# Fast Error-Bounded HPC Data Compressor (sz-1.4)

# User Guide (Version 1.4.12)

**Mathematics and Computer Science (MCS)**
**Argonne National Laboratory**
**Contact: Sheng Di (sdi1@anl.gov)**
**December, 2017**

## Table of Contents

## 1. Brief description

- **SZ** is an **error-bounded** HPC in-situ data compressor for significantly reducing the data sizes, which can be leveraged to improve the checkpoint/restart performance and post-processing efficiency for HPC executions.
- SZ can be used to compress different types of data (single-precision and double-precision) and any shapes of the array. Current version supports up to five dimensions. Higher dimensions can also be extended easily.

- SZ is very easy to use. It supports three programming languages: Fortran, C and Java.
- SZ supports many different architectures, including x86_32bits (denoted by linux_x86 in the Makefile), x86_64bits (denoted by linux_x64 in the Makefile), ARM (denoted by linux_arm), SOLARIS (denoted by solaris), IBM BlueGene series (denoted by pps).
- SZ allows setting the compression error bound based on *absolute error bound* and/or *relative error bound*, or *point-wise relative error bound*, by using sz.config (which can be found in the directory *example*) or by passing arguments through programming interfaces.
  - **Absolute error bound** (namely *absErrBound* in the configuration file sz.config): It is to limit the (de)compression errors to be within an absolute error. For example, absErrBound=0.0001 means the decompressed value must be in [V-0.0001,V+0.0001], where V is the original true value.
  - **Relative error bound** (called *relBoundRatio* in the configuration file sz.config): It is to limit the (de)compression errors by considering the **global data value range size** (i.e., taking into account the range size (max_value - min_value)). For example, suppose relBoundRatio is set to 0.01, and the data set is {100,101,102,103,104,...,110}. That is, the maximum value is 110 and minimum value is 100. So, the global value range size is 110-100=10, and the error bound will actually be 10*0.01=0.1, from the perspective of "relBoundRatio".
  - **Point-wise relative error bound:** It is to control the compression errors based on a relative error ratio in comparison with each data point's value. For example, given point-wise relative error bound = 0.01, then the real compression error bound for each data point will be equal to 0.01*{the current data value}. SZ will adopt the point-wise relative error bound mode when setting errBoundMode to **PW_REL**.
- Users can set the real compression error bound based on only absErrorBound, relBoundRatio, or a kind of combination of them. Two types of combinations are provided: **AND, OR**. ABS_**AND**_REL means that both of the two bounds (absErrorBound and relBoundRatio) will be considered in the compression. ABS_**OR**_REL means that the compression error is satisfied as long as one type of bound is met. Current version doesn't support combination of PW_REL and other types of bounds.
- If there are many variables to be compressed, we recommend to compress them using batch-compression way. Specifically, there are two steps in the batch-compression: (1) register/add variables, and (2) perform the compression. Please reference the description of SZ_batchAddVar() and SZ_batch_compress(). An example code (testfloat_batch_compress.c) can also be found in example/ directory.
- Users are allowed to set the endian type of the data in the sz.config. Please check the comments of this file in the example/ directory.

## 2. How to install SZ

The SZ software can be downloaded from http://collab.mcs.anl.gov/display/ESR/SZ
Perform the following three simple steps to finish the installation:
configure --prefix=[INSTALL_DIR]
make
make install

You'll find all the executables in [INSTALL_DIR]/bin and .a and .so libraries in [INSTALL_DIR]/lib
**Note 1:** If you want to enable fortran compilation, please use --enable-fortran option when running the "configure –prefix=[]" command. The default compilation is without fortran.
**Note 2:** the dynamic link and static link are named as lib**SZ**.so and lib**SZ**.a (uppercases), because libsz.so and libsz.a are generally referred to as szip compressor.

# 3. Quick Start

The testing cases can be found in **[SZ_Package]/example**
You can use "make clean;make" to recompile all the example codes, or compile them by the customized Makefile.bk as follows:
make -f Makefile.bk
(Makefile.bk allows you to compile your customized source codes.)

For simplicity, you can use [SZ_Package]/example/*test.sh* to test all examples.
You can also use the executable command ./sz to test the compression/decompression.

### 3.1 Executable command -- sz

You can use the executable command "sz" to do the compression and decompression simply. The input data file is in binary format.
**Usage: sz <options>**
Options:
* operation type:
    -z: compression
    -x: decompression
    -p: print meta data (configuration info)
* data type:
    -f: single precision (float type)
    -d: double precision (double type)
* configuration file:

-c <configuration file> : configuration file sz.config

* input data file:

-i <original data file> : original data file

-s <compressed data file> : compressed data file in decompression

* output type of decompressed file:

-b (by default) : decompressed file stored in binary format

-t : decompressed file stored in text format

-W : pre-processing with wavelets transform

-T : compression with Tucker Tensor Decomposition.

* dimensions:

-1 <nx> : dimension for 1D data such as data[nx]

-2 <nx> <ny> : dimensions for 2D data such as data[ny][nx]

-3 <nx> <ny> <nz> : dimensions for 3D data such as data[nz][ny][nx]

-4 <nx> <ny> <nz> <np>: dimensions for 4D data such as data[np][nz][ny][nx]

* print compression results:

-a : print compression results such as distortions

* examples:

sz -z -f -c sz.config -i testdata/x86/testfloat_8_8_128.dat -3 8 8 128

sz -x -f -s testdata/x86/testfloat_8_8_128.dat.sz -3 8 8 128

sz -x -f -s testdata/x86/testfloat_8_8_128.dat.sz -i testdata/x86/testfloat_8_8_128.dat -3 8 8 128 -a

sz -z -d -c sz.config -i testdata/x86/testdouble_8_8_128.dat -3 8 8 128

sz -x -d -s testdata/x86/testdouble_8_8_128.dat.sz -3 8 8 128

sz -p -s testdata/x86/testdouble_8_8_128.dat.sz

**Remark**:

- -W: this operation allows you to perform an optional wavelet transform for the raw dataset, and then perform the lossy compression based on the SZ framework.
- -T: This option allows you to use Tucker Tensor Decomposition to compress high-dimensional data set. It will call TuckerMPI to do the compression.
- -a: This option is only valid when doing the decompression, because it includes the analysis of the data distortion. You need to specify the original data file and decompressed data file by “-i” and “s” respectively.

### 3.2 Compression using example codes

**Testing examples**:

Run “**./testdouble_compress sz.config testdata/x86/testdouble_8_8_128.dat 8 8 128**” to compress the data testdouble_8_8_128.dat.

Run “**./testdouble_compress sz.config testdata/x86/testdouble_8_8_8_128.dat 8 8 8 128**” to compress the data testdouble_8_8_8_128.dat.

Run “**./testfloat_compress sz.config testdata/x86/testfloat_8_8_128.dat 8 8 128**” to compress the data testfloat_8_8_128.dat

**Remark**:
testdouble_8_8_128.dat and testdouble_8_8_8_128.dat are two binary testing files, which contain a 3d array (128X8X8) and a 4d array (128X8X8X8) respectively. Their data values are shown in the two plain text files, testdouble_8_8_128.txt and testdouble_8_8_8_128.txt. These two data files are from FLASH_Blast2 and FLASH_MacLaurin respectively (the two test data are both extracted at time step 100). The compressed data files to be generated are named testdouble_8_8_128.dat.sz and testdouble_8_8_8_128.dat.sz respectively.

./testfloat_compress.c is an example to show how to compress single-precision data. Use test**float**_8_8_128.dat as the input when testing the compression of single-precision data.

**sz.config** is the configuration file used to set the compression environment. Please read the comment in the file to understand the parameters.

## 3.3 Error Control Setting

The key settings regarding error controls are *errorBoundMode*, *absErrBound*, and *relBoundRatio*, which are described below.

- ***errorBoundMode*** is to define a combination of the above two types of error bounds. There are six fundamental types of values:
  **ABS, REL, ABS_AND_REL, ABS_OR_REL, PW_REL, and PSNR.**
  - **ABS** takes only "absolute error bound" into account. That is, relative bound ratio will be ignored.
  - **REL** takes only "relative bound ratio" into account. That is, absolute error bound will be ignored.
  - **ABS_AND_REL** takes both of the two bounds into account. The compression errors will be limited using both absErrBound and relBoundRatio*rangesize. That is, the two bounds must be both met.
  - **ABS_OR_REL** takes both of the two bounds into account. The compression errors will be limited using either absErrBound or relBoundRatio*rangesize. That is, only one bound is required to be met.
  - **PW_REL** takes “point-wise relative error bound”. The error bound for a data point is equal to the pw_relBoundRatio * its data value. Please read the comment in sz.config for details.
  - **PSNR** refers to *peak signal to noise ratio.* SZ allows users to do the compression with a fixed PSNR. The PSNR value is set through the parameter “psnr” in the sz.config.
  - **ABS_AND_PW_REL**: MIN{absErrBound, pw_relBoundRatio*value }
  - **ABS_OR_PW_REL**: MAX{absErrBound, pw_relBoundRatio*value }
  - **REL_AND_PW_REL**: MIN{ relBoundRatio*rangesize, pw_relBoundRatio*value }
  - **REL_OR_PW_REL**: MAX{ relBoundRatio*rangesize, pw_relBoundRatio*value }

- ***absErrBound*** refers to the absolute error bound, which is to limit the (de)compression errors to be within an absolute error. For example, absErrBound=0.0001 means the decompressed value must be in [V-0.0001,V+0.0001], where V is the original true value.
- ***relBoundRatio*** refers to value-range based relative bound ratio, which is to limit the (de)compression errors by considering the global data value range size (i.e., taking into account the range size (max_value - min_value)). For example, suppose relBoundRatio is set to 0.01, and the data set is {100,101,102,103,104,...,110}. In this case, the maximum value is 110 and the minimum is 100. So, the global value range size is 110-100=10, and the error bound will be 10*0.01=0.1, from the perspective of "relBoundRatio".
- ***pw_relBoundRatio*** refers to *point-wise relative Bound Ratio.* pw_relBountRatio is to limit the (de)compression errors by considering the point-wise original data values. For example, suppose pw_relBoundRatio is set to 0.01, and the data set is {100,101,102,103,104,...,110}, so the compression errors will be limited to {1,1.01,1.02,....1.10} for the data points. This parameter is only valid when errorBoundMode = PW_REL.

### 3.4 Decompression using example codes

**Testing examples:**

**./testdouble_decompress sz.config testdata/x86/testdouble_8_8_128.dat.sz 8 8 128**
**./testdouble_decompress sz.config testdata/x86/testdouble_8_8_8_128.dat.sz 8 8 8 128**
**./testfloat_decompress sz.config testdata/x86/testfloat_8_8_128.dat.sz 8 8 128**

**Remark**:

- Unlike the compression step, you don’t have to provide the error bound information (such as errBoundMode, absErrBound, and relBoundRatio), when performing the data decompression, because such information is stored in the compressed data stream.
- The output files of the test_decompress.c are .out files, i.e., testdouble_8_8_128.dat.sz.out and testdouble_8_8_8_128.dat.sz.out respectively. You can compare .txt file and .out file for checking the compression errors for each data point. For instance, compare testdouble_8_8_8_128.txt and testdouble_8_8_8_128.dat.sz.out.

# 4. Initialization of SZ environment

As you can see in the test cases, the SZ requires loading some parameters beforehand for compressing the floating-point data sets. This parameter loading step is performed by SZ_Init(configFilePath) or SZ_Init_Params(params) function, and it just needs to be called once in order to compress multiple data sets stored in different variables.

- SZ_Init(configFilePath) loads the parameters by reading a configuration file (named sz.config), which can be found in the ./example directory.
- SZ_Init_Params(params) initialize the compression environment by passing the parameter data structure. Its definition can be found in the sz.h.

## 5. Compression Modes

SZ provides two compression modes, including SZ_BEST_SPEED and SZ_BEST_COMPRESSION.

- **SZ_BEST_SPEED**: SZ will compress the data sets as fast as possible, by ignoring the Gzip step.
- **SZ_BEST_COMPRESSION**: SZ will try to compress the data sets with a high compression factor.

Basically, SZ_BEST_SPEED will lead to a much faster compression than the SZ_BEST_COMPRSSION, while the latter may leads to better compression ratio with the same error bound because it adopts Zlib in the end of its compression procedure.

## 6. Optimization of compression by tuning the configuration

SZ provides different modes and some parameters for users to tune the compression on demand, e.g., to get either best speed or best compression factor.

The most important parameters that may affect the compression speed and compression ratio are quantization_intervals, max_quant_intervals, szMode and gzipMode.

(1) **quantization_intervals** = ? (this parameter refers to the number of quantization bins). When the quantization_intervals is set to 0, the compressor will search the most appropriate number of quantization bins with the maximum value (max_quant_intervals). This searching step may cost 15% execution time. In fact, in some cases, you can easily estimate the appropriate quantization_intervals to avoid the searching cost, if you know the value range and the error bound. For example, if the value range is [10,30], and error bound is 0.01, then there will be at most (30-10)/0.01=2000 bins. Then, the number of quantization intervals could be set to 2048.

(2) **max_quant_intervals** is the maximum number of quantization bins when searching the optimal number of quantization bins. This parameter is valid only when quantization_intervals = 0. The larger the max_quant_intervals is, the better the compression factor generally is, but the slower the execution time is. As for the very hard-to-compress cases with very high-precision demand, you can set it to a high

number such as 2097152 or so. Otherwise, you are recommended to set it to a low number such as 65536 or 256, depending on how easy/smooth the data is and the error bound you give.

(3) **szMode** is the compression mode of SZ. It has three options: SZ_BEST_SPEED, SZ_DEFAULT_COMPRESSION, SZ_BEST_COMPRESSION. The difference between SZ_BEST_SPEED and the other two modes is that it will not miss Gzip step in the compression. Gzip step may take 20-50% time of the whole compression, depending on the data set. SZ_DEFAULT_COMPRESSION and SZ_BEST_COMPRESSION are very similar (the only difference is different sliding window size set in Gzip, which may lead to a little bit different compression time and compression factor).

(4) **gzipMode** is the compression mode of Gzip. Obviously, this parameter setting is valid only when szMode is set to either SZ_DEFAULT_COMPRESSION or SZ_BEST_COMPRESSION.

In summary, the above four parameters can be tuned to get different compression speed and compression factor on demand.

- ♦ The fastest-speed setting is { **quantization_intervals**=256, **max_quant_intervals** =0, **szMode** = SZ_BEST_SPEED, **gzipMode** = Gzip_BEST_SPEED}.
  (**note**: max_quant_intervals and gzipMode will be ignored in this setting)
- ♦ The best-compression-factor setting is { **quantization_intervals**=0, **max_quant_intervals** = 2097152, **szMode** = SZ_BEST_COMPRESSION, **gzipMode** = Gzip_BEST_COMPRESSION}.
  (**note**: max_quant_intervals could be set even higher if needed)

# 7. Application Programming Interface (API)

Programming interfaces are provided in two programming languages – C and Fortran (SZ-0.x versions also provided Java interfaces). The usage methods of the interfaces are quite similar across different programming languages, with only a few differences. For example, In C interface, a *dataType* (SZ_FLOAT, SZ_DOUBLE, SZ_INT8, SZ_INT16, SZ_INT32, or SZ_INT64) is required, while Fortran interface doesn't require this argument because of the function overloading feature.

## 7.1 Compression/Decompression by C Interfaces

There are three key interfaces for compression/decompression in C.

(1) Initialize the compressor by calling SZ_Init();

(2) Compress the data (a floating-point array) by SZ_compress(), or decompress the data by SZ_decompress();

(3) Finalize the compressor by SZ_Finalize() if the compressor won't be used any more.

**Interfaces**:

**(a)** ***SZ_Init*** **and** ***SZ_Init_Params***

Initialize the SZ compressor. SZ_Init() just needs to be called only **once** before performing multiple compressions for different variables (data arrays).

**Synopsis**: **void SZ_Init(char *configFilePath);**

**Input:**

**configFilePath** the configuration file path (such as example/sz.config)

**Return:** none.

**Synopsis**: **void SZ_Init_Params(sz_params * params);**

**Input:** **params** the configuration variable that contains the initialization information.

**Return:** none.

**zz_params data structure:**

```
typedef struct sz_params
{
    unsigned int max_quant_intervals; //max number of quantization intervals
    unsigned int quantization_intervals; //default value: 0
    int dataEndianType; //what is the endian type of the original data set?
    int sysEndianType; //sysEndianType can be ignored, because it can be
detected autumnally by our compressor based on the system architectures.
    ~~int sol_ID~~; //default value: #define SZ 101 (deprecated)
    ~~int layers~~; //default value: 1 (deprecated)
    int sampleDistance; //default value: 50
    float preThreshold; //default value: 0.97
    ~~int offset~~; //default value: 0 (deprecated)
    int szMode; //default value: #define SZ_BEST_COMPRESSION 1
    int gzipMode; //default value: Gzip_BEST_SPEED
    int errorBoundMode; //4 options: ABS, REL, ABS_AND_REL, ABS_OR_REL
    double absErrBound; //example: 0.0001
    double relBoundRatio; //example: 0.001
    double psnr; //peak signal to noise ratio, example: 80
    double pw_relBoundRatio; //point-wise relative error bound
    int segment_size; //# points in each segment for pw_relBoundRatio
    int pwr_type; //point-wise relative error bound byte, example: 25
} sz_params;
```

**(Detailed description of the above parameters can be found in the sz.config)**

**(b)** ***SZ_compress***

Compress the floating-point data array. Two types of interfaces are provided, as shown below. For the first one, the error controlling parameters (such as errBoundMode, absErrBound, and relBoundRatio) will be given by the configuration file sz.config. For the second one, the error controlling parameters will be passed using arguments, so in this case, the parameter settings in the sz.config will be ignored.

There are three compression interfaces with different arguments, as listed below. The user just needs to choose one of them in compressing data.

**Synopsis**:

```
char *SZ_compress(int dataType, void *data, size_t *outSize, size_t r5, size_t r4,
size_t r3, size_t r2, size_t r1);
char *SZ_compress_args(int dataType, void *data, size_t *outSize,
        int errBoundMode, double absErrBound, double relBoundRatio,
        double pwrBoundRatio, int pwrType,
        size_t r5, size_t r4, size_t r3, size_t r2, size_t r1);
int SZ_compress_args2(int dataType, void *data, char* compressed_bytes,
        size_t *outSize,
        int errBoundMode, double absErrBound, double relBoundRatio,
        double pwrBoundRatio, int pwrType,
        size_t r5, size_t r4, size_t r3, size_t r2, size_t r1);
```

**Input:**

| | |
|---|---|
| **dataType** | the indicator that indicates the data type (two options: either *SZ_FLOAT* or *SZ_DOUBLE*) |
| **data** | the variable that contains the data to be compressed. (**Current version only supports "double precision" data**) |
| **compressed_bytes** | the address that contains the compressed bytes |
| **outSize** | the data stream size (in bytes) after compression. |
| **errBoundMode** | Error Bound Mode (e.g., ABS) |
| **absErrBound** | **absolute error bound** |
| **relBoundRatio** | a bound ratio for value range based relative error bound |
| **pwrBoundRatio** | a bound ratio for point wise based relative error bound |
| **pwrType** | the way of computing a statistical value (MIN, AVG, or MAX) for the block-based strategy to do the point-wise relative error bound data compression. This is only valid for 2+ dimensional data. |
| **r5** | size of dimension 5 (the **slowest** changing dimension) |
| **r4** | size of dimension 4 |
| **r3** | size of dimension 3 |
| **r2** | size of dimension 2 |
| **r1** | size of dimension 1 (the **fastest** changing dimension) |

**Return:** Compressed data stream (in the form of bytes)

**Usage tips:** The dimension of the variable is determined based on the five dimension parameters (r5, r4, r3, r2, and r1). For instance, if the variable is a 2D array (M X N), then r5=0, r4=0, r3=0, r2=M, and r1=N. If the variable to protect is a 4D array, then only r5 is set to 0. (See test_compress.c for details).

**(c)** ***SZ_decompress***

Decompress/recover the data. Two options, as listed below.

**Synopsis:**

```
void *SZ_decompress(int dataType, char *bytes, size_t byteLength,
                           size_t r5, size_t r4, size_t r3, size_t r2, size_t r1);
size_t SZ_decompress_args(int dataType, char *bytes, size_t byteLength,
                           void* decompressed_array,
                           size_t r5, size_t r4, size_t r3, size_t r2, size_t r1);
```

**Input:**

| | |
|---|---|
| **dataType** | the indicator to indicate the data type (either *SZ_FLOAT* or *SZ_DOUBLE*) |
| **bytes** | the compressed data stream to be decompressed |
| **byteLength** | length of the compressed data stream |
| **decompressed_array** | the address to store decompressed data |
| **r5** | size of dimension 5 (the **slowest** changing dimension) |
| **r4** | size of dimension 4 |
| **r3** | size of dimension 3 |
| **r2** | size of dimension 2 |
| **r1** | size of dimension 1 (the **fastest** changing dimension) |

**Return:** the recovered data array decompressed from the compressed bytes.

**(d)** ***SZ_batchAddVar***

Register/add a variable (denoted by *var*) to be compressed with other variables together in a batch way.

**Synopsis:**

```
void SZ_batchAddVar(char* varName, int dataType, void* var,
        int errBoundMode, double absErrBound, double relBoundRatio,
        size_t r5, size_t r4, size_t r3, size_t r2, size_t r1);
```

**(e)** ***SZ_batchDelVar***

Deregister/delete a variable (denoted by *var*) from the list of registered variables, that are to be compressed with other variables together in a batch way.

**Synopsis:**

```
int SZ_batchDelVar(char* varName);
```

**Input:**

| | |
|---|---|
| **varName** | the name of variable used in the registration. |

**Return:** 0: success or 1: no corresponding variable is found based on varName.

**(f)** ***SZ_batch_compress***

Compress the data in a batch way: all of the registered variable data will be compressed together (The benefit is improvement of compression factor).

**Synopsis:**

```
char* SZ_batch_compress(size_t *outSize);
```

**Input:**

| | |
|---|---|
| **outSize** | the data stream size (in bytes) after compression. |

**Return:** the compressed stream.

**(g)** ***SZ_batch_decompress***

Decompress the batch-compressed stream.

**Synopsis:**

```
SZ_VarSet* SZ_batch_decompress (char* compressedStream,
```

**size_t compressedLength);**

**Input:**

| | |
|---|---|
| **compressedStream** | the compressed stream |
| **compressedLength** | the length of the compressed stream (in byte) |

**Return:** The data structure containing the decompressed data with multiple variables. See VarSet.h for more details. The global SZ_VarSet is defined in sz.h: SZ_VarSet* sz_varset.

**(h)** ***SZ_Finalize***

Release the memory and compression environment.

**Synopsis:** **int SZ_Finalize();**

**Input:** none.

**Return:** none.

## 7.2 Compression/Decompression by Fortran Interfaces

**Interfaces**:

**(a)** ***SZ_Init***

Initialize the SZ compressor. SZ_Init() just needs to be called only **once** before performing multiple compressions for different variables (data arrays).

**Synopsis**: **SZ_Init(configFilePath, ierr);**

**Input:**

| | |
|---|---|
| **configFilePath** | configuration file path (e.g., sz.config)<br>**CHARACTER(len=32) :: configFilePath** |

**Output:**

| | |
|---|---|
| **ierr** | successful (0) or failed (1)<br>**INTEGER(Kind=4) :: ierr** |

**(b)** ***SZ_Compress***

Compress the floating-point data array. Two types of interfaces are provided, as shown below. For the first one, the three important control parameters (errBoundMode, absErrBound, and relBoundRatio) will be given by the configuration file sz.config. For the second one, the three control parameters will be passed using arguments, so in this case, the parameter settings in the sz.config will be ignored.

**Synopsis A**:

**SZ_compress(data, bytes, outSize);**

**Input:**

| | |
|---|---|
| **data** | the data array to be compressed<br>(the data here is a floating-point data array with up to 5 dimensions. For example, “REAL(KIND=8), DIMENSION(:,:,:) :: *data*” indicates a 3D double-precision array, where *data* refers to the array variable.) |

**Output:**

| | |
|---|---|
| **bytes** | the byte stream generated after the compression |

| | |
|---|---|
| | **INTEGER(kind=1), DIMENSION(:), allocatable :: bytes** |
| **outsize** | the size (in bytes) of the byte stream<br>**INTEGER(kind=4) :: OutSize** |

**Synopsis B**:

**SZ_Compress (data, bytes, outSize,**
**errBoundMode, absErrBound, relBoundRatio);**

**Input:**

| | |
|---|---|
| **data** | the data array to be compressed<br>(the data here is a floating-point data array with up to 5 dimensions. For example, "REAL(KIND=8), DIMENSION(:,:,:) :: *data*" indicates a 3D double-precision array, where *data* refers to the array variable.) |
| **errBoundMode** | the error bound mode.<br>Four options: ABS, REL, ABS_AND_REL, ABS_OR_REL<br>**INTEGER(kind=4) :: ErrBoundMode** |
| **absErrBound** | absolute error bound<br>**REAL(kind=4 or 8) :: absErrBound** |
| **relBoundRatio** | relative bound ratio<br>**REAL(kind=4 or 8) :: relBoundRatio**<br>(Details about error bound mode, absolute error bound, and relative bound ratio can be found in Section 3.1) |

**Output:**

| | |
|---|---|
| **bytes** | the byte stream generated after the compression<br>**INTEGER(kind=1), DIMENSION(:), allocatable :: bytes** |
| **outsize** | the size (in bytes) of the byte stream<br>**INTEGER(kind=4) :: OutSize** |

**(c) *SZ_Decompress***

Decompress/recover the data

**Synopsis:**

**SZ_Decompress(bytes, data, [r1,r2,…])**

**Input:**

| | |
|---|---|
| **bytes** | the compressed data stream to be decompressed<br>**INTEGER(kind=1), DIMENSION(:) :: Bytes** |
| **data** | length of the compressed data stream<br>**REAL(KIND=4 or 8), DIMENSION(:,:,…:,:), allocatable :: data** |
| **r1** | size of dimension 1 (the **fastest** changing dimension) |
| **r2** | size of dimension 2 |
| **r3** | size of dimension 3 |
| **r4** | size of dimension 4 |
| **r5** | size of dimension 5 (the **slowest** changing dimension)<br>**INTEGER(kind=4) :: r1[, r2, r3, r4, r5]** |

**Usage tips:** SZ_Decompress supports the decompression of the array with at most 5 dimensions. The dimension sizes (such as r1, r2, ….) are supposed to be provided. For example, in order to decompress a binary stream whose original data is a 3D array (r3=10,r2=8,r1=8), the function is like "SZ_Decompress(bytes, data, 8, 8, 10).

**(d)** ***SZ_BatchAddVar***

Register/add a data variable (denoted by *var*) to be compressed with other variables together in a batch way.

**Synopsis:**

**void SZ_batchAddVar(varName, var,**
**errBoundMode, absErrBound, relBoundRatio);**

| | |
|---|---|
| **varName** | the name of the variable to be registered/added<br>**CHARACTER(len=128) :: varName** |
| **var** | the variable/data to be registered/added |
| **errBoundMode** | the error bound mode.<br>Four options: ABS, REL, ABS_AND_REL, ABS_OR_REL<br>**INTEGER(kind=4) :: ErrBoundMode** |
| **absErrBound** | absolute error bound<br>**REAL(kind=4 or 8) :: absErrBound** |
| **relBoundRatio** | relative bound ratio<br>**REAL(kind=4 or 8) :: relBoundRatio**<br>(Details about error bound mode, absolute error bound, and relative bound ratio can be found in Section 3.1. |

**(e)** ***SZ_BatchDelVar***

Deregister/delete a variable (denoted by *var*) from the list of registered variables, that are to be compressed with other variables together in a batch way.

**Synopsis:**

**void SZ_batchDelVar(varName, ierr);**

**Input:**

| | |
|---|---|
| **varName** | the name of variable used in the registration.<br>**CHARACTER(len=128) :: varName** |

**Output:**

| | |
|---|---|
| **ierr** | the output status (0: success or 1: no variable found)<br>**INTEGER(kind=4) :: ErrBoundMode** |

**Return:** 0: success or 1: no corresponding variable is found based on varName.

**(f)** ***SZ_Batch_Compress***

Compress the data in a batch way: all of the registered variable data will be compressed together (The benefit is improvement of compression factor).

**Synopsis:**

**void SZ_Batch_Compress(bytes, outSize)**

**Output:**

| | |
|---|---|
| **bytes** | the byte stream generated after the compression<br>**INTEGER(kind=1), DIMENSION(:), allocatable :: bytes** |
| **outsize** | the size (in bytes) of the byte stream |

**(g)** ***SZ_Batch_Decompress***

Decompress the batch-compressed stream.

**Synopsis:**

**void SZ_Batch_Decompress(bytes, outSize)**

**Output:**

| | | |
|---|---|---|
| **bytes** | the compressed data stream to be decompressed | **INTEGER(kind=1), DIMENSION(:) :: Bytes** |
| **outsize** | the size of the decompressed data stream | **INTEGER(kind=4) :: OutSize** |

**(h)** ***SZ_Finalize***

Release the memory and compression environment

**Synopsis:** **SZ_Finalize();**

**Input:** none.

**Return:** none.

## 8. Macros and data structures

**### sz.h**

**##Check version number**

#define SZ_VER_MAJOR 1

#define SZ_VER_MINOR 4

#define SZ_VER_BUILD 12

#define SZ_VER_REVISION 0

**##Check all the error bound modes**

#define ABS 0

#define REL 1

#define ABS_AND_REL 2

#define ABS_OR_REL 3

#define PSNR 4

#define PW_REL 10

#define ABS_AND_PW_REL 11

#define ABS_OR_PW_REL 12

#define REL_AND_PW_REL 13

#define REL_OR_PW_REL 14

**##Check all the data types supported**

#define SZ_FLOAT 0

#define SZ_DOUBLE 1

#define SZ_UINT8 2

#define SZ_INT8 3

#define SZ_UINT16 4

```
#define SZ_INT16 5
#define SZ_UINT32 6
#define SZ_INT32 7
#define SZ_UINT64 8
#define SZ_INT64 9
```

**##SZ compression mode**

```
#define SZ_BEST_SPEED 0
#define SZ_BEST_COMPRESSION 1
```

**##Check the metadata**

```
typedef struct sz_metadata
{
	int versionNumber[3]; //only used for checking the version by calling SZ_GetMetaData()
	int isConstant; //only used for checking if the data are constant values by calling SZ_GetMetaData()
	int isLossless; //only used for checking if the data compression was lossless, used only by calling SZ_GetMetaData()
	int sizeType; //only used for checking whether the size type is "int" or "long" in the compression, used only by calling SZ_GetMetaData()
	size_t dataSeriesLength;
	struct sz_params* conf_params;
} sz_metadata;
```

# 9 Test cases

example/testdouble_compress.c
example/testdouble_decompress.c
example/testfloat_compress.c
example/testfloat_decompress.c
example/testfloat_batch_compress.c
example/testdouble_batch_compress.c
example/testdouble_compress.f90
example/testdouble/decompress.f90

# 10 Optional preprocessing compression model

The executable **sz** also provides two more options, allowing users to do a preprocessing step for the compression, either specifying the wavelet transform (by –W) or using the Tucker tensor decomposition (by –T).

If the user adopts –W option, the SZ compressor will perform a wavelet transform on the given data set, and then conduct the remaining compression steps (including data prediction, quantization, etc.). In the decompression, the SZ compressor will perform the classic decompression steps (quantization + prediction), and then perform the reverse wavelet transform to recover the data finally.

If the user adopts –T option, the SZ compressor will do the Tucker tensor decomposition on the given data set. Unlike –W, there will be no further compression steps after getting the Tucker tensor decomposition results (cores and other matrices), because the output cores and matrices are already highly non-correlated inside, such that further compression will not improve the compression factor clearly. SZ adopts TuckerMPI package to perform the optional tucker tensor decomposition. The compressed data (i.e., output of TuckerMPI) will be put in a directory named “compressed” under the current command execution directory. The decompressed/reconstructed file is always named “tucker-decompress.out”. Note that the current version supports only “double” precision data because TuckerMPI doesn’t support single-precision data. The compression error bound of Tucker tensor decomposition is using absErrBound set in sz.config.

Note that in order to enable the wavelet transform functionality, you need to “./configure” with the option “--enable-gsl”, because our implementation depends on GSL. Specifically, you need to compile SZ as follows:
./configure --prefix=[The installation path] --enable-gsl
(The compilation will try to find GSL on your machine. If failed to find it, you can use --with-gsl-prefix to specify the installation path of the GSL. Details can be found by executing “./configure --help”.

As for enabling –T option, you need to download and install Sandia’s TuckerMPI package first, and then set the environment variable called TUCKERMPI_PATH to the building path of its package.

**Some examples about how to use –W and –T are shown below:**
*For Wavelet transform compression:*
[sdi@sdihost example]$ sz -z -c sz.config -i ~/Data/Hurrican-ISA/CLOUDf48_double.bin.dat -d -W -3 500 500 100
[sdi@sdihost example]$ sz -x -c sz.config -i ~/Data/Hurrican-ISA/CLOUDf48_double.bin.dat -d -s ~/Data/Hurrican-ISA/CLOUDf48_double.bin.dat.sz -a -W -3 500 500 100
*For Tucker tensor decomposition:*
[sdi@sdihost example]$ sz -z -c sz.config -i ~/Data/Hurrican-ISA/CLOUDf48_double.bin.dat -d -T -3 500 500 100
[sdi@sdihost example]$ sz -x -c sz.config -i ~/Data/Hurrican-ISA/CLOUDf48_double.bin.dat -d -a -T -3 500 500 100
(Note: The Tucker tensor decomposition does not require to input the compressed data files,

which were stored in ./compressed directory in the compression step)

## 11. Version history

The latest version (**version 1.4.12**) is the recommended one.

**Version** **New features**

**SZ 0.2-0.4** Compression ratio is the same as SZ 0.5. The key difference is different implementation ways, such that SZ 0.5 is much faster than SZ 0.2-0.4.

**SZ 0.5.1** Support version checking

**SZ 0.5.2** finer compression granularity for unpredictable data, and also remove redundant Java storage bytes

**SZ 0.5.3** Integrate with the dynamic segmentation support

**SZ 0.5.4** Gzip_mode: defaut --> fast_mode ; Support reserved value

**SZ 0.5.5** runtime memory is shrinked (by changing int xxx to byte xxx in the codes)
The bug that writing decompressed data may encounter exceptions is fixed.
Memory leaking bug for ppc architecture is fixed.

**SZ 0.5.6** improve compression ratio for some cases (when the values in some segementation are always the same, this segment will be merged forward)

**SZ 0.5.7** improve the decompression speed for some cases

**SZ 0.5.8** Refine the leading-zero granularity (change it from byte to bits based on the distribution). For example, in SZ0.5.7, the leading-zero is always in bytes, 0, 1, 2, or 3. In **SZ0.5.8** The leading-zero part could be xxxx xxxx xx xx xx xx xxxx xxxx (where each x means a bit in the leading-zero part)

**SZ 0.5.9** optimize the offset by using simple right-shifting method. Experiments show that this cannot improve compression ratio actually, because simple right-shifting actually make each data be multiplied by 2^{-k}, where k is # right-shifting bits. The pros is to save bits because of more leading-zero bytes, but the cons is much more required bits to save. A good solution is SZ 0.5.10!

**SZ 0.5.10** optimze the offset by using the optimized formula of computing the median_value based on optimized right-shifting method. Anyway, SZ0.5.10 improves compression ratio a lot for hard-to-compress datasets. (Hard-to-compress datasets refer to the cases whose compression ratios are usually very limited)

**SZ 0.5.11** In a very few cases, SZ 0.5.10 cannot guarantee the error-bounds to a certain user-specified level. For example, when absolute error bound = 1E-6, the maximum decompression error may be 0.01(>>1E-6) because of the huge value range even in the optimized segments such that the normalized data cannot reach the required precision even stoaring all of the 64 or 32 mantissa bits. SZ 0.5.11 fixed the problem well, with degraded compression ratio less than 1%.

**SZ 0.5.12** A parameter setting called "offset" is added to the configuration file sz.config. The value of offset is an integer in [1,7]. Generally, we recommend offset=2 or 3, while we also find that some other settings (such as offset=7) may lead to better compression ratios in some cases. How to automize/optimize the selection of offset value would be the future work. In addition, the compression speed is improved, by replacing java List by array implementation in the code.

**SZ 0.5.13** Compression performance is improved, by replacing some class instances in the source code by primitive data type implementation.

**SZ 0.5.14** fixed a design bug, which improves the compression ratio further.

**SZ 0.5.15** improved the compression ratio for single-precision data compression, by tuning the offset.

The version 0.x were all coded in Java, and C/Fortran interfaces were provided by using JNI and C/Fortran wrapper. SZ 1.0 is coded in C purely.

**SZ 1.0** Pure C version. In this version, the users don't need to install JDK and make the relative configurations any more. It provides dataEndienType in the sz.config file, so it can be used to compress the data file which was generated on different endian-type systems.

**SZ 1.1** batch_compression function is added to this version. Compression performance is improved slightly due to for(;;) being replaced by memcpy() somewhere.

**SZ 1.2** The compression ratio is improved by 30%-50% in most of datasets (especially for relatively-hard-to-compress ones), and the compression time is reduced by about 10%, compared to SZ1.1.

**SZ 1.3** The compression ratio and speed are improved further compared with SZ1.2, by using 256 quantization intervals and multi-dimensional prediction.

**SZ 1.4** Use 65536 intervals

**SZ 1.4.2** Extending the number of intervals from 255 to 65536, by tailoring/reimplementing the Huffman encoding by ourselves.

**SZ 1.4.3** Add the intervals_count to the configuration file (sz.config), allowing users to control it.

**SZ 1.4.4** Remove segmentation step quantization_intervals

**SZ 1.4.5** Optimize the number of intervals: the # intervals will be automatically optimized before the compression if quantization_itnervals is set to 0.

**SZ 1.4.6-beta** Three compression modes are provided (SZ_BEST_SPEED, SZ_BEST_COMPRESSION, SZ_DEFAULT_COMPRESSION), the maximum # quantization intervals is 65536.

**SZ 1.4.7-beta** Fix some mem leakage bugs. Fix the bugs about memory crash or segmentation faults when the number of data points is pretty large. Fix the sementation fault bug happening when the data size is super small for 1D array. Fix the error bound may not be guaranteed in some cases.

**SZ 1.4.9-beta** Support point-wise relative error bound setting, and optional Fortran compilation.

**SZ 1.4.9.1-beta** Fix the bug in the fortran interface about SZ_batch_compression

SZ 1.4.9.2-beta Update the user guide by describing how to optimize the compression quality on demand.

**SZ 1.4.9.3-beta** Fix the sementation fault bug happening when the data size is super small for 2D array and 3D array. (Specifically, when the data size is very small while the error bound is set to very small too, the Huffman tree overhead will be relatively huge such that the compressed size may exceed the original data size, leading to segmentation fault when further compressing it by the last lossless compression step. Solution: In this case, the data will be compressed by Zlib for simplicity, with no compression errors).

**SZ 1.4.10-beta** (1) Support direct sub-block data compression; (2) Support compression of large data file directly (i.e., the number of data points could be up to as large as LONG size, unlike the previous version that can only compress 2^{32} data points each time): that is, int nbEle --> size_t nbEle; (3) separate the internel functions from the sz.h;

**SZ 1.4.11-beta** (1) Support HDF5. (2) Support integer data compression. (3) Provide optional wavelet transform as a preprocessing step in SZ and an optional Tucker tensor decomposition.

**SZ 1.4.11** (1) This is a stable version which have went through a long period test. (2) Fix a small bug (the maximum compression error may be slightly greater than error bound in some cases); (3) Support integer compression (for all types of integers); (4) Support HDF5-SZ for all types of integers; (5) Support getting the metadata from a given compressed data file (by using SZ_getMetaData) and printing the metadata (by SZ_printMetaData) with -p option of the executable command ("sz"); (6) Change libsz.a to libSZ.a in case of conflict with szip (note that szip has already used libsz.a).

**SZ 1.4.12** (1) Support thresholding-based strategy for 1D data compression based on point-wise relative error bound. (In order to test it, please select errBoundMode = PW_REL, and set the point-wise relative error bound using the parameter pw_relBoundRatio in the sz.config.) For other dimensions of data, point-wise relative error based compression is using block-based strategy (see our DRBSD-2 paper for details) (2) fix the bug in the callZlib.c (previously, segmentation fault might happen when using best_compression mode). (3) Fix a small bug that happened when the data size is extremely huge (nbEle>4G) and the compression mode is SZ_BEST_COMPRSSION. Specifically, the previous call to zlib functions has one potential bug that may lead to segmentation fault, which has been fixed.

# 11. Q&A and Trouble shooting

**1. Do I need to call SZ_init() every time I compress a variable in the program?**

**Answer:** No. In the progress, SZ_init() is to initialize the compression, and it just needs to be called once, and thereafter you can always compress different variables using the compression/decompression functions on demand, until SZ_finalize() is called. There are two ways to initialize the compression environment, please read Section 6 for details.

**2. If I want to use SZ_compress_args() function and specify the errorBoundMode and bounds at run time instead of using the sz.config, do I need to call SZ_init()?**

**Answer:** It depends. In fact, sz.config has some important parameter settings, e.g., data_endian_type (little or big). You can also set these parameters manually in your code or use the default setting or using sz_params data structure. Please check sz.h and conf.c for details.

**3. How to deal with "Error: The input file or data stream is not in SZ format!"?**

**Answer:** This error is because the input file or data stream used to be decompressed is probably not the byte steam compressed/generated by the SZ. Please use the compressed file (such as data.sz) in the decompression.

**4. How to switch on/off the Fortran compilation?**

**Answer:**

Do the following steps to switch on the compilation for Fortran users.

./configure –prefix=[install_dir] --enable-fortran

(The compilation without the option "--enable-fortran" is without Fortran compile by default)

**5. Do I need to initialize the environment by SZ_Init() for decompression?**

**Answer:**

No. SZ_Init() is only required by compression step.

**6. What is the order of the dimension arguments supposed to be set in the interface?**

**Answer:**

For C interface, it is following the C style. For example, the matrix r3xr2xr1, r3 is the slowest changing dimension, and r1 is the fastest.

**7. How to optimize select the parameters for optimizing the compression quality for my data sets?**

The most important parameters that may affect the compression speed and compression ratio are quantization_intervals, max_quant_intervals, szMode and gzipMode. Details are described in Section 4.

**<END>**