IDEADLEM
Implementation of Dynamic Extensible Adaptive Locally Exchangeable Measures

Scientific Data Management Group
Lawrence Berkeley National Laboratory
March 15, 2016

sdmsupport@lbl.gov
http://datagrid.lbl.gov/idealem/
https://code.lbl.gov/projects/idealem/

1 Introduction

Handling large streaming data is essential for various applications such as sensor networks, network traffic analysis, social networks, energy cost trends, and environment modeling. However, it is in general intractable to store, compute, search and retrieve large streaming data. We have addressed a fundamental issue with the earlier developed algorithm, which is to reduce the size of large streaming data and still obtain accurate statistical analysis. For example, when a high-speed network such as 100 Gbps network is monitored, the volume of the collected measurement data rapidly grows so that the polynomial time algorithms (e.g., Gaussian processes) become intractable. One possible solution to reduce the vast amounts of measured data is to store a random sample, such as one out of 1000 network packets. However, such static sampling methods (linear sampling)

have drawbacks: (1) it is not scalable for high-rate streaming data, and (2) there is no guarantee of reflecting the underlying distribution.
IDEALEM implements the novel data reduction and pattern searching method based on statistical similarity, and reduces data dynamically while providing accurate analysis of large streaming data. The software can be used for both online and offline data records.

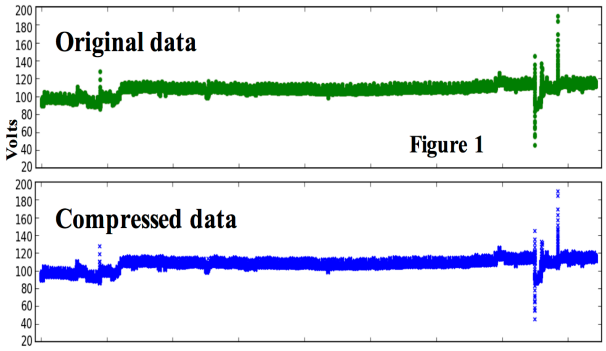

Figure 1: µPMU measurement data

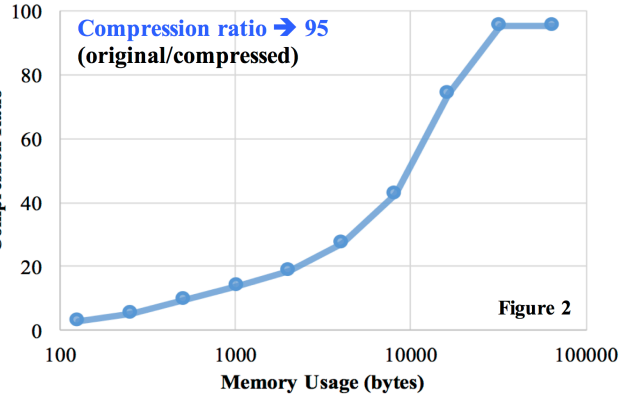

Figure 2: data compression ratio

Fig. 1 shows micro PMU (Phase Measurement Unit) data from power grid electricity measurement from one of the on-site switches at LBNL (in collaboration with Energy Technology Area at the Lab). Compressed (i.e. reduced) data has the accurate representation for the original data, and decompressed (or reconstructed) data has the same statistical data distribution as the original data. Also, the method supports event/feature detection directly on the compressed data.
In Fig 2, the data compression ratio (original size in bytes / compressed size) in the power grid electricity use case is 95.23, and can be achieved using only 64K bytes of memory.

2 Software package

The software is released under a modified BSD license for academic, non-profit, non-commercial use purposes only. For commercial use, please contact the Innovation & Partnerships Office at the Berkeley Lab (http://ipo.lbl.gov).

The software package includes sources for both C and C++ APIs as libraries, a sample program handling data files with a sample data file, and a sample python graph generation tool.

3 Installation and Sample Runs

Web download page http://datagrid.lbl.gov/idealem/ has the downloadable source package. Upon successful download, idealem directory would be created. Configure command will find necessary paths and options. After configure, make and make install will build the C library (libidealem_c.a) and place the files in the distribution directory.

```
cd idealem
./configure
make
make install
ls –l dist/lib
```

Samples directory includes all sample codes and test data files. The details of the programming will be discussed in the next section.

For C++ library, the flag ENABLECXX=1 needs to be added to the make, and C++ library (libidealem_cxx.a) will be built.

```
cd idealem
./configure
make ENABLECXX=1
make install ENABLECXX=1
ls –l dist/lib
```

A sample client tool implementation (idealem_rim) is in samples directory with a test data file. A sample python graph generation tool is located in tester directory.

3.1 Features and Limitations

Compression (encoding) on input files creates binary output data files. Decompression (decoding) on the compressed binary input data creates text output data files, assuming the original data file is in a text format. The input and output format can be changed with a parameter, between text and binary.

Both C and C++ APIs may work up to 255 historical buffers in the current implementations.

Compression ratio is up to 8*N, where N is the number of the LEM window buffer size and 8 is the size of the double. For example, in the above power grid electricity measurement use case, we have used 16 as the number of the LEM window size, and the compression ratio can be achieved up to 128 times (=8*16).

The current implementation uses the Kolmogorov–Smirnov (KS) test as the statistical similarity measure. The similarity measure can have a different method as a plug-in, in the future.

The sample python graph generation tool uses C-based binary-to-text conversion tool in the background for encoded binary output.

3.2 Sample client runs

Samples directory includes idealem_c or idealem_cxx client reference implementation. Client program has a few options:

```
idealem_c -help
usage:  idealem_rim [-options ...]

where options include:
  -h/help
  -debug [default=false]
  -decode [default=false]
  -decodeonly [default=false]
  -encode [default=false]
  -history [default=1]
  -i/in inputfile
  -l/log [logfile, default=./idealem_log.txt]
  -maxhitcount/maxhit/maxcount int [default=255]
  -maxread/read int [default=1000]
  -n/blocklen/lemblklen int [default=16]
  -o/out [outputfile, default=./inputfile.o.data]
  -t/threashold/alpha float [default=0.05]
  -typeei text/bin [encoding input type, default=text]
  -typeeo text/bin [encoding output type, default=bin]
  -typedi text/bin [decoding input type, default=bin]
  -typedo text/bin [decoding output type, default=text]
  -v/version
```

A few example runs are following:

- Usual compression/decompression.
  ```
  idealem_c -i testdata1.txt -encode –decode
  ```
- Input data has multicolumn, and indicating which column to compress or decompress.
  ```
  idealem_c -i testdata7e2.txt  -encode -decode -column 2
  ```
- Using historical buffers
  ```
  idealem_c -i testdata4i8.txt -encode -decode -history 512
  ```
- Having non-default encoded format and decoded format
  ```
  idealem_c -i testdata4e4.txt -encode -decode -typeeo text -typedi text
  ```
- Having a specific decimal point precision
  ```
  idealem_c -i testdata4.txt -encode -decode -p 6
  ```

3.3 Sample plot generator tool commands

Tester directory includes python plot generator tool. The python program has a few options:

```
python ./tester/data_simple_plots.py -h

optional arguments:
  -h, --help            show this help message and exit
  -b BLOCKSIZE, --blocksize BLOCKSIZE               LEM block size
  -c CAPTION_TEXT, --caption CAPTION_TEXT     additional messages on the plot
  -o ORG_FILENAME, --original ORG_FILENAME    original data file path to plot
  -e ENC_FILENAME, --encoded ENC_FILENAME     encoded data file path to plot
  -d DEC_FILENAME, --decoded DEC_FILENAME     decoded data file path to plot
  -l, --l2norm          plot of differences and L2-norm calculation
  -r, --results         additional result messages on the plot
      --allinone        one plot for all data
      --allindep        three plots for data
      --nodisplay           Option not to display plots
      --save OUTPLOT_NAME   output file name for the plot
      --timing              timeing measurements
  -t PLOT_TITLE, --title PLOT_TITLE           plot title
  -x xaxis_title, --xaxis xaxis_title          X axis label on the plot
  -y yaxis_title, --yaxis yaxis_title          Y axis label on the plot
```

A few example runs are following:

- ```
  python data_simple_plots.py \
    -b 16 \
    -o input/testdata1.txt \
    -e input/testdata1.txt.o.data \
    -d input/testdata1.txt.o.data.o.data \
    -t "LEM results" \
    --save "dataplot.png" \
    -l \
    –allindep
  ```
- ```
  python data_simple_plots.py \
    -b 16 \
    --timing \
    --allindep \
    -l \
    --save “dataoplot.png” \
    -o input/testdata1.txt \
    -e input/testdata1.txt.o.data \
    -d input/testdata1.txt.o.data.o.data
  ```

3.4 Notes on the upcoming features

Some work is on the way.

- Work on the binary-based compressed output for multi column streaming data is in progress.

- Other similarity statistics measures are being implemented, besides K-S test. Some applications may benefit from an alternative statistical similarity measure than the K-S test. When ready, the similarity measure will be selective from the parameter to the call.

4 C API

4.1 Header files to include

For client implementation, three files are needed:

```
#include <idealem_common.h>
#include <idealem_util.h>
#include <idealem_file_stream.h>
```

4.2 Compression/Encoding functions

For compression for a single dimensional streaming data, the following call will read the original file, compress and write out a binary compressed file.

```
int idealem_file_stream_inout(
     const char* inputfile_path,
     const char* outputfile_enc_path,     // default=NULL
     const char* logfile_path,            // default=NULL
     bool LOG_flag,                       // default=false
     bool DEBUG_flag,                     // default=false
     int lemblklen,                       // default=16
     double alpha_threashold,             // default=0.05
     int max_hit_count,                   // default=255
                    // max hit count before the encoder sends out index
                    adjusting to smaller value. It can reduce the delay
     int max_input_read,                  // default=1024
     int precision,                       // default=10
     LEM_IO_TYPE_T i_type,                // default=LEM_IO_TEXT
     LEM_IO_TYPE_T o_type);               // default=LEM_IO_BINARY
```

For compression with historical buffers for a single dimensional streaming data, the following call will read the original file, compress and write out a binary compressed file.

```
int idealem_file_stream_inout_with_history(
     const char* inputfile_path,
     const char* outputfile_enc_path,     // default=NULL
     const char* logfile_path,            // default=NULL
     bool LOG_flag,                       // default=false
     bool DEBUG_flag,                     // default=false
     int lemblklen,                       // default=16
     double alpha_threashold,             // default=0.05
     int max_hit_count,                   // default=255
     int max_input_read,                  // default=1024
     int precision,                       // default=10
     int bufferno,                        // default=1, # of historical buffers
     LEM_IO_TYPE_T i_type,                // default=LEM_IO_TEXT
     LEM_IO_TYPE_T o_type);               // default=LEM_IO_BINARY
```

For compression for a multi column/dimensional streaming data, the following call will read the original file, compress and write out a text-based compressed file. Work on the binary-based compressed output for multi column streaming data is in progress.

```
int idealem_file_stream_inout_multivars(
     const char* inputfile_path,
     const char* outputfile_enc_path,     // default=NULL
     const char* logfile_path,            // default=NULL
     bool LOG_flag,                       // default=false
```

```
    bool DEBUG_flag,                    // default=false
    int lemblklen,                      // default=16
    double alpha_threashold,            // default=0.05
    int max_hit_count,                  // default=255
    int max_input_read,                 // default=1024
    int colno,                          // column # to compress, based on 0
    LEM_IO_TYPE_T i_type,               // default=LEM_IO_TEXT
    LEM_IO_TYPE_T o_type);              // default=LEM_IO_TEXT
```

4.3 Decompression/Decoding functions

For decompression from the encoded output for a single dimensional streaming data, the following call will read the compressed/encoded input file, decompress and write a text-based output file as close to the original data file.

```
int idealem_file_stream_decoding(
    const char* inputfile_path,
    const char* outputfile_dec_path,    // default=NULL
    const char* logfile_path,           // default=NULL
    bool LOG_flag,                      // default=false
    bool DEBUG_flag,                    // default=false
    int lemblklen,                      // default=16
    double alpha_threashold,            // default=0.05
    int max_hit_count,                  // default=255
    int max_input_read,                 // default=1024
    int precision,                      // default=10
    LEM_IO_TYPE_T i_type,               // default=LEM_IO_BINARY
    LEM_IO_TYPE_T o_type);              // default=LEM_IO_TEXT
```

For decompression from the encoded output with historical buffers for a single dimensional streaming data, the following call will read the compressed/encoded input file, decompress and write a text-based output file as close to the original data file.

```
int idealem_file_stream_decoding_with_history(
    const char* inputfile_path,
    const char* outputfile_dec_path,    // default=NULL
    const char* logfile_path,           // default=NULL
    bool LOG_flag,                      // default=false
    bool DEBUG_flag,                    // default=false
    int lemblklen,                      // default=16
    double alpha_threashold,            // default=0.05
    int max_hit_count,                  // default=255
    int max_input_read,                 // default=1024
    int precision,                      // default=10
    int bufferno,                       // default=1, # of historical buffers
    LEM_IO_TYPE_T i_type,               // default=LEM_IO_BINARY
    LEM_IO_TYPE_T o_type);              // default=LEM_IO_TEXT
```

For decompression from the encoded output for a multi column/dimensional streaming data, the following call will read the compressed/encoded input file, decompress and write a text-based output file as close to the original data file.

```
int idealem_file_stream_decoding_multivars(
    const char* inputfile_path,
    const char* outputfile_enc_path,    // default=NULL
    const char* logfile_path,           // default=NULL
    bool LOG_flag,                      // default=false
    bool DEBUG_flag,                    // default=false
    int lemblklen,                      // default=16
    double alpha_threashold,            // default=0.05
    int max_hit_count,                  // default=255
    int max_input_read,                 // default=1024
    int colno,                          // column # to compress, based on 0
    LEM_IO_TYPE_T i_type,               // default=LEM_IO_TEXT
```

```
     LEM_IO_TYPE_T o_type);              // default=LEM_IO_TEXT
```

4.4 Detailed functions

Individual function calls contribute to the compression and decompression of the data blocks.

To read a block of text-based input data in double precision.

```
int read_double(
     void *ptr,                // pointer to the block
     size_t nitems,            // size of the block (block size)
     FILE *stream,             // input file descriptor
     int max_input_read);      // default=1024
```

To read a block of binary-based input data in double precision.

```
int read_double_bin(
     void *ptr,                // pointer to the block
     size_t nitems,            // size of the block (block size)
     FILE *stream,             // input file descriptor
     int max_input_read);      // default=1024
```

To read a block of binary-based input data in double precision.

To read a block of multicolumn input data.

```
int read_double_in_multivars(
     void* mptr,               // pointer to the block for the whole line
     int column,               // column # for compression/decompression
     void *ptr,                // pointer to the block
     size_t nitems,            // size of the block (block size)
     FILE *stream,             // input file descriptor
     int max_input_read);      /// default=1024
```

To read a hit count in text-based input data.

```
int read_hit_count(
     void *ptr,                // pointer to the hit counter
     size_t nitems,            // default=1
     FILE *stream,             // input file descriptor
     int max_input_read);      // default=1024
```

To read a hit count in binary-based input data.

```
int read_hit_count_bin(
     void *ptr,                // pointer to the hit counter
     size_t nitems,            // default=1
     FILE *stream,             // input file descriptor
     int max_input_read);      // default=1024
```

To write a block of text-based output data in double precision.

```
int write_double(
     const void *ptr,          // pointer to the block
     size_t nitems,            // size of the block (block size)
     FILE *stream,             // output file descriptor
     int precision);           // floating point decimal precision
```

To write a block of binary-based output data in double precision.

```
int write_double_bin(
     const void *ptr,          // pointer to the block
     size_t nitems,            // size of the block (block size)
     FILE *stream,             // output file descriptor
     int precision);           // floating point decimal precision
```

To write a block of text-based output data in multicolumn.

```
int write_multivars(
```

```
     const void *mptr,         // pointer to the block for the whole line
     size_t nitems,            // size of the block (block size)
     FILE *stream);            // output file descriptor
```

To write a string line buffer in text-based output data.

```
int write_a_string(
     const void *mptr,         // pointer to the character string
     FILE *stream);            // output file descriptor
```

To write a hit count in text-based output data.

```
int write_hit_count(
     const void *ptr,          // pointer to the hit counter
     size_t nitems,            // default=1
     FILE *stream);            // output file descriptor
```

To write a hit count in binary-based output data.

```
int write_hit_count_bin(
     const void *ptr,          // pointer to the hit counter
     size_t nitems,            // default=1
     FILE *stream);            // output file descriptor
```

To write a decompressed block in text-based output data. The decompressed blocks are randomly permuted from the base block.

```
void permuted_output(
     int cnt,             // hit count. # of times to repeat the permuted block
     double* buffer,      // pointer to the base block
     int lemblklen,       // size of the block (block size)
     FILE* outputfile_d,  // output file descriptor
     int precision);      // floating point decimal precision
```

To write a decompressed block in binary-based output data. The decompressed blocks are randomly permuted from the base block.

```
void permuted_output_bin(
     int cnt,             // hit count. # of times to repeat the permuted block
     double* buffer,      // pointer to the base block
     int lemblklen,       // size of the block (block size)
     FILE* outputfile_d,  // output file descriptor
     int precision);      // floating point decimal precision
```

To write a decompressed block for multicolumn data in text-based output. The decompressed blocks are randomly permuted from the base block.

```
void permuted_output_in_multivars(
     int cnt,             // hit count. # of times to repeat the permuted block
     char** buffer,       // pointer to the base block
     int lemblklen,       // size of the block (block size)
     FILE* outputfile_d); // output file descriptor
```

To write a decompressed block from the encoded data with historical buffer in text-based output. The decompressed blocks are randomly permuted from the historical base blocks.

```
int permuted_output_cdf(
     LEM_BLOCK* partitions,    // pointer to the historical buffers and count
     unsigned char partitionsidx,   // index to the historical buffer to output
     int lemblklen,            // size of the block (block size)
     FILE* outputfile_d,       // output file descriptor
     int precision);           // floating point decimal precision
```

To determine exchangeability between two blocks.

```
bool idealem_exchangeability_single_stream(
     double* sorted_block,          // pointer to the block
```

```
        int sorted_block_size,          // size of the block (block size)
        double*prev_sorted_block,       // pointer to the block to be compared to
        int prev_sorted_block_size,     // size of the block to be compared to
        double alpha_threshold,         // threshold for the statistics
        unsigned char *hitcount);       // hit count for the block similarity
```

To provide statistical similarity measure based on the K-S test.

```
double KStest(
        const double* a,     // pointer to the block
        int a_size,          // size of the block (block size)
        const double* b,     // pointer to the block to be compared to
        int b_size);         // size of the block to be compared to
```

For the block definition for compression/decompression with historical buffers.

```
typedef struct lemblk {
        double* block; // block holding original sequence samples
        double* sorted_block; // block holding sorted samples for CDF comparison
        int count;    // counter indicating the repeated blocks (used for decoding)
} LEM_BLOCK;
```

For the I/O type definition for inputs and outputs of compression/decompression. Defaults for compression is LEM_IO_TEXT for input and LEM_IO_BINARY for output. Defaults for decompression is LEM_IO_BINARY for input and LEM_IO_TEXT for output.

```
enum LEM_IO_TYPE { LEM_IO_TEXT, LEM_IO_BINARY, LEM_IO_ONLINE };
typedef enum LEM_IO_TYPE  LEM_IO_TYPE_T;
```

5 C++ API

5.1 Header files to include

For client implementation, three files are needed:

```
#include <idealem_common.h>
#include <idealem_util.h>
#include <idealem_file_stream.h>
```

5.2 Class construction and initialization

For compression or decompression for a single dimensional streaming data, the following calls will construct the class variable and initialize the variable.

```
IDEALEM_SINGLE_STREAM idealem_rim_encoding(     // constructor
        int lemblocklen,            // size of the block (block size)
        double alphathreshold,      // test statistics threshold,default=0.05
        bool encoding=true);        // flag for compression, default=true

idealem_rim_encoding.init_for_files(   // initializing the class
        const char* inputfilepath,     // input file path
        const char* outputfilepath,    // output file path
        LEM_IO_TYPE_T itype,           // input format
        LEM_IO_TYPE_T otype,           // output format
        int maxhitcount=255,           // max hit count
        int maxinputread=1024,         // max size of the input line
        int tprecision=10,             // floating point precision
        int varcolumn=-1,              // column # to compress
        int historybuffer=1);          // # of historical buffers

idealem_rim_encoding.init_for_files(   // alternate initializing the class
        const char* inputfilepath,     // input file path
```

```
        const char* outputfilepath,      // output file path
        int maxhitcount=255,             // max hit count
        int maxinputread=1024,           // max size of the input line
        int tprecision=10,               // floating point precision
        int varcolumn=-1,                // column # to compress
        int historybuffer=1);            // # of historical buffers

    // encoding default itype=LEM_IO_TEXT, otype=LEM_IO_BINARY
    // decoding default itype=LEM_IO_BINARY, otype=LEM_IO_TEXT
```

5.3 Compression/Encoding functions

For compression for a single dimensional streaming data, the following call after the class variable is constructed and initialized (section 5.2) will read the original file, compress and write out a binary compressed file.

```
idealem_rim_encoding.encoding();
```

For compression with historical buffers for a single dimensional streaming data, the following call after the class variable is constructed and initialized (section 5.2) will read the original file, compress and write out a binary compressed file.

```
idealem_rim_encoding.encoding_with_history();
```

For compression for a multi column/dimensional streaming data, the following call after the class variable is constructed and initialized will read the original file, compress and write out a text-based compressed file. Work on the binary-based compressed output for multi column streaming data is in progress.

```
idealem_rim_encoding.encoding_multivars();
```

5.4 Decompression/Decoding functions

For decompression from the encoded output for a single dimensional streaming data, the following call after the class variable is constructed and initialized (section 5.2) will read the compressed/encoded input file, decompress and write a text-based output file as close to the original data file.

```
idealem_rim_decoding.decoding();
```

For decompression from the encoded output with historical buffers for a single dimensional streaming data, the following call after the class variable is constructed and initialized (section 5.2) will read the compressed/encoded input file, decompress and write a text-based output file as close to the original data file.

```
idealem_rim_decoding.decoding_with_history();
```

For decompression from the encoded output for a multi column/dimensional streaming data, the following call after the class variable is constructed and initialized (section 5.2) will read the compressed/encoded input file, decompress and write a text-based output file as close to the original data file.

```
idealem_rim_decoding.decoding_multivars();
```

5.5 Detailed class methods definitions

Class IDEALEM_SINGLE_STREAM can be used for both compression and decompression. Besides the class variable construction and initialization (section 5.2), the following class methods can update the details of the class variable.

To set/get the input file path.

```
bool set_inputfile_path(const char* inputfilepath);
const char* get_inputfile_path();
```

To set/get the output file path.

```
bool set_outputfile_path(const char* outputfilepath);
const char*  get_outputfile_path();
```

To set/get the log output file path, and check the logging status.

```
bool set_logfile_path(const char* logfilepath);
const char*  get_logfile_path();
bool isLog();
```

To set the debugging flag, and check the debugging status.

```
bool set_debug(bool debugflag);
bool isDebug();
```

To check the encoding status. When true, the class variable is for encoding/compression. When false, the class variable is for decoding/decompression.

```
bool isEncoding();
```

To set/get the input/output format type. For encoding, default input type is LEM_IO_TEXT, and default output type is LEM_IO_BINARY. For decoding, default input type is LEM_IO_BINARY, and default output type is LEM_IO_TEXT.

```
LEM_IO_TYPE_T set_input_type(LEM_IO_TYPE_T itype);
LEM_IO_TYPE_T get_input_type();
LEM_IO_TYPE_T set_output_type(LEM_IO_TYPE_T otype);
LEM_IO_TYPE_T get_output_type();
```

To set/get the size of the block for compression/decompression.

```
int set_lem_block_size(int lemblocklen);
int get_lem_block_size();
```

To set/get the threshold for statistical similarity.

```
double set_alpha_threshold(double alphathreashold);
double get_alpha_threshold();
```

To set/get the maximum hit count for similar blocks.

```
int set_max_hit_count(int maxhitcount);
int get_max_hit_count();
```

To set/get the maximum input size for reading files.

```
int set_max_input_read(int maxinputread);
int get_max_input_read();
```

To set/get the precision for the floating point numbers.

```
int set_precision(int tprecision);
int get_precision();
```

To set/get the column number for the compression/decompression for multicolumn inputs.

```
int set_var_column(int varcolumn);
int get_var_column();
```

To set/get the number of historical buffers.

```
int set_history_buffer(int historybuffer);
int get_history_buffer();
```

To determine exchangeability between two blocks.

```
bool get_exchangeability(
     double* sorted_block,        // pointer to the block
     int sorted_block_size,       // size of the block (block size)
     double* prev_sorted_block,  // pointer to the block to be compared to
     int prev_sorted_block_size);     // size of the block to be compared to
```

5.6 Common functions

Class methods from IDEALEM_SINGLE_STREAM use some of the functions from C API.

To provide statistical similarity measure based on the K-S test.

```
double KStest(
     const double* a,     // pointer to the block
     int a_size,          // size of the block (block size)
     const double* b,     // pointer to the block to be compared to
     int b_size);         // size of the block to be compared to
```

For the block definition for compression/decompression with historical buffers.

```
typedef struct lemblk {
     double* block; // block holding original sequence samples
     double* sorted_block; // block holding sorted samples for CDF comparison
     int count;    // counter indicating the repeated blocks (used for decoding)
} LEM_BLOCK;
```

For the I/O type definition for inputs and outputs of compression/decompression. Defaults for compression is LEM_IO_TEXT for input and LEM_IO_BINARY for output. Defaults for decompression is LEM_IO_BINARY for input and LEM_IO_TEXT for output.

```
enum LEM_IO_TYPE { LEM_IO_TEXT, LEM_IO_BINARY, LEM_IO_ONLINE };
typedef enum LEM_IO_TYPE  LEM_IO_TYPE_T;
```

6 References

1. IDEALEM downloads, http://datagrid.lbl.gov/idealem/.
2. J. Choi, K. Hu, A. Sim, "*Relational Dynamic Bayesian Networks with Locally Exchangeable Measures*", LBNL 6341E, 2013.
3. J. Choi, A. Sim, "*Efficient Data Reduction Method with Locally Exchangeable Measures*", LBNL IB-2013-133, 2013.
4. J. Choi, A. Sim, "*Data reduction methods, systems, and devices*", U.S. Patent Pending serial no. 14/555,365, 2014.