Binary Sonar Data Formats

Overview - (How PyHum reads Humminbird files)

PyHum reads data from binary files created by a Humminbird unit (using the 'Record' function). Humminbird units write out the following file formats:

1. *.DAT files which contain basic information about the sonar, time, position and sonar settings. It does this on the first ping, so the time and position refer only to the instant the recording is initiated. The full list of parameters in this file is below
2. *.SON files which contain the 8-bit sonar data (echograms)
3. *.IDX files (1 per SON file) which contain indices of successive pings in the corresponding SON file

One set of data in PyHum consists of up to 4 *.SON files and 1 *.DAT file. If present, the software will use the *.IDX files to more efficiently read in the echogram data from the *.SON files.

How PyHum reads the binary data

The program that reads the data is 'pyread' which is statically compiled from Cython source code (_pyread.pyx), for speed. You can compile this module independently using the 'cython' command and a c-compiler. For example, using the gcc compiler on linux:

```
cython pyread.pyx
gcc -c -fPIC -I/usr/include/python2.7/ pyread.c
gcc -shared pyread.o -o pyread.so
```

The pyread module gets called by the read module in the following way:

```
import pyread
data = pyread.pyread(sonfiles, humfile, c, model, cs2cs_args)
```

where

1. 'sonfiles' is a list of strings containing the paths to the *.SON files
2. 'humfile' is a string contining the filepath to the .DAT file

3. 'c' is the estimated speed of sound (typically 1500 m/s for salt water, 1450 m/s for freshwater)
4. 'model' is the Humminbird unit (units currently known to be supported by PyHum: 798, 898, 998, 1198, and 1199) and 'cs2cs_args' is a string projected coordinate system (for example, "epsg:26949" is Arizona Central State Plane). See the documentation for the module 'pyproj' for more details.

The output variable, 'data', is a python class that returns sonar data.

data.gethumdat() returns data in .DAT file. This is a python dictionary object containing the following keys.

1. 'water_code': 0='fresh' (freshwater), 1='deep salt' (deep saltwater), 2='shallow salt' (shallow saltwater), otherwise = 'unknown'
2. 'sonar_name': a numeric code reported by the instrument
3. 'unix_time': unix (epoch) time in seconds
4. 'utm_x': UTM x coordinate
5. 'utm_y': UTM y coordinate
6. 'filename': string containing the name of the DAT file
7. 'numrecords': number of pings in the SON files
8. 'recordlens_ms': length of time between successive records
9. 'linesize': number of bytes in a line (ping and associated info)
10. 'water_type': a string corresponding to the water code (see above)
11. 'lat': latitude, WGS84 degrees
12. 'lon': longitude, WGS84 degrees

data.getportscans() returns compiled scans from the port side sidescan sonar. This is a 2D (time, distance) numeric array composed of 16-bit floats

data.getstarscans() returns compiled scans from the starboard side sidescan sonar. This is a 2D (time, distance) numeric array composed of 16-bit floats

data.getlowscans() returns compiled scans from the low-frequency (e.g. 83 kHz) side sidescan sonar. This is a 2D (time, distance) numeric array composed of 16-bit floats. Note that a downward scan is taken only every other side scan, so there are half the number of pings in these data as the sidescan data

data.gethiscans() returns compiled scans from the high-frequency (e.g. 200 kHz) side sidescan sonar. This is a 2D (time, distance) numeric array composed of 16-bit floats. Note that a downward scan is taken only every other side scan, so there are half the number of pings in these data as the sidescan data

data.getmetadata() returns a list of metadata compiled per ping. This is a python dictionary object containing the following keys. Each variable is a numeric array of floats.

1. 'lat': latitude, WGS84 degrees
2. 'lon': longitude, WGS84 degrees
3. 'spd': vessel speed, in metres per second
4. 'time_s': unix (epoch) time in seconds
5. 'e': easting coordinate in metres, in projection given by input variable "cs2cs_args"
6. 'n': northing coordinate in metres, in projection given by input variable "cs2cs_args"
7. 'dep_m': depth in metres
8. 'caltime': time elapsed in seconds since start of recording
9. 'heading': course-over-ground heading in degrees

Decoding the .DAT file

700 to 1100 series:

All data are big-endian.

1. byte 1 = spacer
2. bytes 2-5 = integer, water code
3. bytes 6-7 = spacer
4. bytes 8-11 = character, sonar name
5. bytes 12-23 = spacer
6. bytes 24-27 = character, unix time
7. bytes 28-31 = character, utm x coordinate
8. bytes 32-35 = character, utm y coordinate
9. bytes 36-45 = character, filename
10. bytes 46-47 = spacer
11. bytes 48-51 = character, number of records
12. bytes 52-55 = character, record length, milliseconds
13. bytes 56-59 = character, line size
14. bytes 60-65 = spacer

ONIX series:

small hex header with a ascii strings containing info. on sonar frequencies

A note on Humminbird positions

Humminbird records position in 'World Mercator Meters' with no UTM zone (epsg code 3395). Where X = easting, Y = northing, the following formula is used to convert to WGS84:

```
latitude = atan(tan(atan(exp(X/ 6378388.0)) * 2.0 - 1.570796326794897) * 1.00

longitude = Y * 57.295779513082302 / 6378388.0
```

where 'atan' is the inverse tangent in radians, 'tan' is the tangent in radians, 'exp' is the exponential. Note that coordinate transforms using standard geospatial libraries such as Proj.4 (pyproj) and GDAL, are, for some unknown reason, always too inaccurate to use.

A note on Humminbird depths

PyHum applies a time-varying-gain to estimated depth soundings:

```
tvg = ((8.5*10**-5)+(3/76923)+((8.5*10**-5)/4))*c
```

where 'c' is the speed of sound in water. Corrected depth then becomes:

```
depth = tan( 0.4363323 * depth)- tvg
```

Decoding the .SON files

Record structure

The fast way is to use byte indices in IDX files as the start and stop positions (in bytes) of each ping. PyHum will use this way by default. If the IDX files are absent or corrupted, then PyHum will revert to a slightly slower method that finds the start of each record according to the location of header codes. Records are always preceeded by the string of integers: [192,222,171,33,128]. The program finds this header code using the Knuth-Morris-Pratt string-matching algorithm.

Records are composed of 'header' data (containing positions, etc) followed by 'ping' data (composed of time-series of echo levels as 8-bit integers). In order to correctly parse out the two, you need to know how many bytes the header contains. Unfortunately, this length varies with instrument. So far, we have encountered data files with between 67 and 72 bytes per header packet. In the pyhum program, you pass it a variable saying what model of humminbird unit the data comes from. It uses this information in the following way:

```
if model==798:
  headbytes=72
elif model==1199 or onix:
  headbytes=68
else: #tested so far 998, 1198
  headbytes=67
```

SON file header packet

First, we describe the data format of the header packet. All data are big-endian.

Common to all units

1. byte 1 = spacer
2. bytes 2-5 = character, record number
3. byte 6 = spacer
4. bytes 7-10 = character, time in milliseconds
5. bytes 11 = spacer
6. bytes 12-15 = character, UTM x coordinate
7. bytes 16 = spacer
8. bytes 17-20 = character, UTM y coordinate
9. bytes 21 = spacer
10. bytes 22-23 = short integer, GPS quality flag (0 = good, 1=bad)
11. bytes 24-25 = short integer, heading in tenths of a degree
12. byte 26 = spacer
13. bytes 27-28 = short integer, GPS quality flag (0 = good, 1=bad)
14. bytes 29-30 = short integer, speed in cm / s
15. bytes 31-36 = spacer
16. bytes 37-40 = character, depth in cm
17. byte 41 = spacer
18. byte 42 = integer, beam number (=0, 50 or 83 kHz; =1 200 kHz; =2 SI Poort; =3 SI Starboard)

19. byte 43 = spacer
20. byte 44 = integer, volt scale
21. byte 45 = spacer
22. bytes 46-49 = character, frequency in Hz

Then, the data structure is different for different models.

1199 and ONIX models

23. bytes 50-64 = spacer
24. bytes 45-68 = character, sentence length
25. bytes 69 = spacer

798 model

23. byte 50 = spacer
24. bytes 51-62 = character, spacer
25. byte 63 = spacer
26. bytes 64-67 = character, sentence length
27. bytes 68-73 = spacer

898, 998 or 1198 models

23. bytes 50-54 = spacer
24. bytes 55-58 = character, spacer
25. bytes 59-63 = spacer
26. bytes 64-67 = character, sentence length
27. bytes 68 = spacer

SON file sonar data packet

The sonar data is all bytes in the packet after the header bytes have been read in. The data is big endian, unsigned char (8-bit integers).


pyhum

SON

Aus OpenSeaMap-dev

SON is a file format that is used by Humminbird instruments. It structure allows to store data from several sonar sensors to allow for storing so called side scan data. Side scans are continous depth measurments not only below the ship but to some extent to the side of the ships.

Inhaltsverzeichnis

Basic Structure

Byte order of data is most significant byte is the first byte in the stream (Big Endian).

DAT File

For each recording a numbered file is created starting with R00001.DAT. That files stores general information such as time and date of recording and initial position. A corresponding directory R00001 holds a list of files that contain sonar sensor recordings. For earch sensor an index file and a data file is created. A multiple sensor setup may look like having a 50kHz, a 200kHz and two 455kHz side scan sensors for each side of the ship. This will result in four sensor files and four index files totalling to 9 files including the dat file.

```
8 bytes : unknown
int32 : seconds (NOT milliseconds) since January 1, 1970, 00:00:00 GMT.
int32 : longitude (projected)
int32 : latitude (projected)
bytes[6] : Name
bytes[4] : Extension
```

Index Files

Index files contain allow to identify headers and subsequent data in the son file

```
typedef struct indexstructure
{
  int32_t globalRecordNumber; // record number
  int32_t positionInStream; // start of header in other file
}
```

SON Files

Having analyzed the index structure one can access the positions in the file directly. The header consists of a sequence of control characters that indicate the following data (indicated as signed byte value).

Header

```
-128 : int32 // global record number
-127 : int32 // recording time since start
-126 : int32 // longitude (projected)
-125 : int32 // latitude (projected)
-124 : int16 : gps enabled (0/1); int16 : gps heading to be divided by 10.0
-123 : 2 unused bytes; int16 gps speed to be divided by 10.0
-122 : 4 bytes // unknown
-121 : int32 : depth in decimeter
-110 : int32 : sensor frequency in Hz (e.g. 85000Hz)
-107 : int32 // unknown
-96 : int32 // data length for this data block following the header

80 : byte : Beam id (0 for first beam, 1 for second beam, ... encodes the same id as the file name)
81 : byte // unknown
83 : byte // unknown
84 : byte // unknown
86 : byte // unknown
87 : byte // unknown
33 : header termination
```

Example Recording time since start with 1 second in a stream [-127, 0 , 0 , 0, 1]

Data

Yet to be discovered or documented