h5geo

Release 0.2

kerim khemraev

Jun 27, 2022

CONTENTS

To see C++ API documentation please refer to tierra-colada.github.io/h5geo/

CHAPTER

ONE

ABOUT

C++17 and python API to work with geo-data (seismic, wells, maps, other in process) based on HDF5. Aimed at geoscientists and developers.

1.1 General

In geoscience there are different types of data and many file formats for storing these data. Also the data belong to some spatial reference system. In the same time the developper or scientist prefer not to use many libraries when working with data (i.e. reading, converting spatial reference system and units). All we want is convenient API, fast execution and availability in python. That's what **h5geo** is aimed at.

Being written in C++ it uses pybind11 to make python bindings, HDF5 for storing data, GDAL for spatial reference conversion, units for units conversion and Eigen as container for storing numerics in memory.

For now **h5geo** provides API to work with:

- seismic
- wells
  - deviations
  - logs
  - welltops
- maps
- points

Originally the project was started as part of a free and open source geophysical platform Colada.

1.2 License

MIT

CHAPTER

TWO

BUILD INSTRUCTIONS

2.1 Pre-requisites

- CMake
- compiler with C++17 support
- python (optional)

2.2 Dependencies

- HDF5
- units
- h5gt (header only)
- Eigen (header only)
- magic_enum (header only)
- mio (header only)
- TBB (optional)
- GDAL with PROJ support (optional)

2.3 Build using prebuilt dependencies

```
git clone https://github.com/tierra-colada/h5geo.git
cd h5geo
mkdir build
cd build
cmake ..
  # find package dirs
  -DEigen3_ROOT:PATH=/path/to/eigen
  -Dmio_ROOT:PATH=/path/to/mio
  -DZLIB_ROOT:PATH=/path/to/zlib
  -DHDF5_ROOT:PATH=/path/to/hdf5
  -Dh5gt_ROOT:PATH=/path/to/h5gt
  -Dmagic_enum_ROOT:PATH=/path/to/magic_enum
```

(continues on next page)

(continued from previous page)

```
  -Dunits_DIR:PATH=$/path/to/units/lib/cmake/units
  -DTBB_ROOT:PATH=/path/to/tbb
  -DGDAL_ROOT:PATH=/path/to/gdal
  -DPYTHON_EXECUTABLE=/path/to/python.exe
  # Lib settings
  -DH5GEO_SUPERBUILD:BOOL=OFF
  -DH5GEO_USE_THREADS:BOOL=ON
  -DH5GEO_USE_GDAL:BOOL=ON
  -DH5GEO_BUILD_SHARED_LIBS:BOOL=ON
  -DH5GEO_BUILD_TESTS:BOOL=OFF
  -DH5GEO_BUILD_h5geopy:BOOL=ON
  # to 'import h5geopy' to python runtime deps should be resolved
  -DH5GEOPY_RESOLVE_RUNTIME_DEPS:BOOL=ON
  -DH5GEOPY_COPY_RUNTIME_DEPS:BOOL=OFF
  # path to runtimes (used at h5geopy installation step)
  -DHDF5_RUNTIME_DIRS:PATH=/path/to/hdf5/bin
  -DZLIB_RUNTIME_DIRS:PATH=/path/to/zlib/bin
  -DUNITS_RUNTIME_DIRS:PATH=/path/to/units/bin
  -DTBB_RUNTIME_DIRS:PATH=/path/to/tbb/bin
  -DGDAL_RUNTIME_DIRS:PATH=/path/to/tbb/bin
  -DH5GEO_RUNTIME_DIRS:PATH=/path/to/h5geo/bin
  # CMake settings
  -DCMAKE_BUILD_TYPE=Release
  -G "some generator"
cmake --build . --config Release
cmake --install . --prefix /path/to/h5geo-install
```

Note: If you see that some of the dependencies were not resolved at installation step then change `<lib>_RUNTIME_DIRS`, rerun cmake & cmake install steps.

Warning: `H5GEOPY_COPY_RUNTIME_DEPS` copies all the resolved runtimes to `site-packages/h5geopy` dir. There might be OS-specific runtimes that nobody wants to copy. Thus this option is not recommended yet.

2.4 Build using superbuild

```
git clone https://github.com/tierra-colada/h5geo.git
cd h5geo
mkdir build
cd build
cmake ..
  -DCMAKE_INSTALL_PREFIX=/path/to/h5geo-install
  -DCMAKE_BUILD_TYPE=Release
  -DH5GEO_SUPERBUILD=ON
  -DH5GEO_BUILD_h5geopy=ON
  -DH5GEO_USE_THREADS=ON
  -DH5GEO_BUILD_SHARED_LIBS=ON
  -DH5GEO_USE_GDAL=ON
```

(continues on next page)

(continued from previous page)

```
  -DH5GEO_BUILD_TESTS=OFF
  -DCOPY_H5GTPY_RUNTIME_DEPS=OFF
  -DRESOLVE_H5GTPY_RUNTIME_DEPS=ON
  -DH5GEOPY_COPY_RUNTIME_DEPS=OFF
  -DH5GEOPY_RESOLVE_RUNTIME_DEPS=ON
  -DPYTHON_EXECUTABLE=/path/to/python.exe
```

Note: `h5gtpy` and `GDAL` (`osgeo`) will be installed in `site-packages` as well.

2.5 Build using pip

```
pip install git+https://github.com/tierra-colada/h5geo.git@<tag> --verbose
```

where <tag> is git-tag.

Note: `h5gtpy` and `GDAL` (`osgeo`) will be installed in `site-packages` as well.

`h5geo` dependencies will be installed in `site-packages/h5geopy.<some_postfix>` dir.

Warning: No prebuilt wheels is supplied yet. `pip install` simply runs superbuild. Thus installation takes pretty much time. The `verbose` option helps you to see the build progress and catch errors if any.

2.6 Supported platforms

Windows, Linux, MacOS

CHAPTER

THREE

API & BASIC USAGE

3.1 Description

h5geo follows interface strategy. That means API is exposed through implementation classes (suffixed with `Impl`). Also there is no public constructors. To instantiate a class one needs to use factory function.

The most basic class is `H5Base`. It can't be instantiated explicitely.

The classes `H5BaseContainer` and `H5BaseObject` are direct successors of `H5Base`. The main difference between them is that `H5BaseContainer` is built around `h5gt::File` while `H5BaseObject` explores `h5gt::Group`. Thus container (HDF5 file) may store many geo-objects (HDF5 objects).

Next in the hierarchy follows more specific classes: `H5SeisContainer/H5Seis`, `H5MapContainer/H5Map`, `H5WellContainer/H5Well`, `H5DevCurve`, `H5LogCurve`, `H5BasePoints`, `H5Points1` etc.

Note: A designated container may store only appropriated geo-objects. In that sense you can't store `H5Map` in `H5SeisContaner` for example. But as you can see there is no container for `H5Points`. That means you are free to create and store points in any container.

Warning: **h5geo** works with **column-major** Eigen matrices only (the default Eigen storage order)!

3.2 Usage

3.2.1 containers

To create a container one should use one the following factory functions:

```
H5BaseContainer* createContainer(
   h5gt::File h5File, h5geo::ContainerType cntType, h5geo::CreationType createFlag);

H5BaseContainer* createContainerByName(
   std::string& fileName, h5geo::ContainerType cntType, h5geo::CreationType createFlag);
```

Using `dynamic_cast<>` one is able to cast returned container to the one defined with `h5geo::ContainerType` argument.

Note: There are helper functions to create specific containers: `h5geo::createMapContainer`, `h5geo::createMapContainerByName` that implicitely cast returned type to `H5Map` (most of geo-objects have such functions).

And to open a container:

```
H5BaseContainer* openContainer(
   h5gt::File h5File);

H5BaseContainer* openContainerByName(
   const std::string& fileName);
```

Note: As when creating object you may want to use helper functions like: `h5geo::openMapContainer`, `h5geo::openMapContainerByName` that implicitely cast returned type to `H5Map` (most of geo-objects have such functions).

3.2.2 geo-objects

All geo-objects are represented by `h5gt::Group` with assotiated and nested DataSets, Groups and Attributes.

To create a geo-object one must have an instance of an appropriate container (or parent geo-object: for example `H5DevCurve` and `H5LogCurve` have `H5Well` as parent). Thus to create `H5Well` one must have instance of `H5WellContainer` and use one of the following method:

```
H5Well* H5WellContainer::createWell(
   std::string& name,
   WellParam& p,
   h5geo::CreationType createFlag)

H5Well* H5WellContainer::createWell(
   h5gt::Group group,
   WellParam& p,
   h5geo::CreationType createFlag)
```

`WellParam` inherites `BaseObjectParam` and defines parameters like: spatial reference, length/temporal/angular/data units, null value, uwi, kelly bushing and well head coordinates.

Note: Most geo-objects may be created/opened with name or `h5gt::Group` object.

To open a geo-object one may use parent object instance:

```
H5Well* H5WellContainer::openWell(
   const std::string& name);
H5Well* H5WellContainer::openWell(
   h5gt::Group group);
```

Note: There are helper functions to open them without having parent object: `h5geo::openWell`, `h5geo::openWellByName`

Or more generally (use it in pair with `dynamic_cast<>`): `h5geo::openObject`, `h5geo::openObjectByName`

3.2.3 pointers

Currently only unique pointers are provided. They are named in the following manner: `H5WellCnt_ptr` and `H5Well_ptr`.

The preffered way to create objects:

```
#include <iostream>
#include <h5geo/h5wellcontainer.h>
#include <h5geo/h5well.h>

int main(){
   std::string fileName = "wells.h5";
   H5WellCnt_ptr wellCnt(h5geo::createWellContainerByName(
      fileName, h5geo::CreationType::OPEN_OR_CREATE));

   if (!wellCnt){
      std::cout << "Unable to open or create well container" << std::endl;
      return -1;
   }

   WellParam p;
   p.headX = 444363;
   p.headY = 7425880;
   p.kb = 50.88;
   p.uwi = "my_uwi";
   p.lengthUnits = "meter";

   std::string wellName = "myWell";
   H5Well_ptr well(wellCnt->createWell(
      wellName, p, h5geo::CreationType::OPEN_OR_CREATE));
   if (!well){
      std::cout << "Unable to open or create well" << std::endl;
      return -1;
   }

   return 0;
}
```

3.2.4 units & spatial reference

units

All geo objects have spatial reference and length/temporal/angular/data units. Not all them may be used by geo-object but the idea is: *a geo-object must match to the units*.

That means if for example `H5Well` has length units `meter` then all length units must be given in meters (header coordinates as well as kelly bushing for example). The same also concerns temporal, angular and data units.

Data units is units of Z-axis of `H5Map` object for example. Or units of `H5Seis` traces (maybe `psi` in case of marine seismic).

Every geo-object provides API to automatically convert units. For example when writing data to `H5Map` one is free to specify the data units that the data currently is in: `bool H5Map::writeData(Eigen::Ref<Eigen::MatrixXd> M, "m")`

The data will be converted from `m` to `H5Map::getDataUnits`.

And one needs to get data in some units: `Eigen::MatrixXd H5Map::getData("cm")`

Then the conversion is done in reverse order: from `H5Map::getDataUnits` to `cm`.

Note: No conversion is done if no units were specified.

Sometimes it is impossible to predict what units are going to be used. For example when working with seismic trace headers the API provides two arguments: `unitsFrom` and `unitsTo`. The conversion is done in direct order: `unitsFrom` will be converted to `unitsTo`.

One can check if the units are convertable through the web-service.

spatial reference

Spatial reference is given from `PROJ-install/share/proj/proj.db`. In `projected_crs` table find `auth_name` and `code` columns. Usually the spatial reference is shaped as: `auth_name:code`.

An example: `EPSG:32056`.

Basically OGRSpatialReference::SetFromUserInput function is used to create spatial reference object.

If you work with many objects that belong to different spatial reference you may want to set a spatial reference for the session and pass `doCoordTransform` as `true` when writing/getting the data. Take a look at h5core_sr_settings.h.

CHAPTER

FOUR

TUTORIAL

SUKA

As **h5geo** consists of several independent geo-types then the tutorial is splitted in several parts.

4.1 Maps

Map is an object that is represented by a matrix with coordinates of **upper-left** matrix corner (origin `X0`, `Y0`), coordinates of **upper-right** matrix corner (point1 `X1`, `Y1`) and coordinates of **lower-left** matrix corner (point2 `X2`, `Y2`). Thus only single `Z` value corresponds for each `XY` point.

4.1.1 Create Map

Define map parameters for column-major Eigen matrix of size `[20, 10]` whose Z-values represent time (`ms`) and the coordinates are given in `millimeters`.

```
#include <iostream>
#include <h5geo/h5mapcontainer.h>
#include <h5geo/h5map.h>


int main(){
   std::string fileName = "maps.h5";
   H5MapCnt_ptr cnt(h5geo::createMapContainerByName(
      fileName, h5geo::CreationType::OPEN_OR_CREATE));
   if (!cnt){
      std::cout << "Unable to open or create map container" << std::endl;
      return -1;
   }

   MapParam p;
   p.nX = 10;
   p.nY = 20;
   p.X0 = 0;
   p.Y0 = 0;
   p.X1 = 100;
   p.Y1 = 0;
   p.X2 = 0;
   p.Y2 = 100;
```

(continues on next page)

(continued from previous page)

```
   p.lengthUnits = "millimeter";
   p.dataUnits = "ms";
   p.xChunkSize = 5;
   p.yChunkSize = 5;
   p.spatialReference = "EPSG:8139";

   std::string mapName = "myMap";
   H5Map_ptr map(cnt->createMap(
      mapName, p, h5geo::CreationType::OPEN_OR_CREATE));
   if (!map){
      std::cout << "Unable to open or create map" << std::endl;
      return -1;
   }

   return 0;
}
```

4.1.2 Write/Read data

Let's suppose that we have data in `sec` and we want to write it. To be sure that **h5geo** is able to convert `sec` to `ms` we can use the web-service.

```
Eigen::MatrixXd m = Eigen::MatrixXd::Random(p.nY, p.nX);
if (!map->writeData(m, "sec")){
   std::cout << "Unable to write data" << std::endl;
   return -1;
}
```

Now we want to read the data in `sec`:

```
Eigen::MatrixXd M = map->getData("sec");
if (M.size() < 1){
   std::cout << "Unable to read data" << std::endl;
   return -1;
}
```

4.1.3 Read data from GDAL supported file

There are useful methods aimed at reading files using GDAL API.

Before reading one should call *GDALAllRegister()* defined in *gdal.h* (the same also concerns when using **h5geopy**).

```
// don't forget to initialize GDAL readers first (maybe at application initialization␣
↪time)
GDALAllRegister();

if (!map->readRasterCoordinates("data_file.zmap", "meter")){
   std::cout << "Unable to read coordinates from raster data" << std::endl;
   return -1;
}
```

(continues on next page)

(continued from previous page)

```
if (!map->readRasterSpatialReference("data_file.zmap")){
   std::cout << "Unable to read spatial reference from raster data" << std::endl;
   return -1;
}
if (!map->readRasterLengthUnits("data_file.zmap")){
   std::cout << "Unable to read length from raster data" << std::endl;
   return -1;
}
if (!map->readRasterData("data_file.zmap")){
   std::cout << "Unable to read data from raster data" << std::endl;
   return -1;
}
```

4.1.4 Working with attribute map

Let's suppose the created time-domain map has velocity attribute i.e. we have somehow sliced volume of velocities and kept the data. Lets generate such attribute map first:

```
MapParam p_attrMap = p;
p_attrMap.dataUnits = "feet/s"

std::string attrMapName = "myAttrMap";
H5Map_ptr attrMap(cnt->createMap(
   attrMapName, p_attrMap, h5geo::CreationType::OPEN_OR_CREATE));
if (!attrMap){
   std::cout << "Unable to open or create attribute map" << std::endl;
   return -1;
}

Eigen::MatrixXd v = Eigen::MatrixXd::Random(p_attrMap.nY, p_attrMap.nX);
if (!attrMap->writeData(v, "km/ms")){
   std::cout << "Unable to write data" << std::endl;
   return -1;
}
```

To add attribute map:

```
// addAttributeMap returns std::optional<h5gt::Group> of created map
if (!map->addAttributeMap(attrMap, "velocity").has_value()){
   std::cout << "Unable to add attribute map" << std::endl;
}
```

Then we can open the attribute and work with it as with usual map:

```
H5Map_ptr velocityMap(map->openAttributeMap("velocity"));
if (!velocityMap){
   std::cout << "Unable to open attribute map" < std::endl;
}
```

Finally to remove attribute map we can call the following method:

```
if (!map->removeAttributeMap("velocity")){
   std::cout << "Unable to remove attribute map" << std::endl;
}
```

Note: Attribute map is simply HDF5 soft link within `H5Map` object.

4.2 Wells

Well is an object that is responsible for well head coordinates, kelly bushing and manages deviations and log curves.

4.2.1 Create Well

We define well head coordinates, kelly bushing and uwi:

```
#include <iostream>
#include <h5geo/h5wellcontainer.h>
#include <h5geo/h5well.h>
#include <h5geo/h5devcurve.h>
#include <h5geo/h5logcurve.h>


int main(){
   std::string fileName = "wells.h5";
   H5WellCnt_ptr cnt(h5geo::createWellContainerByName(
      fileName, h5geo::CreationType::OPEN_OR_CREATE));
   if (!cnt){
      std::cout << "Unable to open or create well container" << std::endl;
      return -1;
   }

   WellParam p;
   p.headX = 444363;
   p.headY = 7425880;
   p.kb = 50.88;
   p.uwi = "my_uwi";
   p.lengthUnits = "meter";

   std::string wellName = "myWell";
   H5Well_ptr well(cnt->createWell(
      wellName, p, h5geo::CreationType::OPEN_OR_CREATE));
   if (!well){
      std::cout << "Unable to open or create well" << std::endl;
      return -1;
   }

   return 0;
}
```

4.2.2 Open well by uwi

The straight forward way to open well is by its name. Another way is by uwi (slower but useful):

```
H5Well_ptr wellByUwi(cnt->openWellByUWI("my_uwi"));
if (!wellByUwi || !well->isEqual(wellByUwi.get())){
   std::cout << "Unable to open well by UWI" << std::endl;
   return -1;
}
```

4.2.3 Create Dev Curve

Dev curve is responsible for trajectory storing and transformations using minimum curvature method implemented in h5deviation.h5.

```
DevCurveParam p_devCurve;
p_devCurve.lengthUnits = "meter";
p_devCurve.temporalUnits = "millisecond";
p_devCurve.angularUnits = "degree";

std::string devCurveName = "myDevCurve";
H5DevCurve_ptr devCurve(
   well->createDevCurve(
      devCurveName, p_devCurve, h5geo::CreationType::OPEN_OR_CREATE));
if (!devCurve){
   std::cout << "Unable to open or create dev curve" << std::endl;
   return -1;
}
```

4.2.4 Write/Read Dev Curve

`H5DevCurve` stores the following curves: `MD,AZIM,INCL,DX,DY,TVD,OWT`. All other curves are calculated based on these curves. That is done to prevent calculation errors i.e. everytime `MD_AZIM_INCL` is transformed to `X_Y_TVD` an error is accumulating. The same concerns when doing that in backward order: `X_Y_TVD` to `MD_AZIM_INCL`.

```
Eigen::VectorXd dx(3), dy(3), tvd(3);
dx << 0, 3, 5;
dy << 0, 0.3, 0.5;
tvd << 0, 1, 2;
if (!devCurve->writeDX(dx, "m") ||
    !devCurve->writeDY(dy, "m") ||
    !devCurve->writeTVD(tvd, "m")){
   std::cout << "Unable to write DX, DY, TVD" << std::endl;
   return -1;
}

// update is needed to calculate MD, AZIM, INCL based on DX, DY, TVD
devCurve->updateMdAzimIncl();
```

To get back data:

```
Eigen::VectorXd tvdss_out = devCurve->getCurve("TVDSS", "km");
if (tvdss_out.size() < 1){
   std::cout << "Unable to get TVDSS" << std::endl;
   return -1;
}
```

H5Well also provides API to work with active deviation. To set current deviation to be active:

```
if (!well->setActiveDevCurve(devCurve.get())){
   std::cout << "Unable to set active dev curve" << std::endl;
   return -1;
}

// or simply: devCurve->setActive();
```

To check is dev curve is active use H5DevCurve::isActive().

Note: Active dev curve is simply soft link to the real dev curve within HDF5 file.

4.2.5 Create Log Curve

Log curve is represented by [N, 2] matrix. The first column is MD and the second is VAL (value).

```
LogCurveParam p_logCurve;
p_logCurve.lengthUnits = "meter";
p_logCurve.dataUnits = "kg/m2";

std::string logCurveName = "myLogCurve";
H5LogCurve_ptr logCurve(
   well->createLogCurve(
      logCurveName, p_logCurve, h5geo::CreationType::OPEN_OR_CREATE));
if (!logCurve){
   std::cout << "Unable to open or create log curve" << std::endl;
   return -1;
}
```

4.2.6 Write/Read Log Curve

Write MD and VAL:

```
Eigen::VectorXd md(3), vals(3);
md << 0, 3, 5;
vals << 500, 700, 800;
if (!logCurve->writeCurve(h5geo::LogDataType::MD, md) ||
    !logCurve->writeCurve(h5geo::LogDataType::VAL, vals)){
   std::cout << "Unable to write MD and VALS" << std::endl;
   return -1;
}
```

And to read data:

```
Eigen::VectorXd md_out = logCurve->getCurve(h5geo::LogDataType::MD, "cm");
if (md_out.size() < 1){
   std::cout << "Unable to get MD" << std::endl;
   return -1;
}

Eigen::VectorXd vals_out = logCurve->getCurve("VAL", "g/cm2");
if (vals_out.size() < 1){
   std::cout << "Unable to get VAL" << std::endl;
   return -1;
}
```

4.3 Seismic

Seis is an object that is composed of several datasets and groups: traces are separated from trace headers. Trace headers are kept on `double` format while trace data is `float`. Seis is designed to provide high perfomance while keeping simple API.

4.3.1 Create Seis

Seismic parameters includes: survey type (2D or 3D), data type (stack or prestack), number of traces, number of samples etc:

```
#include <iostream>
#include <h5geo/h5seiscontainer.h>
#include <h5geo/h5seis.h>


int main(){
   std::string fileName = "seis.h5";
   H5SeisCnt_ptr cnt(h5geo::createSeisContainerByName(
      fileName, h5geo::CreationType::OPEN_OR_CREATE));
   if (!cnt){
      std::cout << "Unable to open or create seis container" << std::endl;
      return -1;
   }

   SeisParam p;
   p.domain = h5geo::Domain::OWT;
   p.lengthUnits = "millimeter";
   p.temporalUnits = "millisecond";
   p.angularUnits = "degree";
   p.dataUnits = "psi";
   p.dataType = h5geo::SeisDataType::PRESTACK;
   p.surveyType = h5geo::SurveyType::TWO_D;
   p.nTrc = 30;
   p.nSamp = 10;
   p.srd = 20;
   p.spatialReference = "EPSG:8139";
```

(continues on next page)

(continued from previous page)

```
   std::string seisName = "mySeis";
   H5Seis_ptr seis(cnt->createSeis(
      seisName, p, h5geo::CreationType::OPEN_OR_CREATE));
   if (!seis){
      std::cout << "Unable to open or create seis" << std::endl;
      return -1;
   }

   return 0;
}
```

4.3.2 Write/Read data

By analogy with SEGY format H5Seis contains: text header, binary header, traces headers, trace data.

Write/Read text header

```
std::vector<std::string> txtHdr;
for (size_t i = 0; i < 40; i++)
   txtHdr.push_back("Bart Simpson");
if (!seis->writeTextHeader(txtHdr)){
   std::cout << "Unable to write text header" << std::endl;
   return -1;
}
// another way is to write C-array: 'char txtHdr_c [40][80];'
```

Note: Text header is a dataset of size `[40, 80]`. Thus everything outside of this range will be lost.

To read text header:

```
std::vector<std::string> txtHdr_out =
   seis->getTextHeader();
if (txtHdr_out.size() < 1){
   std::cout << "Unable to read text header" << std::endl;
   return -1;
}
```

Write/Read binary header

The simplest way to write binary header is:

```
// convert 'seconds' to the temporal units of seis object
if (!seis->writeBinHeader("SAMP_RATE", 0.002, "sec", seis->getTemporalUnits())){
   std::cout << "Unable to write samp rate" << std::endl;
   return -1;
}
```

and to get it back:

```
double sampRate = seis->getBinHeader("SAMP_RATE", seis->getTemporalUnits(), "ms");
if (isnan(sampRate))
   std::cout << "Unable to get samp rate" << std::endl;
   return -1;
}
```

Note: List of binary header names is available through `getBinHeaderNames` function declared in h5core_util.h. Header names are consistent to those used in `SEGY` viewer SeiSee

Write/Read trace headers

There are many functions to do this. Here is one of them:

```
Eigen::MatrixXd cdp(3);
cdp << 1, 2, 3;
// write starting from 5th trace
if (!seis->writeTraceHeader("CDP", cdp, 5)){
   std::cout << "Unable to write CDP trace header from 5th trace" << std::endl;
   return -1;
}
```

and to get it back:

```
// get 'cdp' trace header from 3 traces starting from 5th trace
Eigen::MatrixXd cdp_out = seis->getTraceHeader("CDP", 5, 3);
if (cdp_out.size() < 1){
   std::cout << "Unable to get CDP trace header">> std::endl;
   return -1;
}

// update trace header limits is needed when trace headers are written
if (!seis->updateTraceHeaderLimits()){
   std::cout << "Unable to update trace header limits" << std::endl;
   return -1;
}
```

Note: List of trace header names is available through `getTraceHeaderNames` function declared in h5core_util.h. Header names are consistent to those used in `SEGY` viewer SeiSee

Warning: Call `updateTraceHeaderLimits` everytime when trace header `min/max` values changed.

Write/Read trace data

Once again there are many functions to do this, here are some:

```
Eigen::MatrixXd traces(p.nSamp, 3);
Eigen::MatrixXf traces = Eigen::MatrixXf::Random(
   seis->getNSamp(), seis->getNTrc());
// write starting from zero's trace
if (!seis->writeTrace(traces, 0)){
   std::cout << "Unable to write traces" << std::endl;
   return -1;
}
```

Get traces back:

```
// from 3rd trace, 10 traces, from 2nd sample, 5 samples
traces_out = seis->getTrace(3, 10, 2, 5);
if (traces_out.size() < 1){
   std::cout << "Unable to get traces">> std::endl;
   return -1;
}
```

Note: `write/get` trace headers and trace data have pretty wide opportunities including trace selection and working with sorted data. Take a look at seis.h to see all them.

4.3.3 Sorting

The idea behind sorting is to prepare sorting by primary keys (PKey). To accelerate the IO process the user need to add PKey sorting first `addPKeySort` and then use `getSortedData` function to retrieve the data. No need to manually resort data, **h5geo** only keeps indexes and unique values of prepared sortings. In theory this should make work with big data pretty effective.

For example there is widely used sorting `CDP-OFFSET` (`OFFSET` is called `DSREG` in **h5geo**). Add Pkey `CDP` and then you are free to retrieve any `CDP-...` sorted data.

```
if (!seis->addPKeySort("CDP")){
   std::cout << "Unable to add CDP PKey" << std::endl;
   return -1;
}

// then you are allowed to use convenient 'getSortedData' function
Eigen::MatrixXf trace_out;
Eigen::MatrixXd trc_header_out;
// from CDP 1 to 2, from DSREG 0 to 500
// 'trc_ind' - contains indexes of selected traces
Eigen::VectorX<size_t> trc_ind = seis->getSortedData(
   trace_out, trc_header_out, {"CDP", "DSREG"}, {1, 0}, {2, 500});
```

Note: Use `updatePKeySort` when data was mixed.

Sorting uses parallelization over the threads.

Warning: Sorting idea is effetive only if the chosen PKey has many repeating values.

4.3.4 Updating XY boundary around the survey

There is a convenient function to prepare XY boundary around survey. For 3D and 2D prestack data it uses convex hull algorithm. For 2D stack data it simply shows coordinates of traces.

```
if (!seis->updateBoundary()){
   std::cout << "Unable to update boundary" << std::endl;
   return -1;
}
```

To get calculated values:

```
// returned values (two column array) in 'meters' without coordinate system transformation
Eigen::MatrixXd xy_boundary = getBoundary("m", false);
if (!xy_boundary.size() < 1){
   std::endl << "Unable to get boundary" << std::endl;
   return -1;
}
```

4.3.5 Read SEGY

Reading SEGY is pretty simple:

```
if (!seis->readSEGYTextHeader("file.sgy")){
   std::cout << "Unable to read segy text header" << std::endl;
   return -1;
}
if (!seis->readSEGYBinHeader("file.sgy")){
   std::cout << "Unable to read segy binary header" << std::endl;
   return -1;
}
// SEGY files will be concatenated
if (!seis->readSEGYTraces({"file1.sgy", "file2.sgy", "file3.sgy"})){
   std::cout << "Unable to read segy binary header" << std::endl;
   return -1;
}
```

Note: To read SEGY files **h5geo** uses memory-mapping technique and parallelization over the threads (OpenMP library). Thus it should work pretty fast but there is a limitation with memory-mapping: the SEGY files should be on the PC's hard drive. See more on wiki.

4.3.6 Map SEGY

The user may want not to spend time on reading SEGY file but simply map it. In **h5geo** you are allowed to do this at H5Seis creation time:

```
SeisParam p_mapped = p;
p_mapped.mapSEGY = true;
p_mapped.segyFiles = {"file1.sgy", "file2.sgy", "file3.sgy"};

std::string mappedSeisName = "seisMapped";
H5Seis_ptr seisMapped(cnt->createSeis(
   mappedSeisName, p_mapped, h5geo::CreationType::OPEN_OR_CREATE));
if (!seisMapped){
   std::cout << "Unable to open or create mapped seis" << std::endl;
   return -1;
}
```

Then you are free to use it as with regular seis object but with some limitations:

- probably it is impossible to resize file
- data loss when writing to trace headers and binary header (`double` is casted to `int` and `short`)
- only SEGY `ieee-32` format are supported

CHAPTER

FIVE

INDEXES AND TABLES

- genindex
- modindex
- search