructured grids-only format, and thus can only be used for a subset of grids.

The CFD General Notation System (CGNS) format is a dedicated file format that can store information pertinent to CFD simulations, most importantly, the mesh itself. It has support for multi-block structured and unstructured grids, parallel file input and output, it can store arbitrary polyhedra as cell types, and it can store the CFD solution itself.

The CGNS file format makes use of HDF5, which efficiently stores the grid information in binary form. It is used in many scientific applications where data compression is required due to the size of the data stored. CFD is no different in that large file sizes for computational grids are common for industrial-scale problems. The HDF5 compression and storage routines have proven themselves to be the best choice in this case.

This library, then, provides a convenient wrapper around the CGNS library's API calls, to read both structured and unstructured grids. These are returned via a few getter functions that return information on the grid points, interfaces, and boundary conditions.

Installation

The library comes with dedicated build scripts that compile and install the library into either a shared or static library. Different installation scripts are provided for different platforms.

All source code is provided within the `meshReaderLib` folder. In order to test that the library is working correctly, the compilation can be tested with a few tests that are provided in the `tests` folder. Within that folder, the file `mainTest.cpp` collects all tests, which in turn are provided in the `tests/unit` folder (i.e. it contains all unit tests). Running the executable that is generated by the `tests/mainTest.cpp` file will verify that the compilation has worked correctly.

In order for the library to be compiled correctly, a few libraries need to be compiled beforehand, namely the CGNS, HDF5, and gtest library. Below is a collection of links, together with build scripts that automatically build these libraries on either Windows or UNIX:

- Install CGNS and HDF5
- Install gtest

Once these libraries are installed and available, you can go ahead and use one of the build scripts discussed below to compile the mesh reading library on either Windows or UNIX.

Windows

On Windows, we can either build the mesh reading library all into a single executable (library + tests), or into separate test code and either static or dynamic library. The build scripts to achieve that are given below. All instructions assume that you are using a developer PowerShell, which is automatically installed with Visual Studio.

> [!NOTE]
> In order to execute PowerShell scripts, you will have to allow to run remotely signed scrips first (see the first example). You may also need to unblock the PowerShell files first before using them. You will also need to have Visual Studio with the C++ development kit installed and run this scrupt inside a Developer PowerShell console. You may also need to activate 64-bit compilation, which may cause issues if the compilation tries to mix 32-bit and 64-bit binaries. If this sounds too complicated, simply install WSL and follow the (simplified) instructions for UNIX below.

Single executable (tests + library)

`.\buildAndRun.ps1`

Tests with separate static mesh reader library

`.\buildStaticAndRun.ps1`

Tests with separate dynamic mesh reader library

`.\buildDynamicAndRun.ps1`

UNIX (macOS, Ubuntu, etc.)

Similar to the instruction given on Windows, we can run the build bash script to check that the compilation and execution have worked correctly. These scripts assume that you have the CGNS and HDF5 library compiled and installed within your home directory, i.e. within `~/libs`.

Unlike Windows, we can run scripts directly but sometimes permissions may not be granted for the current user, in which case we can use the `chmod` command to grant permission to run a specific file as `chmod +x buildAndRun.sh`, for example. We can also grant permissions for all bash scripts using regular expressions with `chmod +x *.sh`.

To build the tests and library, open a terminal and type in one of the following commands depending on what you want to build

Single executable (tests + library)

`./buildAndRun.sh`

Tests with separate static mesh reader library

`./buildStaticAndRun.sh`

Tests with separate dynamic mesh reader library

`./buildDynamicAndRun.sh`

Usage

In this section, we will look at a few basic examples of how to read different mesh properties.

Reading structured grid information

In this section, we will see how we can read multi-block structured grid information from the library within the `meshReaderLib` folder. In all cases, we need to first create a structured grid instance and then read the mesh. This is achieved with the following function calls:

```
#include <filesystem>
#include "meshReaderLib/meshReader.hpp"

int main() {
  // provide a relative or absolute path to where the CGNS file is located that should be read
  auto structuredMeshFile = std::filesystem::path("path/to/structured/mesh.cgns");

  // create an instance of the structured mesh reading class
  ReadStructuredMesh structuredMesh(structuredMeshFile);

  // read all structured grid properties at once
  structuredMesh.readMesh();

  // perform any other function calls here to get grid properties, discussed below in detail
  // ...

  return 0;
}
```

After the structured mesh has been set up in this way, we can proceed to read specific mesh information.

Reading coordinate information

To read the coordinates of a structured grid, we call the dedicated getter function. An example of how to loop over the mesh is provided as well, as well as how to get the `x` and `y` coordinates.

```
// get the coordinates from the structured mesh
auto coordinates = structuredMesh.getCoordinates();

// get the number of grid blocks (could be a multi-block structured grid)
auto numberOfBlocks = coordinates.size();

// loop over all blocks
for (unsigned block = 0; block < numberOfBlocks; ++block) {
  // get the number of x-coordinate
  auto numberOfIndicesInX = coordinates[block].size();

  // loop over all x-coordinates
  for (unsigned indexX = 0; indexX < numberOfIndicesInX; ++indexX) {
    // repeat the same for the y-coordinate
    auto numberOfIndicesInY =  coordinates[block][indexX].size();

    // loop over all y-coordinates
    for (unsigned indexY = 0; indexY < numberOfIndicesInY; ++indexY) {
      const auto &x = coordinates[block][indexX][indexY][COORDINATE::X];
      const auto &y = coordinates[block][indexX][indexY][COORDINATE::Y];

      // process results and use x and y as required
      // ...

    }
  }
}
```

Reading interface connectivity information

If we have more than a single zone/block in our mesh, then we need to know at which interface these blocks are connected, so that we can request the correct neighboring vertices in our calculation later, where we update values close the the interface and where the computational stencil requires information from a neighboring block. The code below shows how to read this information.

```
// get the interface connectivity from the structured mesh
auto interfaceConnectivity = structuredMesh.getInterfaceConnectivity();

// get the number of interfaces in the mesh
auto numberOfInterfaces = interfaceConnectivity.size();

// loop over all interfaces
for (unsigned interface = 0; interface < numberOfInterfaces; ++interface) {
  auto ownerBlock = interfaceConnectivity[interface].zones[0];
  auto neighbourBlock = interfaceConnectivity[interface].zones[1];

  // get the number of vertices in interface
  auto numberOfVerticesInInterface = interfaceConnectivity[interface].ownerIndex.size();

  // loop over interface
  for (int vertex = 0; vertex < numberOfVerticesInInterface; ++vertex) {
    // get owner indices in x and y (i.e. indices i, j)
    const auto &owner_i = interfaceConnectivity[interface].ownerIndex[vertex][COORDINATE::X];
    const auto &owner_j = interfaceConnectivity[interface].ownerIndex[vertex][COORDINATE::Y];

    // get neighbour indices in x and y (i.e. indices i, j)
    const auto &neighbour_i = interfaceConnectivity[interface].neighbourIndex[vertex][COORDINATE::X];
    const auto &neighbour_j = interfaceConnectivity[interface].neighbourIndex[vertex][COORDINATE::Y];

    // use vertices here, for example, discretising a second order partial derivative d2u/dx2 as
    // u^new = (u[i           + 1] - 2 * u[i]       + u[i       - 1]) / (dx * dx), may be written as
    // u^new = (u[neighbour_i + 1] - 2 * u[owner_i] + u[owner_i - 1]) / (dx * dx)
    // ...
  }
}
```

Reading boundary condition information

Finally, we want to apply specific boundary conditions at domain boundaries. For this, we first need to know which boundary condition is prescribed at the boundary. To read boundary information, use the code below as a starting point.

```
// get the boundary condition information
auto boundaryConditions = structuredMesh.getBoundaryConditions();

// get the number of blocks in the mesh
auto numberOfBlocks = boundaryConditions.size();

// loop over all blocks
for (unsigned block = 0; block < numberOfBlocks; ++block) {
  // check how many boundaries are within each block
  auto numberOfBoundaries = boundaryConditions[block].size();

  // loop over all boundaries in current block
  for (unsigned boundary = 0; boundary < numberOfBoundaries; ++boundary) {
    // get the boundary type, can be of type BC::WALL, BC::SYMMETRY, BC::INLET, or BC::OUTLET
    auto bcType = boundaryConditions[block][boundary].boundaryType;

    // get the number of vertices in boundary
    auto numberOfVertices = boundaryConditions[block][boundary].indices.size();

    // loop over all vertices in boundary
    for (unsigned vertex = 0; vertex < numberOfVertices; ++vertex) {
      // get current vertex
      const auto bcVertex = boundaryConditions[block][boundary].indices[vertex];

      // apply boundary condition based on specified type
      if (bcType == BC::WALL) {
        // process wall ...
      } else if (bcType == BC::SYMMETRY) {
        // process symmetry ...
      } else if (bcType == BC::INLET) {
        // process inlet ...
      } else if (bcType == BC::OUTLET) {
        // process outlet ...
      }
    }
  }
}
```

Reading unstructured grid information

As we saw for the structured grid, we need to first set up an instance of the unstructured mesh reading class and read the mesh itself. Afterwards, we have access to the mesh information via getters just as with the structured grid. The information is now in an unstructured grid format and thus different from the structured grid information. Additional information on the vertex connectivity is available as well and the examples shown below indicate how to read an unstructured grid with all of its information. But first, we have to set up the unstructured grid as mentioned above.

```
#include <filesystem>
#include "meshReaderLib/meshReader.hpp"

int main() {
  // provide a relative or absolute path to where the CGNS file is located that should be read
  auto unstructuredMeshFile = std::filesystem::path("path/to/unstructured/mesh.cgns");

  // create an instance of the unstructured mesh reading class
  ReadUnstructuredMesh unstructuredMesh(unstructuredMeshFile);

  // read all unstructured grid properties at once
  unstructuredMesh.readMesh();

  // perform any other function calls here to get grid properties, discussed below in detail
  // ...

  return 0;
}
```

Reading coordinate (vertex) information

The unstructured grid exposes the `getCoordinates()` function which provides the `x` and `y` coordinates for each vertex in the unstructured grid. Since the unstructured grid is allowed to be split into several zones/blocks, we need to loop over the number of blocks first, just as we did with our structured grid as well. A complete example of accessing the `x` and `y` coordinates for an unstructured grid is shown below.

```
// get the coordinates from the unstructured mesh
auto coordinates = unstructuredMesh.getCoordinates();

// get the number of grid blocks (could be a multi-block unstructured grid)
auto numberOfBlocks = coordinates.size();

// loop over all blocks
for (unsigned block = 0; block < numberOfBlocks; ++block) {
  // get the number of vertices
  auto numberOfVertices = coordinates[block].size();

  // loop over all vertices
  for (unsigned vertex = 0; vertex < numberOfVertices; ++vertex) {
    const auto &x = coordinates[block][vertex][COORDINATE::X];
    const auto &y = coordinates[block][vertex][COORDINATE::Y];

    // process results and use x and y as required
    // ...

  }
}
```

Reading element connectivity table

As alluded to above, unstructured grids require additional element connectivity information, which specifies how vertices are connected into a cell/element. The unstructured grid class has the `getInternalCells()` function available for this, which exposes, for each cell/element the vertex ID that makes up the current cell. We can then get the location for each vertex in the cell by looking up the information in the coordinate array that we read in the previous code example.

```
// get the internal elements from the unstructured mesh
auto interalElements = unstructuredMesh.getInternalCells();

// get the number of grid blocks (could be a multi-block unstructured grid)
auto numberOfBlocks = interalElements.size();

// loop over all blocks
for (unsigned block = 0; block < numberOfBlocks; ++block) {
  // get the number of internal elements in current block
  auto numberOfVertices = interalElements[block].size();

  // loop over all elements
  for (unsigned element = 0; element < numberOfVertices; ++element) {
    // get number of vertices making up current element
    auto numberOfVertices = interalElements[block][element].size();

    if (numberOfVertices == 3) {
      auto v0 = interalElements[block][element][0];
      auto v1 = interalElements[block][element][1];
      auto v2 = interalElements[block][element][2];

      std::cout << "triangle with vertices: " << v0 << ", " << v1 << ", " << v2 << std::endl;

      std::cout << "triangle has vertices at: ";

      std::cout << "(" << coordinates[block][v0][COORDINATE::X] << ", ";
      std::cout << coordinates[block][v0][COORDINATE::Y] << "), ";

      std::cout << "(" << coordinates[block][v1][COORDINATE::X] << ", ";
      std::cout << coordinates[block][v1][COORDINATE::Y] << "), ";

      std::cout << "(" << coordinates[block][v2][COORDINATE::X] << ", ";
      std::cout << coordinates[block][v2][COORDINATE::Y] << "), ";
    }

    // process results and use x and y as required
    // ...

  }
}
```

Reading interface connectivity information

Similar to the structured grid, we may have several zones or blocks in our grid. At the interface between two zones/blocks, we need to know which faces of the unstructured cells are connected to each other so that we can request information from neighboring cells on different zones/blocks for the calculation of flow properties. An example to obtain the matching vertex pairs that make up a face is shown below.

```
// get the interface connectivity from the unstructured mesh
auto interfaceConnectivity = unstructuredMesh.getInterfaceConnectivity();

// get the number of blocks in the mesh
auto numberOfBlocks = interfaceConnectivity.size();

// loop over all interfaces
for (unsigned block = 0; block < numberOfBlocks; ++block) {
  // get the number of interfaces in current block
  auto numberOfInterfaces = interfaceConnectivity[block].size();

  // loop over all interfaces in current block
  for (unsigned interface = 0; interface < numberOfInterfaces; ++interface) {
    auto ownerBlock = interfaceConnectivity[block][interface].zones[0];
    auto neighbourBlock = interfaceConnectivity[block][interface].zones[1];

    // get the number of vertices in interface
    auto numberOfFacesInInterface = interfaceConnectivity[block][interface].ownerIndex.size();

    // loop over faces in interface
    for (int face = 0; face < numberOfFacesInInterface; ++face) {
      // get the starting vertex of the owning face
      const auto &ownerFaceStart = interfaceConnectivity[block][interface].ownerIndex[face][0];

      // get the ending vertex of the owning face
      const auto &ownerFaceEnd = interfaceConnectivity[block][interface].ownerIndex[face][1];

      // get the starting vertex of the neighbour face
      const auto &neighbourFaceStart = interfaceConnectivity[block][interface].neighbourIndex[face][0]

      // get the ending vertex of the neighbour face
      const auto &neighbourFaceEnd = interfaceConnectivity[block][interface].neighbourIndex[face][1];
    }
  }
}
```

Reading boundary condition information

Getting boundary conditions is also no different from reading boundary conditions on a structured grid. The only difference here is that instead of reading vertices that are on the boundary, we read vertex pairs that make up the face of the boundary. The example below shows how to extract the type of boundary, as well as the faces that are contained within that boundary patch.

```
// get the boundary condition information
auto boundaryConditions = unstructuredMesh.getBoundaryConditions();

// get the number of blocks in the mesh
auto numberOfBlocks = boundaryConditions.size();

// loop over all blocks
for (unsigned block = 0; block < numberOfBlocks; ++block) {
  // check how many boundaries are within each block
  auto numberOfBoundaries = boundaryConditions[block].size();

  // loop over all boundaries in current block
  for (unsigned boundary = 0; boundary < numberOfBoundaries; ++boundary) {
    // get the boundary type, can be of type BC::WALL, BC::SYMMETRY, BC::INLET, or BC::OUTLET
    auto bcType = boundaryConditions[block][boundary].boundaryType;

    // get the number of faces in boundary
    auto numberOfFaces = boundaryConditions[block][boundary].indices.size();

    // loop over all faces in boundary
    for (unsigned face = 0; face < numberOfFaces; ++face) {
      // get current starting location of boundary face
      const auto bcFaceStart = boundaryConditions[block][boundary].indices[face][0];

      // get current ending location of boundary face
      const auto bcFaceEnd = boundaryConditions[block][boundary].indices[face][1];

      // apply boundary condition based on specified type
      if (bcType == BC::WALL) {
        // process wall ...
      } else if (bcType == BC::SYMMETRY) {
        // process symmetry ...
      } else if (bcType == BC::INLET) {
        // process inlet ...
      } else if (bcType == BC::OUTLET) {
        // process outlet ...
      }
    }
  }
}
```