SEGY manipulation libraries documentation

Chad M. Hogan

June 18 2004

This article documents the SEGY manipulation libraries found in the CREWES software collection. Specifically documented are the functions:

- `SEGY_GetBinaryHeader`
- `SEGY_GetTextHeader`
- `SEGY_GetTrace`
- `SEGY_ReadBinaryHeader`
- `SEGY_ReadTextHeader`
- `SEGY_ReadTrace`
- `SEGY_WriteBinaryHeader`
- `SEGY_WriteGathers`
- `SEGY_WriteStack`
- `SEGY_WriteTextHeader`
- `SEGY_WriteTrace`

This set of functions is designed to generate standards-compliant SEGY rev. 1 (May, 2002) files. It includes a full EBCDIC (or ASCII, if you must) text header, a binary header, and traces with trace headers and data written in big-endian (or little-endian, if you must) format.

Warning

I am not 100% comfortable with my geometry calculations within the sample functions `SEGY_WriteGathers` and `SEGY_WriteStack`. I think they're correct, but it is difficult for me to test them exhaustively. If you are writing SEGY and you are confident in your setting but you think that the produced SEGY file is in error with respect to geometry, it's entirely possible that you are correct. Please contact me and we'll discuss it.

1 A Short Introduction to SEGY

The SEGY data format was first defined a long, long time ago. It arranges things for the convenience of punch-card users, it defaults to EBCDIC text format, and big-endian data. The intent of this library is to write a relatively easy-to-use library of SEGY writing utilities. These are meant to be used as tools incorporated into larger programs, so they are not optimized for use directly from the command line in order to save a random amount of seismic data to a file.

The SEGY data format consists of three main parts. The first 3200 bytes in a file is defined as a text header. This text header, usually in EBCDIC, describes various notes and details that may be interesting to processing and interpreting staff. Although conventions are defined, this block is fairly free-form and cannot be reliably used to automate processing (for example). It is meant to be read by humans, not computers. This is more of a job for the next part of the SEGY file.

Following the 3200-byte text header is the binary header. The binary header is 400 bytes of strictly ordered numbers, usually in big-endian format. These numbers define parameters about the entire data set contained in the file. The line number, the sampling interval, the number of samples per trace, the sorting of the traces, and many other parameters are set in this binary header. These elements are designed to allow computers to quickly understand the data within the file so that they can reliably read and process the traces.

The final part of the SEGY file contains the traces themselves. Each trace has two parts: a trace header and the trace data. The trace header contains information specific to each trace in binary format. This will contain information such as the trace number, the source location for the trace, the receiver group location, the time sample interval, and many others. Following this trace header is the actual trace series consisting of all the samples one after another as a string of binary numbers. There is no "end of trace" marker, which is why it is important to ensure that the number of samples is correctly specified in the binary and trace headers.

For a thorough explanation of the SEGY data format, please see the SEG website at `http://www.seg.org`.

2 Quick Start and Examples

So, you want to just get on with it and write some SEGY already? Ok. This is your section. The first thing to do is to generate your data. You'll want it arranged in one of the following ways:

- In gathers (CDP or shot).
- In a stack.

In the case of gathers, then you will have the entire set of gathers stored in a cell array, with one cell for each gather. Within each gather, the traces will be arranged in an $N \times M$ array, where N is the number of samples in each trace, and M is the number of traces in the gather. For example, we'll imagine that you have all of your gathers stored in a cell array called `gathers`. The third gather in this set is:

```
thirdgather = gathers{3};
```

The second trace within this gather is referenced like this:

```
secondtrace = thirdgather(:, 2);
```

2.1 SEGY_WriteGathers

This function is written mostly as an example for you to follow if you are interested in generating SEGY in your own programs. However, it is directly useful if you are willing to manipulate your data into the format required by this function.

If you can put your data into that format, then you are entirely set for writing SEGY data. You simply use the `SEGY_WriteGathers` function as follows:

```
SEGY_WriteGathers(filename, gathers, dt, type, g1, g2, text, num);
```

Where `filename` is a string containing the full path to the filename that you want to write, `gathers` is your cell array of gathers as previously described, `dt` is the sampling interval in second, `type` is a string containing either 'cdp' or 'shot' to describe the type of gather, `g1` and `g2` are geometry parameters that will be described shortly, `text` is the text header format – either 'ebcdic' or 'ascii', and `num` is the number format – either 'b' for big-endian or 'l' for little endian.

The two geometry parameters `g1` and `g2` require a bit of explanation. They mean different things for the different kinds of gathers.

2.1.1 Writing CDP gathers

To write a CDP gather, you need two elements for the geometry. The first parameter, `g1`, represents the horizontal distance between the depth points within a given gather. If your CDP gathers are each separated by 100 meters, then `g1` will be 100.

The second parameter, `g2`, represents the offset interval from the shot point to the receivers within the gathers. So, if the offsets in a particular CDP gather are something like: 0, 100, 200, 300, ...; then your interval is 100. Note that this particular function assumes you have a zero offset shot. Since I imagine that this library will primarily be used for writing synthetic data, this isn't a bad thing.

From this, geometry will be calculated. It is assumed that the first trace in the gather is the zero-offset gather, and the last trace is the longest-offset gather. Obviously this is going to be a very limited sort of geometry. If you have more complicated data and geometry, then you will have to create your own writing function using the more lower level functions described later.

2.1.2 Writing shot gathers

To write a shot gather, you need two elements for the geometry that are slightly different from the CDP case. The first parameter, `g1`, represents the distance between each shot point. If your shots are separated by 50 meters, then `g1` will be 50.

The second parameter, `g2`, represents the offset interval between receivers. So if you have receivers placed every 100 meters, then `g2` will be 100.

Once again, this functionality is limited. It is assumed that there will be an odd number of traces in this gather. The first and last trace are the longest offset traces (on either side of the shot) and the middle trace is a zero-offset trace. If this does not fit your needs, then you will have to create your own custom writing routine using the lower level functions described in the rest of this document.

2.1.3 Example

This function call will generate a SEGY file in `/tmp/test.segy` using the gathers in `cdpgathers`. The gathers are sampled every 2 milliseconds, and they are sorted into CDP gathers. Each gather's CDP is separated by 50 meters, and the shot-to-receiver interval is 100 meters.

```
    SEGY_WriteGathers('/tmp/test.segy', cdpgathers, 0.002, ...
                      'cdp', 50, 100, 'ebcdic', 'b');
```

2.2 SEGY_WriteStack

This is another example function that you may use if it is convenient. It is much more likely that this will be useful to you for writing stacks, as stack geometry is a lot simpler than gather geometry.

It is called as follows:

```
SEGY_WriteStack(filename, stack, dt, separation, text, num);
```

Where `filename` is a string containing the filename to write, `stack` is an $N \times M$ array of N samples for each of M traces in the stack. `dt` is the sample interval in seconds, `separation` is the horizontal separation distance between each stacked trace, `text` is either 'ebcdic' or 'ascii', and `num` is either 'b' for big-endian or 'l' for little-endian.

2.2.1 Example

This call will save the stack in `stackdata` into the file `/tmp/stacked.segy` with a sample interval of 2 ms. The text will be written in EBCDIC and the data in big-endian format.

```
SEGY_WriteStack{'/tmp/stacked.segy', stackdata, 0.002, 100, ...
                'ebcdic', 'b');
```

3 Driving Functions

These functions do all the heavy lifting, when writing SEGY files. I think of them in two separate groups. The first group is the “retrieving a template” group:

- `SEGY_GetTextHeader`
- `SEGY_GetBinaryHeader`
- `SEGY_GetTrace`

These functions return a template of sorts that can be used as a starting point for your own binary header, text header, and trace when you’re writing SEGY.

The next group is the “writing to file” group:

- `SEGY_WriteTextHeader`
- `SEGY_WriteBinaryHeader`
- `SEGY_WriteTrace`

3.1 SEGY_GetTextHeader

This function will return a text header that is initialized to be the correct size as a standard text header for a SEGY file. It will be returned to you in ASCII, but this is ok because `SEGY_WriteTextHeader` expects an ASCII header input as an argument. All of the ASCII vs. EBCDIC stuff is worked out via arguments to the writing functions. Don’t try to manipulate EBCDIC directly yourself – the 1970s are long over.

The text header itself is a 3200 byte string of information. Take a look at it if you like, and modify it in place to fill in the blanks. Do **not** change the size of this array. The text header is generally used purely as notes to the users of the data, so you can frequently just get a text header as it is, and write it directly without any changes.

```
thead = SEGY_GetTextHeader;
```

3.2 SEGY_GetBinaryHeader

This function returns an initialized structure that represents all of the data found in a SEGY binary header. The details of the SEGY binary header are documented exhaustively in lots of places. One thing worth noting is that I tend to name the binary header elements in the same way as Seismic Unix names them. If you are familiar with Seismic Unix keywords, then you know all of the keywords for this structure as well.

```
bhead = SEGY_GetBinaryHeader;
```

A few of the more important header elements that you probably want to set in every SEGY file are:

bhead.jobid job id number

bhead.lino line number, one line per reel

bhead.reno reel number

bhead.hdt sampling interval in microseconds, this reel

bhead.dto sampling interval in microseconds, in field

bhead.hns number of samples per trace, this reel

bhead.nso number of samples, in field

bhead.tsort trace sorting code

3.3 SEGY_GetTrace

This function returns a template for a SEGY trace. As with the other functions, you get a trace, and then you modify the things that you want to change. You also have to add in the data for the trace. This trace structure includes both the trace header and the trace data itself. So you can use retrieve one template, then for every trace simply replace the time series data for each new trace, and also update the relevant geometry and trace id number information.

```
trace = SEGY_GetTrace;
```

Here are a few of the most important trace header elements. Again, they follow Seismic Unix keyword names.

trace.tracl trace sequence number within the line

trace.tracr trace sequence number within the reel

trace.fldr field record number

trace.ep energy source point number

trace.cdp ensemble number (doesn't have to be cdp)

trace.cdpt trace number within the ensemble

trace.offset distance from source to receiver

trace.sx x source coordinate

trace.gx x group coordinate

trace.dt sample interval in microseconds

trace.data the time series, as an array

Some of these elements will likely be the same for every single trace within a line. For example, `trace.dt` is probably going to be the same for every trace. Others must be altered, usually in a non-trivial way, to accomodate the geometry of the shooting.

3.4 SEGY_WriteTextHeader

This function will write the text header to the SEGY file.

```
SEGY_WriteTextHeader(FILE, thead, format);
```

`FILE` is the opened file handle for the SEGY file, `thead` is the text header as returned by `SEGY_GetTextHeader` and subsequently modified, `format` is equal to 'ebdic' or 'ascii'. You should probably choose 'ebcdic' unless you know otherwise, because EBCDIC conforms to the original SEGY standard.

If there is a text header already in the file that you are writing to, it will be overwritten without warning. You cannot use this function to insert a text header in front of a binary header and/or traces. It will simply overwrite the first 3200 bytes of a file without regard for what exists in those 3200 bytes.

3.5 SEGY_WriteBinaryHeader

This function writes the binary header to the file.

```
 SEGY_WriteBinaryHeader(FILE, bhead);
```

`FILE` is the opened file handle for the SEGY file, and `bhead` is the binary header as returned by the `SEGY_GetBinaryHeader` function and subsequently modified. Note that endianness is determined within your call to `fopen`, and is not a part of this library.

If there is a binary header in the file, this function will overwrite it without warning. This function will overwrite the bytes from 3201 to 3600 (inclusive) without regard for what exists in those bytes. Therefore, you cannot use this to insert a binary header into a file that does not have one.

3.6 SEGY_WriteBinaryHeader

This function will write a SEGY trace to the file.

```
SEGY_WriteTrace(FILE, trace);
```

`FILE` is the opened file handle for the SEGY file and `trace` is the trace as returned by the `SEGY_GetTrace` function and subsequently modified. Note that endianness is determined within your call to the `fopen`, and is not a part of this library.

The trace will be written to the file at exactly the point where the current file pointer resides. If you do not understand the previous sentence, then simply ensure that you include the following line before you try to use the function `SEGY_WriteTrace`:

```
fseek(FILE, 0, 'eof');
```