Z-checker (ZC): A lossy compression assessment tool

User Guide (Version 0.2.0)

Mathematics and Computer Science (MCS)
Argonne National Laboratory,
Exascale Computing Project (ECP)
Contact: Sheng Di (sdi1@anl.gov)
Developers: Sheng Di, Dingwen Tao, Hanqi Guo, Julie Bessac
Advisor: Franck Cappello
December, 2018

Table of Contents

1. Brief description

Due to vast volume of data being produced by today's scientific simulations and experiments, lossy data compressor allowing user-controlled loss of accuracy during the compression is a relevant solution for significantly reducing the data size. However, lossy compressor developers and users are missing a tool to explore the features of scientific data sets and understand the data alteration after compression in a systematic and reliable way. To address this gap, we design and implement a generic framework, called Z-checker. On the one hand, Z-checker combines a battery of data analysis components relevant for data compression. On the other, Z-checker is implemented as an open-source community tool for which users and developers can contribute and add new analysis components based on their additional analysis demand.

In this user guide, we present the design framework of Z-checker, in which we integrated evaluation metrics proposed in prior work as well as other analysis required by lossy compressor developers and users. For lossy compressor developers, Z-checker can be used to characterize critical properties (such as entropy, distribution, power spectrum, principle component analysis, auto-correlation) of any data set to improve compression strategies. For lossy compression users, Z-checker can obtain the compression quality (compression ratio, bit-rate), provide various global distortion analysis comparing the original data with the decompressed data (PSNR, SNR, normalized MSE, rate-distortion, rate-compression error, spectral, distribution, derivatives) and statistical analysis of the compression error (maximum/minimum/average error, autocorrelation, distribution of errors). Z-Checker can perform the analysis with either course (throughout the whole data set) or fine granularity (by user defined blocks), such that the users/developers can select the best-fit, adaptive compressors for different parts of the data set. Z-checker features a visualization interface displaying all analysis results in addition to some basic views of the data sets such as time series.

2. Design Framework

Z-checker has three important features. (1) Z-checker can be used to explore the properties of original data sets for the purpose of data analytics or improvement of lossy compression algorithms. (2) Z-checker is integrated with a rich set of evaluation algorithms and assessment functions for selecting best-fit lossy compressors for specific data sets. (3) Zchecker features both static data visualization scripts and an interactive visualization system, which can generate visual results on demand.

The design architecture of Z-checker is presented in Figure 1, which involves three critical parts, including user interface, processing module, and data module.

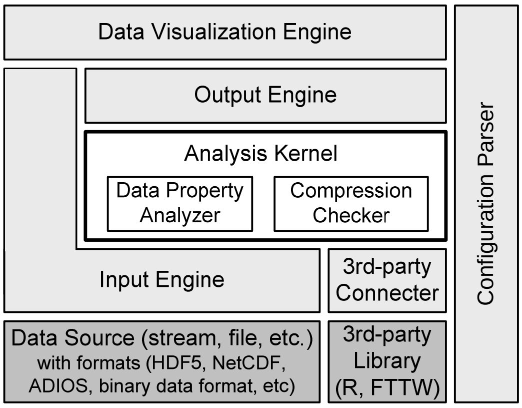


Fig. 1 Design Framework of Z-checker

- **User interface** includes three key engines, namely input engine, output engine, and data visualization engine, as shown in the light-gray rectangles in Figure 1. They are in charge of reading the floating-point data stream (either original data or compressed bytes), dumping the analyzed data to disks/PFS, and plotting data figures for the visualization of analysis results. The Data Visualization Engine also provides the interactive mode through a web-browser interface.
- **Processing module**, in the whole framework, is the core module, which includes Analysis Kernel and Configuration Parser. The former is responsible for performing the critical analysis, and the latter is in charge of parsing user's analysis requirement (such as specifying input file path, specifying compression command/executable, and customizing the analysis metrics on demand). Specifically, the analysis kernel is composed of two critical sub-modules, data property analyzer and compression checker, which are responsible for exploring data properties based on original data sets and analyzing the compression results with specified lossy compressors, to be discussed later in more details.
- **Data source module** (shown as dark-gray box in the figure) is the bottom layer in the whole framework, and it represents the data source (such as data stream produced by scientific applications at runtime or the data files stored in the disks).

3. Installation of Z-checker

We provide two alternative ways to install Z-checker, stand-alone installation or z-checker-installer.
The former is installing Z-checker individually, without installing third-party dependencies or compression libraries. The z-checker-installer is recommended for generic users, because it will check if the required third-party libraries are already installed. If not, it will download and install the dependencies automatically before installing Z-checker. It also includes some state-of-the-art lossy compressors such as SZ and ZFP, and also provides a set of scripts allowing users to generate the compression report by running only one command.

3.1 Stand-alone installation

The Z-checker library/tool can be cloned by the following command:
git clone https://github.com/CODARcode/z-checker
or downloaded from http://www.mcs.anl.gov/~shdi/download/z-checker-0.1.1.tar.gz
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 or [DOWNLOAD_PACKAGE]/examples.
The static library (.a files) and shared library (.so files) can be found in [INSTALL_DIR]/lib.
The key executables are described in Section 4.

Note: You can add plugin codes when doing the configuration step of the installation by "./configure" with different options, as shown below. Different options (--enable-xxx or --with-xxx-prefix) can be combined. "--prefix=[install_dir]" is still strongly recommended.

- **Support MPI library**: ./configure --prefix=[install_dir] --enable-mpi
  (You need to install mpi library such as mpich before hand.)
- **Support NetCDFlibrary** : usage can be found in the subdirectory NetCDFReader/
  ./configure --prefix=[install_dir] --enable-netcdf --with-netcdf-prefix=DIR
- **Support HDF5 library**: details can be found in HDF5Reader/
  ./configure --prefix=[install_dir] --enable-hdf5
  (You need to install HDF5 and set its environment variables according to HDF5 guide before hand.)
- **Support FFTW3 library**: computation of 3D auto correlation requires FFTW3
  ./configure --prefix=[indall_dir] --enable-fftw3 --with-fftw3-prefix=DIR
- **Support R language/library**: the functions coded in R script can be executed in the data analysis and compression analysis. In particular, SSIM function is coded in R, so it

requires R library in Z-checker.
./configure --prefix=[index_dir] --enable-r --with-r-prefix=DIR

In addition, you can also use --with-xxx-include-path and --with-xxx-lib-path to specify the header directory and lib directory, respectively, instead of --with-xxx-prefix. More options can be found by executing "./configure –help".

3.2 One-command installation (using Z-checker-installer)

1. git clone https://github.com/CODARcode/z-checker-installer
2. Run z-checker-install.sh will download latexmk, gnuplot, Z-checker, ZFP, and SZ and install them one by one automatically, and then add the patches to let ZFP and SZ fit for Z-checker.

(*Note: in the future, If you want to update your package based on your already installed z-checker-installer, you can execute z-checker-update.sh for simplicity, instead of rerunning z-checker-install.sh*)

4. Testing and Usage (single process version)

4.1 Offline testing based on stand-alone installation

The executables are in [INSTALL_DIR]/bin and [DOWNLOAD_PACKAGE]/examples.

After the installation described above, you can test the sample data sets in [DOWNLOAD_PACKAGE]/examples/testdata/x86. You can also download more data sets, CESM-ATM and MD-simulation (exaalt), which are available only for the purpose of compression research.

- CESM-ATM: http://www.mcs.anl.gov/~shdi/download/CESM-ATM-tylor.tar.gz
- MD-simulation (exaalt): http://www.mcs.anl.gov/~shdi/download/exaalt-dataset.tar.gz

You can compress the data files using some lossy compressor such as SZ or ZFP, and also generate the decompressed data files. Then, you are ready to use our provided executables to do the compression quality analysis. For the convenience of your testing, we already generated some compressed and decompressed climate simulation data files, downloadable here: http://www.mcs.anl.gov/~shdi/download/runOfflineCase_testdata.tar.gz

We provide several executables (including analyzeDataProperty, compareDataSets, generateGNUPlot and generateReport) to do the standalone assessment based on the original data files and decompressed data files.
Note that we also provide an executable (called runOfflineCase) to wrap the

analyzeDataProperty and compareDataSets together for the convenience of the usage.
In the following, we first show an example, and then describe the details thereafter.

Quick Start:

This example describes how to generate data property analysis results and compression results, give original data files (in binary), compressed files and decompressed files.
- Use createOfflineCase.sh to create a use-case directory, which will contain all the executables. Then, do the following steps.

//Preparing the testing data

.1. Download the testing data from here:

http://www.mcs.anl.gov/~shdi/download/runOfflineCase_testdata.tar.gz

(Tips: the testing data package contains two original data files and their corresponding SZ-compressed files and decompressed files based on different error bounds)

.2. tar -xzvf runOfflineCase_testdata.tar.gz

//Generate the property analysis results and compression results

.3. ./createOfflineCase.sh testcase1

.4. cd testcase1

.5. ./runOfflineCase -C varCmpr.inf -A -N sz

(Tips: More options can be shown by executing ./runOfflineCase without any input options; in the above example, we set the compress name as "sz" because the compressed/decompressed data files were generated by sz.)

//Generate figures based on the property/compression results in form of GNUPlot

.6. Edit zc.config as follows:

compressors = sz:[the absolute path of the directory of the test case] (i.e., echo `pwd`)

(Tips: ***sz*** here refers to the compressor's name. You can replace it by your own compressor)

comparisonCases = sz(1E-3)

.7. ./generateGNUPlot zc.config

Then, you can find rate-distortion eps files generated in the current directory.
More information can be found in the doc/userguide.pdf
(Tips: You can run the executables or scripts without any inputs to see the help information)

Description of executables:

- runOfflineCase

Usage: runOfflineCase <options>

Options:

* input information:

-N <compressor name>: the name of the compressor

-C <information file> : the file containing the data information

* analysis options:

-A : perform the full analysis of the compression results

-a <metric> : perform quick analysis for specific metric

*metric options: (including all variables)

```
            cr    : compression ratio (min, avg, max)
            err : compression error (min, avg, max)
            full: complete information listed as above
* metadata options:
      -n : print number of variables
      -m : print the names of variables
      -l : list complete information of all variables
      -p : print all the precisions used in the analysis
Examples:
      runOfflineCase -C varCmpr.inf -l
      runOfflineCase -C varCmpr.inf -m
      runOfflineCase -C varCmpr.inf -A
      runOfflineCase -C varCmpr.inf -a err
      runOfflineCase -C varCmpr.inf -a cr
```

As for the information file, we provide a sample varCmpr.inf, as shown below. All parameters in this file should be set manually, including compression/decompression time. If you don't care about compression time and decompression time, you can remove these two parameters, whose values will be set to 0 by default.

- ori_data: the information of the original raw data, including "name of variable", "endian type", "data type", "dimension", "file path".
- prec: precision of the compression, e.g., error bound.
- cpr_time/dec_time: compression time and decompression time (in seconds).
- cpr_data: compressed data file's path
- dec_data: decompressed data file's path (in binary format)

```
#RscriptPath = /home/sdi/z-checker-0.1-online/R/test/data_analysis_script.R
#information about the variables and compression demands
ori_data=CLDHGH_1_1800_3600:LITTLE_ENDIAN:FLOAT:1800x3600:../runOfflineCa
se_testdata/CLDHGH_1_1800_3600.dat
prec=1E-3
cpr_time=0.2
dec_time=0.1
cpr_data=../runOfflineCase_testdata/CLDHGH_1_1800_3600_1E-3.dat.sz
dec_data=../runOfflineCase_testdata/CLDHGH_1_1800_3600_1E-3.dat.sz.out
prec=1E-4
cpr_time=0.4
dec_time=0.2
cpr_data=../runOfflineCase_testdata/CLDHGH_1_1800_3600_1E-4.dat.sz
dec_data=../runOfflineCase_testdata/CLDHGH_1_1800_3600_1E-4.dat.sz.out
prec=1E-5
cpr_time=0.8
dec_time=0.4
cpr_data=../runOfflineCase_testdata/CLDHGH_1_1800_3600_1E-5.dat.sz
dec_data=../runOfflineCase_testdata/CLDHGH_1_1800_3600_1E-5.dat.sz.out
```

```
prec=1E-6
cpr_time=1.2
dec_time=0.6
cpr_data=../runOfflineCase_testdata/CLDHGH_1_1800_3600_1E-6.dat.sz
dec_data=../runOfflineCase_testdata/CLDHGH_1_1800_3600_1E-6.dat.sz.out

ori_data=CLDLOW_1_1800_3600:LITTLE_ENDIAN:FLOAT:1800x3600:../runOfflineCa
se_testdata/CLDLOW_1_1800_3600.dat
prec=1E-3
cpr_time=0.2
dec_time=0.1
cpr_data=../runOfflineCase_testdata/CLDLOW_1_1800_3600_1E-3.dat.sz
dec_data=../runOfflineCase_testdata/CLDLOW_1_1800_3600_1E-3.dat.sz.out
prec=1E-4
cpr_time=0.4
dec_time=0.2
cpr_data=../runOfflineCase_testdata/CLDLOW_1_1800_3600_1E-4.dat.sz
dec_data=../runOfflineCase_testdata/CLDLOW_1_1800_3600_1E-4.dat.sz.out
prec=1E-5
cpr_time=0.8
dec_time=0.4
cpr_data=../runOfflineCase_testdata/CLDLOW_1_1800_3600_1E-5.dat.sz
dec_data=../runOfflineCase_testdata/CLDLOW_1_1800_3600_1E-5.dat.sz.out
```

- analyzeDataProperty (or analyzeDataProperty.sh):

Usage: ./analyzeDataProperty.sh [datatype (-f or -d)] [data directory] [dimension sizes....]

Example: ./analyzeDataProperty.sh -f /home/shdi/CESM-testdata/1800x3600 3600 1800

Description: The analysis results will be put in the directory called dataProperties.

In particular,

- [variable_name].fft keeps the spectrum information (generated by FFT), including real part and imaginary part.
- [variable_name].fft.amp contains the amplitude of the spectrum data.

Notice: You need to set *checkingStatus* to ANALYZE_DATA and set *executionMode* to OFFLINE in the configuration file before running the analyzeDataProperty command.

- compareDataSets (or compareDataSet.sh)

Usage: compareDataSets [dataType -f or -d] [config_file] [compressionCase] [varName] [oriDataFilePath] [decDataFilePath] [dimension sizes...]

Example: compareDataSets -f zc.config SZ 8_8_128 testfloat_8_8_128.dat testfloat_8_8_128.dat.out 8 8 128

Description:

If you just want to compare the reconstructed data with the original data, you can use executable compareDataSets, as shown below.

compareDataSets [config_file] [oriDataFilePath] [decDataFilePath] [dimension sizes...]

Three output files (.cmp, .autocorr, and .dis) will be stored in the directory compareResults/:

- .cmp file keeps the general information about the compression results.
- .autocorr file keeps the auto correlation of the compression errors with different lags.
- .dis is about the PDF of compression errors.

Notice: You need to set *checkingStatus* to ANALYZE_DATA and set *executionMode* to OFFLINE in the configuration file before running the compareDataSets command.

- generateGNUPlot:

Usage: ./generateGNUPlot [config_file]

Example: ./generateGNUPlot zc.config

Description: generateGNUPlot will find compression results in the three directories: dataProperties, compressionResults and compareCompressors, and plot figures accordingly.

Notice: Before running the command generateGNUPlot, you need to make sure you already run analyzeDataProperty and comapreDataSets to generate the following two directories in the working space: ./dataProperties and compressionResults. You also need to set the *checkingStatus* and *executionMode* to COMPARE_COMPRESSOR and OFFLINE respectively. The parameter *compressors* are composed of a set of two-tuples – compressor_name:working_space_path. The working_space_path is the directory that contains the dataProperties directory and the compressionResults directory. The parameter comparisonCases includes the compression cases, which should also be consistent with the parameters used when running the compareDataSets command. We give some examples to further illustrate how to use generateGnuPlot correctly. In the following examples, suppose orig.raw stores the double-precision floating-point data, and approx.raw and approx.raw2 store reconstructed data generated by SZ compressor using error bound of 1E-2 and 1E-4, respectively.

Suppse you already generate the

Step 1: Generate dataProperties

Go to the working directory of Z-checker (suppose it is /home/sdi/Z-checker/examples), and then execute analyzeDataProperty as follows.

#./analyzeDataProperty **var1** -d zc.config /home/sdi/Data/orig.raw 8000000

Step 2: Generate compression results based on the decompressed data files.

Go to the working directories of all compressors and run the following command respectively. (As follows, we use SZ and ZFP as examples)

#cd **/home/sdi/sz_workspace**

#./compareDataSets -d zc.config **SZ(1E-2)** **var1** /home/sdi/Data/orig.raw /home/sdi/Data/approx_sz_1E-2.raw 8000000

#./compareDataSets -d zc.config **SZ(1E-4)** **var1** /home/sdi/Data/orig.raw /home/sdi/Data/approx_sz_1E-4.raw 8000000

#cd **/home/sdi/zfp_workspace**

#./compareDataSets -d zc.config **ZFP(1E-2)** **var1** /home/sdi/Data/orig.raw /home/sdi/Data/approx_sz_1E-2.raw 8000000

#./compareDataSets -d zc.config **ZFP(1E-4)** **var1** /home/sdi/Data/orig.raw

/home/sdi/Data/approx_sz_1E-4.raw 8000000
Notice: the variable name (here, var1) should be consistent in Step 1 and Step 2.
Step 3: Generate data figures
Go back to the working directory of Z-checker and execute generateGNUPlot:
cd /home/sdi/Z-checker/examples
./generateGNUPlot zc.config
Notice: before running the step 3 to generate the data figures, you need to make sure zc.config is configured correctly based on the compressor name and the working directory.
compressors = **SZ**:**/home/sdi/sz_workspace** **ZFP**:**/home/sdi/zfp_workspace**
compressionCases = **SZ(1E-2)**,**ZFP(1E-2)** **SZ(1E-4)**,**ZFP(1E-4)**
Then, you will find a set of data files generated in the current directory.

- generateReport (or generateReport.sh):

Usage: ./generateReport [config_file] [title of dataset]
Example: ./generateReport zc.config CESM-ATM-tylor-data
Description: generateReport assumes that the data properties and compression results are already generated in the three key directories, dataProperties, compressionResults and compareCompressors, and it will generate the figures and the pdf report based on them.

Notice:
The testing cases can be found in **[ZC_Download_Package]/examples**
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 [ZC_Package]/example/*test.sh* to test some examples (such as data property analysis code).
More tests related to compression analysis need to be performed with some data compressor such as SZ (https://collab.cels.anl.gov/display/ESR/SZ). More details will be described later.

4.2 One-command testing with Z-checker-installer

After the simple one-command installation described in Section 3.2, you can download the testing data sets, CESM-ATM and MD-simulation (exaalt).

- CESM-ATM: http://www.mcs.anl.gov/~shdi/download/CESM-ATM-tylor.tar.gz
- MD-simulation (exaalt): http://www.mcs.anl.gov/~shdi/download/exaalt-dataset.tar.gz

Then, you are ready to perform the compression quality checking by Z-checker.
You can generate compression results with SZ and ZFP using the following simple steps:
(Note: you have to run z-checker-install.sh to install the software before doing the following tests)

(1) Configure the error bound setting and comparison cases in errBounds.cfg.
(Example settings are already in the errBounds.cfg. You can change it based on your demand, such as the list of compression errors)

errBounds.cfg:

```
#sz_f_ERR_BOUNDS specifies the list of error bounds in the evaluation for the compression with SZ (fast mode).
sz_f_ERR_BOUNDS="1E-1 1E-2 1E-3 1E-4 1E-5"

#sz_d_ERR_BOUNDS specifies the list of error bounds in the evaluation for the compression with SZ (default mode).
sz_d_ERR_BOUNDS="1E-1 1E-2 1E-3 1E-4 1E-5"

#ZFP_ERR_BOUNDS specifies the list of error bounds for ZFP compression.
zfp_ERR_BOUNDS="1E-1 1E-2 1E-3 1E-4 1E-5 1E-6 1E-7"

#comparisonCases specifies the cases in which the different compressors will be compared with each other.
#For instance, "sz_f(1E-2),sz_d(1E-2),zfp(1E-2) sz_f(1E-4),sz_d(1E-4),zfp(1E-4)" means that the three compressors will be compared (w.r.t. compression ratio and PSNR) based on error bound = 1E-2 and 1E-4 respectively.
comparisonCases="sz_f(1E-2),sz_d(1E-2),zfp(1E-2) sz_f(1E-4),sz_d(1E-4),zfp(1E-4)"

#numOfErrorBoundCases specifies the number of error bound cases you want to present for each compression metric in the report.
#For instance, numOfErrorBoundCases=3 means that the compression metrics (such as distribution of compression errors), will only select three cases evenly in the list of error bounds.
#More specifically, if SZ_ERR_BOUNDS="1E-1 1E-2 1E-3 1E-4 1E-5", then only 1E-1, 1E-3, and 1E-5 will be used to plot the compression results for SZ.
#Without the setting numOfErrorBoundCases, the compression results with each error bound will be presented, leading to a very long report.
numOfErrorBoundCases="3"
```

(2) Create a new test-case, by executing "createNewZCCase.sh [test-case-name]". You need to replace [test-case-name] by a meaningful name.

For example:
[user@localhost z-checker-installer] ./createNewZCCase.sh CESM-ATM-tylor-data

Note: after creating a test-case, you should be able to find:

- two new directories [test-case-name]_fast, [test-case-name]_deft are generated in z-checker-installer/SZ ;

- one new directory called [test-case-name] is generated in z-checker-installer/zfp;
- one new directory called [test-case-name] is made in z-checker-installer/Z-checker

The new directories above contain the corresponding running scripts, which will be called later (by running runZCCase.sh) to perform the checking operations.

(3) Perform the checking by running the command "runZCCase.sh": runZCCase.sh [data_type] [error-bound-mode] [test-case-name] [data dir] [dimensions....].

There are two ways to specify the data files to be analyzed.

Option 1: Putting the data files with the same dimension sizes in the same directory. In the following example, we put five single-precision floating-point data files whose dimensions are all 1800x3600 in the directory /home/shdi/CESM-testdata/1800x3600.

[user@localhost z-checker-installer] ./runZCCase.sh -f REL CESM-ATM-tylor-data /home/shdi/CESM-testdata/1800x3600 3600 1800

Description:

- -f means that the data set is single-precision floating-point data.
- REL means that the error bounds used in the compression are based on value_range.
- CESM-ATM-tylor-data is the name of the testing case.
- [dimensions….] follows the column-major style. In the above example, the matrix in C programming code style is 1800x3600, so the dimensions sizes should be 3600 1800.

Option 2: Listing all the data files and corresponding information in a configuration file (e.g., varInfo.txt"), as follows:

#varInfo.txt:

CLDHGH:LITTLE_ENDIAN:FLOAT:1800x3600:/home/sdi/Data/CESM-ATM-tylor/1800x3600/CLDHGH_1_1800_3600.dat

CLDLOW:LITTLE_ENDIAN:FLOAT:1800x3600:/home/sdi/Data/CESM-ATM-tylor/1800x3600/CLDLOW_1_1800_3600.dat

FLDSC:LITTLE_ENDIAN:FLOAT:1800x3600:/home/sdi/Data/CESM-ATM-tylor/1800x3600/FLDSC_1_1800_3600.dat

FREQSH:LITTLE_ENDIAN:FLOAT:1800x3600:/home/sdi/Data/CESM-ATM-tylor/1800x3600/FREQSH_1_1800_3600.dat

PHIS:LITTLE_ENDIAN:FLOAT:1800x3600:/home/sdi/Data/CESM-ATM-tylor/1800x3600/PHIS_1_1800_3600.dat

Then, run the following command:

[user@localhost z-checker-installer] ./runZCCase.sh -f REL varInfo.txt

(4) Then, the report are generated in z-checker-installer/Z-checker/[test-case-name]/report. All the figures can be found in z-checker-installer/Z-checker/[test-case-name]/report/figs.

For more information, you can locate the specific results as follows:

- In z-checker-installer/Z-checker/[test-case-name]/dataProperties: The *.prop files contain the property analysis results about the data set, such that min value, max value and entropy value. The *.fft* files are the FFT analysis result about the data set. The *.autocorr* files contain the auto-correlation coefficient results with different lags.
- In z-checker-installer/Z-checker/[test-case-name]/compressionResults: The *.cmp

files contain the results about compression quality, such as compression ratio, maximum compression error, PSNR, SNR, MSE, compression time, and decompression time. The *.dis files are about the distribution of compression errors. The *.autocorr files contain the auto-correlation coefficients of the compression error with different lags.

- In z-checker-installer/Z-checker/[test-case-name]/compareCompressors: This directory contains the results about the comparison among different compressors. For example, rate-distortion_psnr_FREQSH_1_1800_3600.dat.txt presents rate-distortion figure (PSNR vs. bit-rate) for the variable FREQSH with the dimension size 1800x3600; the file crate_sz_f(1E-2)_sz_d(1E-2)_zfp(1E-2).txt compares the compression rate (i.e., compression speed) across three compression solutions with the same error bound 10^{-2}. cratio_sz_f(1E-2)_sz_d(1E-2)_zfp(1E-2).txt refers to compression ratio and drate_sz_f(1E-2)_sz_d(1E-2)_zfp(1E-2).txt indicates the decompression rate.

4.3 Integrate the a compressor’s results into z-checker-report

Up to now, we already have two ways to collect the compression results. In this section, we will introduce how to add a new compressor into Z-checker and how to remove a compressor from the Z-checker.

There are two ways to do so, based on how you want to assess the compressors. On the one hand, suppose you already have a set of compressed data files and **decompressed data files** corresponding to different error levels, based on one or more original raw data files. Then, you can perform an assessment based on the method mentioned in Section 4.1 to produce the compression results. On the other hand, you can also insert the Z-checker API functions (such as ZC_startCmpr(), ZC_endCmpr(), ZC_startDec(), and ZC_endDec()) into your compression codes (if the main() program of the compressor is editable), and use the z-checker-installer package to perform the overall assessment, which will run the compression/decompression code to generate the assessment results online and Z-checker’s executables to generate the report. In what follows, we will introduce the two methods, respectively.

4.3.1 Integration with compressed/decompressed data files

For the first way, the question is: can we generate a Z-checker-report (the .pdf file), based on the offline assessment results which were generated using given original data files and existing compressed/decompressed files, in order to compare with SZ and ZFP? The answer is yes.

Step 1: In the z-checker-installer package, create a usecase: “./createZCCase.sh [usecase_name]”. As for the CESM_ATM_Tylor dataset, for example, you can run this command: "./createZCCase.sh CESM_ATM_testcase", to generate the usecase.

Step 2: Run the script ./runZCCase.sh as mentioned in Section 4.2, based on either option 1 or option 2.

Step 3: Go to z-checker-installer/Z-checker/examples, and run "./runOfflineCase.sh [compression case name]". You need to replace "[compression case name]" by your own meaningful name, e.g., ./runOfflineCase CESM_Tom_Compressor. Then, you will find a new directory namely **CESM_Tom_Compressor** generated on the current directory.

Step 4: Do the 7 steps described in the "Quick Start" of Section 4.1 to generate .eps files. Specifically, you need to edit the configuration file "varCmpr.inf" based on the original data files and compressed/decompressed data files.
Note that the variables' names should be consistent with the names used when generating the z-checker-report by following Section 4.2 (3). For instance, if you choose the option 1 of Section 4.2 (3) to generate Z-checker report for SZ and ZFP, then variables' names are the file names excluding the extensions in the specified data directory. If you choose the option 2 of Section 4.2 (3), the variables' names are set in the configuration file varInfo.txt.

Step 5: Go to the corresponding workspace directory in the z-checker-installer/Z-checker directory, e.g., cd z-checker-installer/Z-checker/CESM_ATM_testcase, and then modify the following line in the configuration file "zc.config" as follows:

compressors = sz_f:../../SZ/CESM_ATM_testcase_fast sz_d:../../SZ/
CESM_ATM_testcase_deft zfp:../../zfp/CESM_ATM_testcase
[compression case name]:[the path of the compression case dir]

Specifically, in the above example, you need to append the following line "CESM_Tom_Compressor:../examples/CESM_Tom_Compressor" to the end of the line "compressors = …." as below:

compressors = sz_f:../../SZ/CESM_ATM_testcase_fast
sz_d:../../SZ/CESM_ATM_testcase_deft zfp:../../zfp/CESM_ATM_testcase
CESM_Tom_Compressor:../examples/CESM_Tom_Compressor

Step 6: Run "./generateReport.sh [name of testcase]". You can set the name of testcase as whatever name you want, which will be shown in the title of the z-checker-report.

4.3.2 Add/remove a compressor in Z-checker-installer

In addition to the offline integration of compression results into z-checker-report based on the existing compressed/decompressed data files, our package also allows integrating your compressor code into Z-checker, such that you can generate the z-checker report by simply running the runZCCase.sh command.

Add a new compressor.

Step 1. Make a monitoring program (e.g., called testfloat_CompDecomp.c) for your compressor. An example can be found in SZ/example/testfloat_CompDecomp.c, which is used for SZ compressor.
Step 2. Modify the manageCompressor.cfg based on the workspaceDir on your computer and directory containing the compiled executable monitoring program.
Step 3. Suppose the new compressor's name is zz and the compression mode is called 'best'; then, run the following command to add the new compressor:

./manageCompressor -a zz -m best -c manageCompressor.cfg

Step 4. Then, open errBounds.cfg to modify the error bounds for the new compressor; and also modify the comparison cases as follows (the compressor name 'zz_b' was set in manageCompressor.cfg):

comparisonCases="sz_f(1E-1),sz_d(1E-1),zfp(1E-1) sz_f(1E-2),sz_d(1E-2),zfp(1E-2)" →

comparisonCases="sz_f(1E-1),sz_d(1E-1),zfp(1E-1),**zz_b(1E-2)** sz_f(1E-2),sz_d(1E-2),zfp(1E-2),**zz_b(1E-2)**"

Step 5. Finally, create a test case like this: ./createZCCase.sh case_name
Step 6. Perform the assessment by runZCCase.sh.

Remove a compressor.
Some examples are shown below:

- Remove sz_f (sz fast mode):
  $ manageCompressor -d sz -m fast -c manageCompressor-sz-f.cfg
- Remove sz_d (sz fast mode):
  $ manageCompressor -d sz -m deft -c manageCompressor-sz-d.cfg
- Remove zfp:
  $ manageCompressor -d zfp -c manageCompressor-zfp.cfg

4.4. Manual testing without Z-checker-installer

<u>(This manual testing is only for the user who wants to play with z-checker's configuration file zc.config in details)</u>
Testing procedure:
Download a compression library, such as SZ (https://collab.cels.anl.gov/display/ESR/SZ), and then put the data compression monitoring interfaces before and after the compression/decompression function. Recompile the compression library. And then, perform the compression using some data set. The analysis results (such as compression rate, compression ratio, compression errors) will be stored in the directory called compareData/.

Example (with SZ):
1. Download SZ-1.4.9 from https://collab.cels.anl.gov/display/ESR/SZ
2. Install SZ package by the following steps:
(tar –xzvf sz-1.4.9-beta.tar.gz;cd sz-1.4.9.1-beta;./configure --prefix=[INSTALL_DIR];

make;make install)
3. Copy the testing code testfloat_CompDecomp.c from Z-checker's examples/ directory to SZ's example/ directory, and compile it by the following command (you need to replace the bolded parts by the installation paths on your own machine)
#compile_CompDecomp.sh
#!/bin/bash
SZPATH=**[SZ_INSTALL_DIR]**
ZCPATH=**[Z-Checker_INSTALL_DIR]**
SZFLAG="-I$SZPATH/include -I$ZCPATH/include $SZPATH/lib/libsz.a $SZPATH/lib/libzlib.a $ZCPATH/lib/libzc.a"
gcc -lm -g -o testfloat_CompDecomp testfloat_CompDecomp.c $SZFLAG
4. Run testfloat_CompDecomp:
./Testfloat_CompDecomp [sz_config_file] [zc_config_file] [compressor_name] [testcase] [absErrBound] [input_data_file_path] [dimension_sizes…..]
Example:
Set the compression mode in configuration file sz.config to *SZ_BEST_SPEED*, then:

absErrBound=1E-3
compressor_case=**sz1.4_f**($absErrBound) #sz1.4_f refers to sz1.4(best_speed_mode)
testcase=varName
datapath=testdata/x86/testfloat_8_8_128.dat

testfloat_CompDecomp sz.config zc.config "${compressor_case}" ${testcase} ${absErrBound} ${datapath} 8 8 128
(***Note***: You need to copy zc.config from Z-checker's examples/ directory to SZ's example/ before running the above command.)
5. After running testfloat_compDecomp, the compression results will be kept in compareData/ directory.
"sz(1E-3):varName.cmp" keeps the compression results, including compression time, decompression time, compression rate, decompression rate, PSNR, SNR, compression factor, maximum compression error, etc.
"sz(1E-3):varName.autocorr" keeps the auto-correlation values of the compression errors with different lags.
"sz(1E-3):varName.dis" keeps the distribution (PDF) of the compression errors.
6. Change the compression mode in sz.config to be *SZ_BEST_COMPRESSION* , and then rewind the above steps 4 with the compressor_case tagged as **sz1.4_c**, as shown below.
absErrBound=1E-3
compressor_case=**sz1.4_c**($absErrBound) #sz1.4_c is sz1.4(best_compression_mode)
testcase=varName
datapath=testdata/x86/testfloat_8_8_128.dat

testfloat_CompDecomp sz.config zc.config "${compressor_case}" ${testcase} ${absErrBound} ${datapath} 8 8 128

7. Now, we are ready to compare the compression results between the two “compressors”: sz1.4_f vs. sz1.4_c, as follows:
Go back to Z-checker’s examples/ directory, and then set the compressors as follows:
compressors = sz1.4_f:sz-1.4.9-beta/example sz1.4_c:sz-1.4.9-beta/example
comparisonCases = sz1.4_f(1E-3),sz1.4_c(1E-3)
Note: the format of the compressors string is [compressor_name]:[directory of running command]. The comparisonCases follows such as format: [compressor_case1],[compressor_case2] [compressor_case3],[compressor_case4] ……
8. Run the following command to generate the comparison figure files (in .eps format):
./generateGNUPlot zc.config
(Note: the eps files will be generated and put in the current directory).

4.4 Online Assessment with Z-checker

We provide an example code (called heatdis.c) in the directory examples/ to show how to do an online assessment in an MPI program. This example code simulates the diffusion of the heat on a board using Jacobi algorithm over time.
The key codes are shown below:

```
for (i = 0; i < ITER_TIMES; i++) {
	localerror = doWork(nbProcs, rank, M, nbLines, g, h);

	if(i%2==0) //control the compression frequency over time steps
	{
		sprintf(propName, "%s_%04d", varName, i); //make a name for the current target data property (variable_name)
		sprintf(cmprCaseName, "%s(1E-3)", compressorName); //name the compression case

		ZC_DataProperty* basicDataProperty = ZC_startCmpr(propName, ZC_DOUBLE, g, 0, 0, 0, nbLines, M); //start compression
		cmprBytes = SZ_compress(SZ_DOUBLE, g, &cmprSize, 0, 0, 0, nbLines, M);
		ZC_CompareData* compareResult = ZC_endCmpr(basicDataProperty, cmprCaseName, cmprSize); //end compression

		ZC_startDec(); //start decompression
		decData = SZ_decompress(SZ_DOUBLE, cmprBytes, cmprSize, 0, 0, 0, nbLines, M);
		ZC_endDec(compareResult, decData); //end decompression

		freeDataProperty(basicDataProperty); //free the basic data property generated at current time step
		//Basic data property includes only basic properties such as min, max, value_range of the data, which are necessary for assessing compression quality.
		//generate the full data property analysis results, which are optional to users. It includes more
```

```
information such as entropy and autocorrelation.
		ZC_DataProperty* fullDataProperty = ZC_genProperties(propName, ZC_DOUBLE, g, 0, 0, 0,
nbLines, M);
		if(rank==0)
			ZC_writeDataProperty(fullDataProperty, "dataProperties");

		freeDataProperty(fullDataProperty); //free data property generated at current time step
		free(cmprBytes);
		free(decData);
	}

	if (((i%ITER_OUT) == 0) && (rank == 0)) {
		printf("Step : %d, error = %f\n", i, globalerror);
	}
	if ((i%REDUCE) == 0) {
		MPI_Allreduce(&localerror, &globalerror, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
	}
	if(globalerror < PRECISION) {
		break;
	}
}
```

Note: The compression assessment (ZC_startCmpr(), ZC_endCmpr(), ZC_startDec(), and ZC_endDec()) and data property analysis (ZC_genProperties()) are separate calls.

5. Testing and Usage (MPI version)

In order to activate the MPI version of Z-checker, you need to install an mpi library such as MPICH and adopt "--enable-mpi" option during the installation of Z-checker. We provide three examples to do the parallel data analysis and parallel compression assessment respectively. You need to switch executionMode from OFFLINE to ONLINE in zc.config; or set the global variable "executionMode" to be ZC_ONLINE (its value is 1) during initialization (SZ_Init()).

(*Remember to install z-checker with "--enable-mpi" if you use ".configure;make;make install" and set executionMode = ONLINE*. *For installation, you can also use cmake to install it alternatively. Detailed can be found in Section 5.2*)

5.1 Examples of Online MPI Version

- **examples/analyzeDataProperty_online.c**
  The key data analysis is performed by each rank in parallel, inside the following line:
  property = ZC_genProperties(varName, ZC_FLOAT, data, r5, r4, r3, r2, r1);
  The global data analysis results will be reduced into the global variable *property* only at the root rank (rankID == 0).
  (Tips: ZC_genProperties() performs the offline or online execution based on the setting of executionMode).
- **examples/compareDataSets_online.c**
  The compression assessment is performed in parallel, when executing following line:
  compareResult =ZC_compareData(varName,ZC_FLOAT,data1,data2, r5, r4, r3, r2, r1);
  The global compression assessment will be reduced into the global variable *compareResult* only at the root rank (rankID == 0).
- **examples/heatdis.c**
  The heatdis.c is a heat distribution simulation using Jocobi method. In this example code, there are two key state variables (called g and h). In the example, we demonstrate how to compress, decompress, and perform online assessment every 50 time steps during the simulation, by calling ZC_startCmpr(), ZC_endCmpr(), ZC_startDec() and ZC_endDec(). The key codes are shown below.

```
for (i = 0; I < ITER_TIMES; i++) {
        localerror = doWork(nbProcs, rank, M, nbLines, g, h); //perform the simulation
        if(i%50==0) //control the compression frequency over time steps
        {
                sprintf(propName, "%s_%04d", varName, i); //make a name for the current target data property
                ZC_DataProperty* dataProperty = ZC_startCmpr(propName, ZC_DOUBLE, g, 0, 0, 0, nbLines,
        M); //start compression
                cmprBytes = SZ_compress(SZ_DOUBLE, g, &cmprSize, 0, 0, 0, nbLines, M);

                ZC_CompareData* compareResult = ZC_endCmpr(dataProperty, cmprSize); //end compression
                sprintf(cmprCaseName, "%s_%s_%04d(1E-3)", varName, compressorName, i);
                ZC_startDec(); //start decompression
                decData = SZ_decompress(SZ_DOUBLE, cmprBytes, cmprSize, 0, 0, 0, nbLines, M);
                ZC_endDec(compareResult, cmprCaseName, decData); //end decompression

                if(rank==0)
                        ZC_writeDataProperty(dataProperty, "dataProperties");

                freeDataProperty(dataProperty); //free data property generated at current time step
                freeCompareResult(compareResult); //free compression assessment results at current time step
                free(cmprBytes);
                free(decData);
```

```
	}
	if (((i%ITER_OUT) == 0) && (rank == 0)) {
		printf("Step : %d, error = %f\n", i, globalerror);
	}
	if ((i%REDUCE) == 0) {
		MPI_Allreduce(&localerror, &globalerror, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
	}
	if(globalerror < PRECISION) {
		break;
	}
}
```

Note: heatdis.c needs to be compiled with SZ library and ZC library. Please see Makefile.bk in examples/ for details.

5.2 Online Visualization

Please follow the following steps to install and test it.
Basically, there are two steps to execute, respectively.
First of all, you need to install boost, mpich and SZ (http://github.com/disheng222/SZ) on your machine because they are not included in Z-checker. After that, you can download z-checker package from github (http://github.com/CODARcode/Z-checker). All online codes (including web server and daemon codes) are included in the Z-checker package.

Step 1: compiling z-checker and heatdis.c run normally on your laptop.

Compile z-checker using cmake.
1. Go to the Z-checker directory, then “mkdir build”
2. cd build; and then run the following command:
CC=mpicc CXX=mpicxx cmake .. -DSZ_INCLUDES=/home/sdi/Install/sz-2.0-install/include -DSZ_LIBRARIES="/home/sdi/Install/sz-2.0-install/lib/libSZ.so;/home/sdi/Install/sz-2.0-install/lib/libzstd.so"
(You need to specify the SZ's include and lib path in the above command
3. make
Then, you can go to the Z-checker/build/bin and test heatdis.

```
mpirun -np 8 ./heatdis 100 SZ/example/sz.config Z-checker/examples/zc.config temp SZ
```

(In the above example, "SZ/example/sz.config" is the path of sz.config and "Z-checker/examples/zc.config" is the path of zc.config

Then, you should be able to see that heatdis is running with 8 cores; and please keep it running.

Step 2. Launch the web server.
1. Go to the directory Z-checker/public and run the following command:
python -m SimpleHTTPServer

2. Then, launch a browser and type "localhost:8000". Then, a window will be popped up for you to specify the port number. In there, you need to input *localhost* in the first field and *9091* in the second field, which is the "port number" to receive compression results data from the program heatdis.

6. Application Programming Interface (API)

Programming interfaces are provided in C, and we provide a wrapper to call R script in the C code.

6.1 Initialization and finalization of the Z-checker environment

6.1.1 ZC_Init

int ZC_Init(char *configFilePath);

Description: ZC_Init is used to load the parameters set in the configuration file and allocate memory.

6.1.2 ZC_Finalize

void ZC_Finalize();

Description: ZC_Finalize is used to free the memory (data structure such as hash_table and compression result elements) allocated during the assessment.

6.2 Generic I/O

6.2.1 ZC_readDoubleData in bytes

double *ZC_readDoubleData(char *srcFilePath, size_t *nbEle);

Description: ZC_readDoubleData is used to read the double-precision binary data file.

Arguments: char *srcFilePath – the file path of the binary data file

Int *nbEle – the number of data points in the data set.

Return: the pointer that points to the data set read from the file

6.2.2 ZC_readFloatData in bytes

float *ZC_readFloatData(char *srcFilePath, size_t *nbEle);

Description: ZC_readFloatData is used to read the single-precision binary data file.

Arguments: char *srcFilePath – the file path of the binary data file

size_t *nbEle – the number of data points in the data set.

Return: the pointer that points to the data set read from the file

6.2.3 ZC_writeFloatData_inBytes

void ZC_writeFloatData_inBytes(float *data, size_t nbEle, char* tgtFilePath);

Description: write single-precision floating-point data into a file in binary format.

Arguments: float *data – the data set

size_t nbEle – the number of data points

char* tgtFilePath – the file path of the data file

6.2.4 ZC_writeDoubleData_inBytes

void ZC_writeDoubleData_inBytes(double *data, size_t nbEle, char* tgtFilePath);

Description: write double-precision floating-point data into a file in binary format.

Arguments: double *data – the data set

size_t nbEle – the number of data points

char* tgtFilePath – the file path of the target data file

6.2.5 ZC_writeData in text

void ZC_writeData(void *data, int dataType, size_t nbEle, char *tgtFilePath);

Description: write data in the textual format

Arguments: void *data – the data set

Int dataType – the data type (either ZC_FLOAT or ZC_DOUBLE)

size_t nbEle – the number of data points

char* tgtFilePath – the file path of the target data file

6.3 Data property analysis

6.3.1 ZC_genProperties

ZC_DataProperty* ZC_genProperties(char* varName, int dataType, void *oriData, size_t r5, size_t r4, size_t r3, size_t r2, size_t r1);

Description: perform property analysis and generate the analysis results..

Arguments: char* varName – the name of the variable

int dataType – the data type (either ZC_FLOAT or ZC_DOUBLE)

void* oriData – the data set analyze

size_t r5,r4,r3,r2,r1 – the sizes along each dimension (r1 changes fastest)

Return: the pointer pointing to the data structure ZC_DataProperty.

ZC_DataProperty:

```
typedef struct ZC_DataProperty
{
        char* varName;
        int dataType; /*ZC_DOUBLE or ZC_FLOAT*/
        size_t r5;
        size_t r4;
        size_t r3;
        size_t r2;
        size_t r1;

        void *data;

        long numOfElem;
```

```
        double minValue;
        double maxValue;
        double valueRange;
        double avgValue;
        double entropy;
        double zeromean_variance;
        double* autocorr; /*array of autocorrelation coefficients*/
        complex* fftCoeff; /*array of fft coefficients*/
        double* lap;
} ZC_DataProperty;
```

6.3.2 ZC_printDataProperty

void ZC_printDataProperty(ZC_DataProperty* property);

Description: Print the key property analysis results to the screen.

Arguments: ZC_DataProperty* property – the property results to print.

6.3.3 ZC_writeDataProperty

void ZC_writeDataProperty(ZC_DataProperty* property, char* tgtWorkspaceDir);

Description: Write the property analysis results to the files.

Arguments: ZC_DataProperty* property – the property results to dump

char* tgtWorkspaceDir – the target workspace directory to contain the results.

6.3.4 ZC_loadDataProperty

ZC_DataProperty* ZC_loadDataProperty(char* propResultFile);

Description: Load the data property from a property result file.

Arguments: char* propResultFile – the property analysis result file (*.prop).

Return: ZC_DataProperty* – the property analysis result

6.3.5 ZC_computeDataLength_online

long ZC_computeDataLength_online(size_t r5, size_t r4, size_t r3, size_t r2, size_t r1)

Arguments: dimension sizes

Return: the total size of the data set including all ranks.

6.4 Data Comparison

6.4.1 ZC_compareData

ZC_CompareData* ZC_compareData(char* varName, int dataType, void *oriData, void *decData, size_t r5, size_t r4, size_t r3, size_t r2, size_t r1);

Description: Compare the original data and decompressed data and generate the comparison results.

Arguments: char* varName – variable name

Int dataType – the type of data (ZC_FLOAT or ZC_DOUBLE)

Void *oriData – the original data set

Void *decData – the decompressed data set

Size_t r5,r4,r3,r2,r1 – the sizes along each dimension (r1 changes fastest)

Return: ZC_CompareData* – the copmression result (the comparison between original data and decompressed data)

ZC_CompareData:

```
typedef struct ZC_CompareData
{
        ZC_DataProperty* property;

        double compressTime;
        double compressRate;
        int compressSize;
        double compressRatio; /*compression factor = orig_size/compressed_size*/
        double rate; /*# bits to be represented for each data point*/
```

```
        double decompressTime;
        double decompressRate;

        double minAbsErr;
        double avgAbsErr;
        double maxAbsErr;
        double* autoCorrAbsErr;
        double* absErrPDF; /*keep the distribution of errors (1000 elements)*/
        double* pwrErrPDF;
        double err_interval;
        double err_interval_rel;
        double err_minValue;
        double err_minValue_rel;

        double minRelErr;
        double avgRelErr;
        double maxRelErr;

        double minPWRErr;
        double avgPWRErr;
        double maxPWRErr;
        double snr;
        double rmse;
        double nrmse;
        double psnr;
        double valErrCorr;
        double pearsonCorr;

        complex *fftCoeff;
} ZC_CompareData;
```

6.4.2 ZC_printCompressionResult

void ZC_printCompressionResult(ZC_CompareData* compareResult);

Description: Print the compression results onto the screen.

Arguments: ZC_CompareData* compareResult – comparison results

6.4.3 ZC_writeCompressionResult

void ZC_writeCompressionResult(ZC_CompareData* compareResult, char* solution, char* varName, char* tgtWorkspaceDir);

Description: Write the comparison results to files.

Arguments: ZC_CompareData* compareResult – comparison results

Char* solution – the compression case, such as SZ_f(1E-3) or SZ_d(1E-3)

Char* varName – the name of the variable

Char* tgtWorkspaceDir – the target workspace directory to contain the results.

6.4.4 ZC_loadCompressionResult

ZC_CompareData* ZC_loadCompressionResult(char* cmpResultFile);

Description: Load the compression results into the memory (in order to for example generate the gnuplot figures)

Arguments: char* cmpResultFile – the compression result file, such as .cmp file in the 'compressionResults' directory.

6.5 Setting monitoring calls in compressor codes

The four critical functions are ZC_startCmpr(), ZC_endCmpr(), ZC_startDec(), and ZC_endDec(). They will perform offline or online assessment, depending on the executionMode (either ZC_OFFLINE or ZC_ONLINE) set in zc.config. Examples are shown in heatdis.c, compareDataSets.c.

6.5.1 ZC_startCmpr

ZC_DataProperty* ZC_startCmpr(char* varName, int dataType, void *oriData, size_t r5, size_t r4, size_t r3, size_t r2, size_t r1);

Description: ZC_startCmpr() function indicates a starting point of the compression. (You need to put this function right before calling a compression operation)

6.5.2 ZC_startCmpr_withDataAnalysis

ZC_DataProperty* ZC_startCmpr_withDataAnalysis(char* varName, int dataType, void *oriData, size_t r5, size_t r4, size_t r3, size_t r2, size_t r1);

Description: ZC_startCmpr_withDataAnalysis( ) integrates the data analysis. It supports **ONLY offline mode** in this version.

Note:
This function includes the data analysis. That is, it will analyze the data and output the analysis results into the dataProperty/ directory. The analysis result includes auto-correlation of the data, spectrum result (based on FFT), distribution of the data, entropy, etc.

data analysis is costly, so we suggest to call ZC_startCmpr instead of ZC_startCmpr_withDataAnalysis in most cases.

6.5.3 ZC_endCmpr

ZC_CompareData* ZC_endCmpr(ZC_DataProperty* dataProperty, size_t cmprSize);

Description: This function indicates an ending point of the compression operation.

Arguments: ZC_DataProperty* dataProperty – the data property generated/returned by ZC_startCmpr.

size_t cmprSize – the compressed size

Return: ZC_CompareData* - the compression results.

6.5.4 ZC_startDec

void ZC_startDec();

Description: This function indicates the starting point of the decompression operations.

6.5.5 ZC_endDec

void ZC_endDec(ZC_CompareData* compareResult, char* solution, void *decData);

Description: This function specifies the ending point of the decompression.

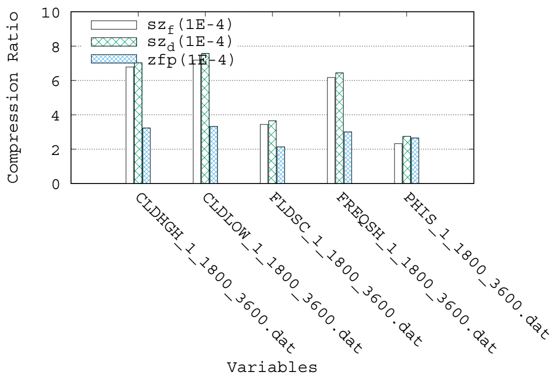
6.6 Plotting the analysis data suitable for Gnuplot

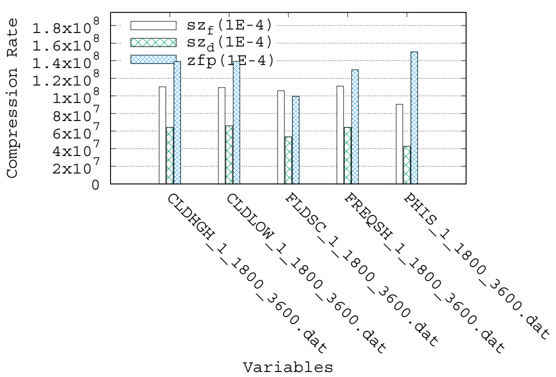
6.6.1 ZC_ plotHistogramResults

void ZC_plotHistogramResults(int cmpCount, char** compressorCases);

Description: Generate .eps files for histogram-style results, including compression ratios, compression rate, decompression rate, and PSNR.

Example output (in the directory 'compareCompressors') based on compression ratio and compression rate:





Arguments: int cmpCount – the number of compressor cases

Char** compressorCases – the compressor cases

6.6.2 ZC_plotComparisonCases

void ZC_plotComparisonCases();

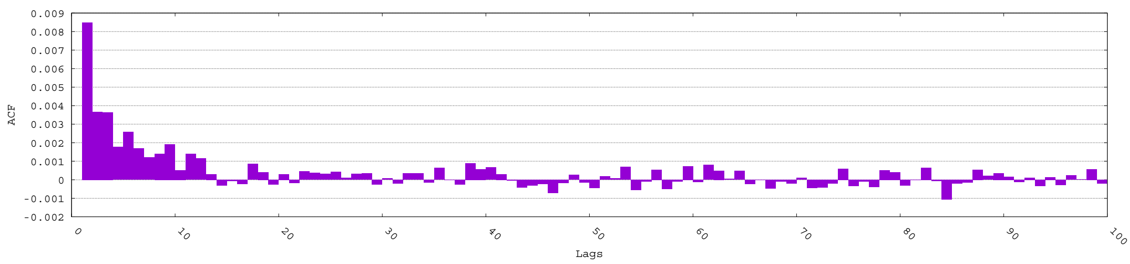
Description: Plot comparison cases. This function calls ZC_plotHistogramResults to complete the plotting work.

6.6.3 ZC_plotAutoCorr_CompressError

void ZC_plotAutoCorr_CompressError();

Description: Plot auto-correlation coefficients based on compression errors.

Example output:

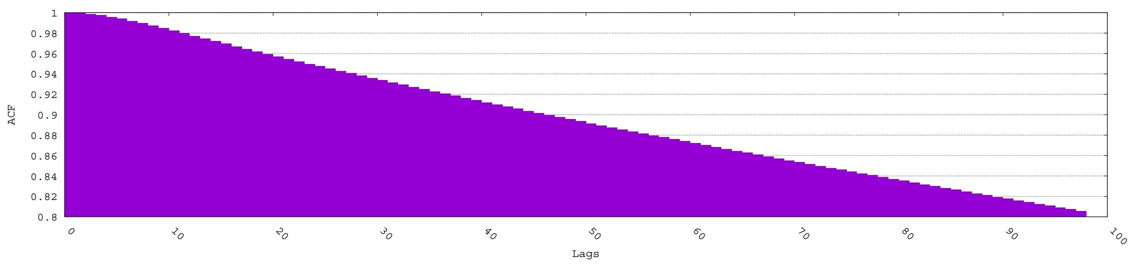


6.6.4 ZC_plotAutoCorr_DataProperty

void ZC_plotAutoCorr_DataProperty();

Description: Plot auto-correlation coefficients based on data set.

Example output:

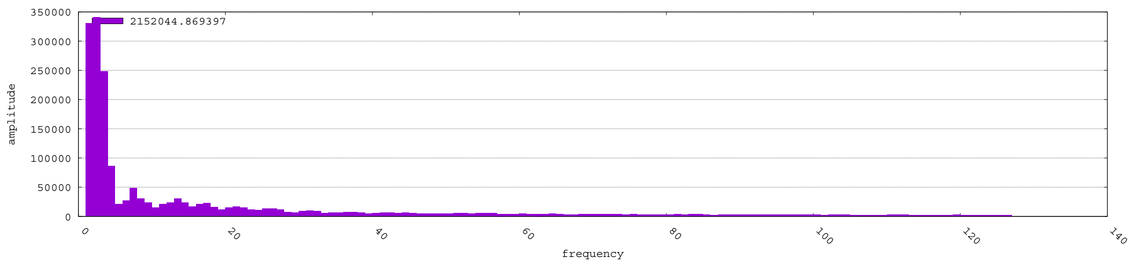


6.6.5 ZC_plotFFTAmplitude_OriginalData

void ZC_plotFFTAmplitude_OriginalData();

Description: Plot FFT amplitude for original data.

Example output:

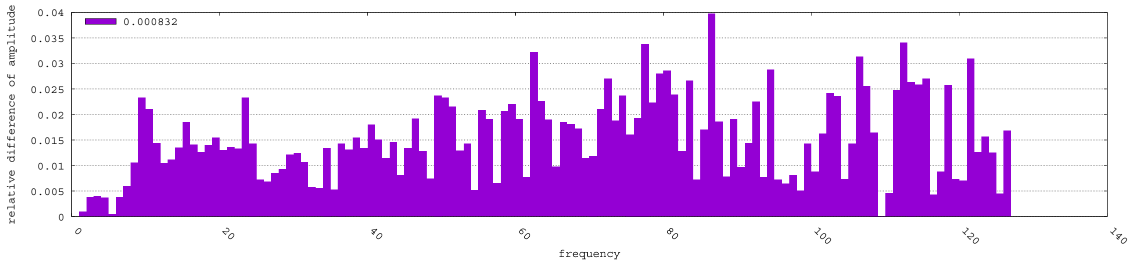


6.6.6 ZC_plotFFTAmplitude_DecompressData

void ZC_plotFFTAmplitude_DecompressData();

Description: Plot FFT amplitude difference between the original data and decompressed data.

Example output:

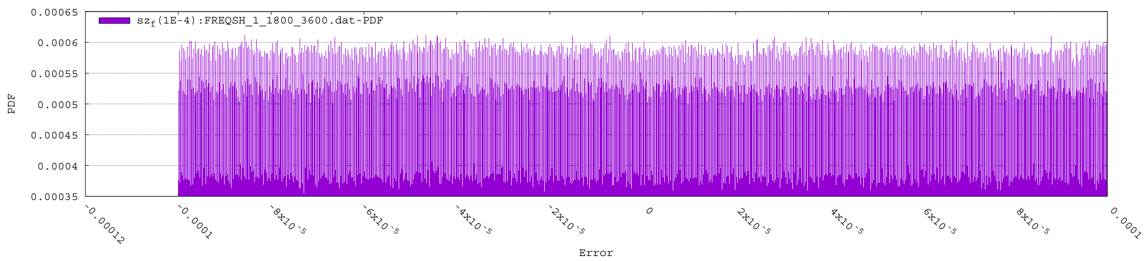


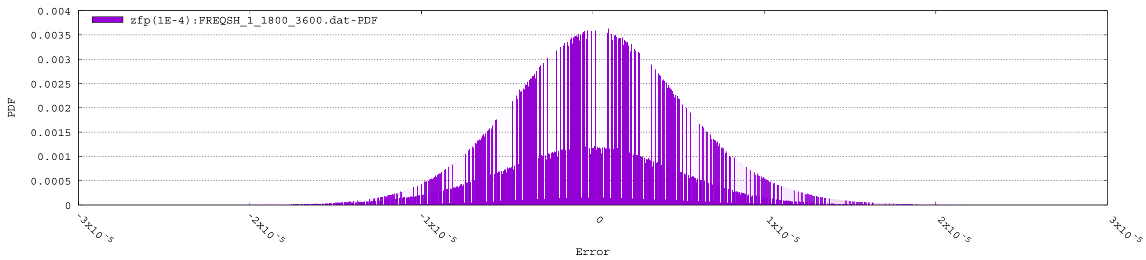
6.6.7 ZC_ plotErrDistribtion

void ZC_plotErrDistribtion();

Description: Plot the distribution of compression errors.

Example output (the first one is SZ and the second is based on ZFP)





6.7 Generating Gnuplot scripts

6.7.1 genGnuplotScript_linespoints

char** genGnuplotScript_linespoints(char* dataFileName, char* extension, int fontSize, int columns, char* xlabel, char* ylabel);

Description: Generate Gnuplot script based on lines-points format.

6.7.2 genGnuplotScript_histogram

char** genGnuplotScript_histogram(char* dataFileName, char* extension, int fontSize, int columns, char* xlabel, char* ylabel, long maxYValue);

Description: Generate Gnuplot script for histogram data.

6.7.3 genGnuplotScript_lines

char** genGnuplotScript_lines(char* dataFileName, char* extension, int fontSize, int columns, char* xlabel, char* ylabel);

Description: Generate Gnuplot script based on line-type format.

6.7.4 genGnuplotScript_fillsteps

char** genGnuplotScript_fillsteps(char* dataFileName, char* extension, int fontSize, int columns, char* xlabel, char* ylabel);

Description: Generate Gnuplot script based on fill-steps format.

6.8 Generating analysis report in PDF file format

Please see examples/generateReport.c for details.

6.9 Calling R script from Z-checker

The wrapper codes for calling R scripts from Z-checker can be found in [DOWNLOAD_PACKAGE]/R. You need to modify Makefile by setting the R installation path, before the compilation with “make”.
Specifically, the R installation path is set as follows:
R_BASE = [Your installation path of R] (e.g., /usr/lib64/R); hint: you could run the command "which Rscript" to check where it is.
For example, in my machine, R_Base=/usr/lib64/R.

After the compilation of R, you can use the executable called “zccallr” in the directory ./test/ to run any R script. It calls different functions such as ZC_callR_1_1d, SZ_callR_1_2d, to execute the function in a specific R script.

Usage: zccallr <options>

Options:

* Rscript file:

    -s <script_file>: specify the path of the R_script_file

    -c <function_name>: specify the function

* data type:

    -i : integer data (int type)

    -f : single precision (float type)

    -d : double precision (double type)

* input data files:

    -e <endian_type>: endian type of the binary data in input files: 0(little-endian); 1(big-endian)

    -A <first data file> : first data file such as original data file

    -B <second data file> : second data file such as decompressed data file

    -C <third data file> : third data file for analysis

    -D <fourth data file> : fourth data file for analysis

    -E <fifth data file> : fifth data file for analysis

    -F <sixth data file> : sixth data file for analysis

* output type of result file:

    -r : print the result on the screen.

    -b : analysis result stored in binary format

    -t : analysis result stored in text format

    -o <output file path> : the path of the output file.

* 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]

* examples:

    zccallr -s func.R -c add1 -r -f -e 0 -A ../../examples/testdata/x86/testfloat_8_8_128.dat -3 8 8 128

    zccallr -s func.R -c computeErr -r -f -e 0 \
-A ../../examples/testdata/x86/testfloat_8_8_128.dat \
-B ../../examples/testdata/x86/testfloat_8_8_128.dat.sz.out -3 8 8 128

Before calling ZC_callR_x_xd, you need to initialize the environment as follows:

```
//Initialize an embedded R session
int r_argc = 2;
char *r_argv[] = { "R", "--silent" };
Rf_initEmbeddedR(r_argc, r_argv);

// Invoke a function in R
source(rscriptPath);    //rscriptPath is the file path of a R script
```

We describe the API as follows.

6.9.1 ZC_callR_1_1d

int ZC_callR_1_1d(char* rRuncName, int vecType, size_t inLen, void* in, size_t * outLen, double** out);

Description: This function calls an R function with one 1D data set (i.e., a vector) as input.

Arguments: char* rRuncName – the function name in the R script.

Int vecType – the data type of the vector

size_t inLen – the number of elements in the vector

void* in – the pointer that points to the data set

size_t* outLen – the number of elements in the output data set

double** out – the pointer pointing to the output data set. (new memory will be allocated in the function)

Return: status (either ZC_R_SUCCESS or ZC_R_ERROR)

6.9.2 ZC_callR_2_1d

int ZC_callR_2_1d(char* rFuncName, int vecType, size_t in1Len, void* in1, size_t in2Len, void* in2, size_t * outLen, double** out);

Description: This function calls an R function with two 1D data sets (i.e., two vectors).

Arguments: char* rRuncName – the function name in the R script.

Int vecType – the data type of the vector

size_t in1Len – the number of elements in the first vector

void* in1 – the pointer that points to the first vector.

size_t in2Len – the number of elements in the second vector

void* in2 – the pointer that points to the second vector.

size_t* outLen – the number of elements in the output data set double** out – the pointer pointing to the output data set. (new memory will be allocated in the function)

Return: status (either ZC_R_SUCCESS or ZC_R_ERROR)

6.9.3 ZC_callR_1_2d

int ZC_callR_1_2d(char* rFuncName, int vecType, size_t in_n2, size_t in_n1, void* in, size_t * outLen, double** out);

Description: This function calls an R function with one 2D data set (i.e., a matrix) as input.

Arguments: char* rRuncName – the function name in the R script.

Int vecType – the data type of the vector

size_t in_n2 – the size of the higher dimension for the matrix

size_t in_n1 – the size of the lower dimension for the matrix

void* in – the pointer that points to the matrix data

size_t* outLen – the number of elements in the output data set

double** out – the pointer pointing to the output data set. (new memory will be allocated in the function)

Return: status (either ZC_R_SUCCESS or ZC_R_ERROR)

6.9.4 ZC_callR_2_2d

int ZC_callR_2_2d(char* rFuncName, int vecType, size_t in1_n2, size_t in1_n1, void* in1, size_t in2_n2, size_t in2_n1, void* in2, size_t* outLen, double** out);

Description: This function calls an R function with two 2D data sets (i.e., a matrices).

Arguments: char* rRuncName – the function name in the R script.

Int vecType – the data type of the vector

size_t in1_n2 – the size of the higher dimension for the first matrix

size_t in1_n1 – the size of the lower dimension for the first matrix void* in1 – the pointer that points to the first matrix data.

size_t in2_n2 – the size of the higher dimension for the second matrix

size_t in2_n1 – the size of the lower dimension for the second matrix

void* in2 – the pointer that points to the second matrix data.

size_t* outLen – the number of elements in the output data set

double** out – the pointer pointing to the output data set. (new memory will be allocated in the function)

Return: status (either ZC_R_SUCCESS or ZC_R_ERROR)

6.9.5 ZC_callR_1_3d

int ZC_callR_1_3d(char* rFuncName, int vecType, size_t in_n3, size_t in_n2, size_t in_n1, void* in, size_t * outLen, double** out);

Description: This function calls an R function with one 3D data set (i.e., a matrix) as input.

Arguments: char* rRuncName – the function name in the R script.

Int vecType – the data type of the vector

size_t in_n3 – the size of the highest dimension for the matrix

size_t in_n2 – the size of the middle dimension for the matrix

size_t in_n1 – the size of the lowest dimension for the matrix

void* in – the pointer that points to the matrix data

size_t* outLen – the number of elements in the output data set

double** out – the pointer pointing to the output data set. (new memory will be allocated in the function)

Return: status (either ZC_R_SUCCESS or ZC_R_ERROR)

6.9.6 ZC_callR_2_3d

int ZC_callR_2_3d(char* rFuncName, int vecType, size_t in1_n3, size_t in1_n2, size_t in1_n1, void* in1, size_t in2_n3, size_t in2_n2, size_t in2_n1, void* in2, size_t * outLen, double** out);

Description: This function calls an R function with two 3D data sets (i.e., two matrices).

Arguments: char* rRuncName – the function name in the R script.

Int vecType – the data type of the two matrices

size_t in1_n3 – the size of the highest dimension for the first matrix

size_t in1_n2 – the size of the middle dimension for the first matrix

size_t in1_n1 – the size of the lowest dimension for the first matrix

void* in1 – the pointer that points to the first matrix data

size_t in2_n3 – the size of the highest dimension for the second matrix

size_t in2_n2 – the size of the middle dimension for the second matrix

size_t in2_n1 – the size of the lowest dimension for the second matrix

void* in2 – the pointer that points to the second matrix data

size_t* outLen – the number of elements in the output data set

double** out – the pointer pointing to the output data set. (new memory will be allocated in the function)

Return: status (either ZC_R_SUCCESS or ZC_R_ERROR)

7 Configuration file

You can switch on/off the metrics to check in the configuration file (zc.config) based on your demand. Please see the comments in the sz.config file for details.
In particular, there are two modes for running z-checker, “probe” mode and “analysis” mode. The former indicates the online checking by running the compressor, and the latter is to collect the compression results based on the previous probe-mode runs.

The configuration file actually is not a must when running a compressor with the z-checker as a probe/monitor. If configuration file is not used in the probe-mode running, all metrics will be switched on by default.

Note that if the z-checker is running in the “analysis” mode, please do remember to switch the checkingStatus parameter to “analysis” from “probe”.

8. Version history

The latest version (**version 0.1.x**) is the recommended one.

Version	New features
ZC 0.1.0	Prototype of Z-checker. It’s able to perform the data analysis for the original data set and compression quality analysis based on specific data compressors.
ZC 0.1.1	In the z-checker-installer, the users are able to set different error bounds for different compressors. It also supports using a configuration file (such as varInfo.txt) to specify the data files with the data having different dimension sizes. – 12/2017
ZC 0.1.2	We added a more easy-to-use executables (createOfflineCase.sh and runOfflineCase). It also supports to put the offline assessment results into z-checker-report, by comparing a customized compressor to SZ and ZFP. Some online functions (MPI implementation) have been added to the z-checker. – 01/2018
ZC 0.1.3	We implemented the online-version of Z-checker, allowing to run MPI applications with Z-checker dynamically to generate compression results. In this version, we also support convenient integration of a new compressor into z-checker-installer by using the manageCompressor command. See Section 4.3.2 for details.
ZC 0.2.0	Z-checker 0.2.0 is a stable version that passes a series of tests based on different configurations and datasets. The executables (including examples and executable command) in the Z-checker package can be used to do the regression testing. The best testing methods are still using z-checker-installer. We also provided the support of Travis CI for the test of continuous integration. Specifically, it creates an individual working space for each simulation dataset such that its compression assessment results would not be affected by the package update or the assessments of other datasets. Some new features in 0.2 compared with 0.1 are listed as follows (more details could be referenced according to the commits history) • Web visualization module f SZ has been updated. • fix a bug in computing the entropy for serial version. • fix bugs in CMakelists.txt

- revise user guide based on online functions.
- add heat distribution MPI code example to demonstrate how to use online functions.
- add 2D SSIM calculation.
- add/test laplace derivative metric.
- remove costly default functions in z-checker-installer, such as autocorrelation.

9. Q&A and Trouble shooting

TBD

Please contact sdi1@anl.gov if you encounter any problems.

<END>