zfp Documentation

Release 0.5.2

Peter Lindstrom

Oct 02, 2018

Contents

1 Introduction **1**

2 Overview **3**

3 License **5**

4 Installation **7**
- 4.1 GNU Builds . . . 7
- 4.2 CMake Builds . . . 8
- 4.3 Compile-Time Macros . . . 8

5 Algorithm **11**

6 Compression Modes **13**
- 6.1 Expert Mode . . . 13
- 6.2 Fixed-Rate Mode . . . 14
- 6.3 Fixed Precision . . . 15
- 6.4 Fixed Accuracy . . . 15

7 High-Level C API **17**
- 7.1 Macros . . . 17
- 7.2 Types . . . 18
- 7.3 Constants . . . 19
- 7.4 Functions . . . 20

8 Low-Level C API **23**
- 8.1 Stream Manipulation . . . 23
- 8.2 Encoder . . . 24
- 8.3 Decoder . . . 25
- 8.4 Utility Functions . . . 27

9 Bit Stream API **29**
- 9.1 Strided Streams . . . 29
- 9.2 Macros . . . 30
- 9.3 Types . . . 30
- 9.4 Constants . . . 30
- 9.5 Functions . . . 30

10 Compressed Arrays 33
10.1 Array Classes . . . 33
10.2 Caching . . . 36
10.3 References . . . 36
10.4 Pointers . . . 37
10.5 Iterators . . . 38

11 Tutorial 41
11.1 High-Level C Interface . . . 41
11.2 Low-Level C Interface . . . 44
11.3 Compressed C++ Arrays . . . 45

12 File Compressor 51
12.1 Usage . . . 52

13 Code Examples 55
13.1 Simple Compressor . . . 55
13.2 Diffusion Solver . . . 55
13.3 Speed Benchmark . . . 56
13.4 PGM Image Compression . . . 56
13.5 In-place Compression . . . 56
13.6 Iterators . . . 57

14 Regression Tests 59

15 FAQ 61

16 Troubleshooting 73

17 Limitations 79

18 Future Directions 81

19 Contributors 83

20 Release Notes 85

CHAPTER 1

Introduction

zfp is an open source C/C++ library for compressed numerical arrays that support high throughput read and write random access. zfp also supports streaming compression of integer and floating-point data, e.g., for applications that read and write large data sets to and from disk.

zfp was developed at Lawrence Livermore National Laboratory and is loosely based on the *algorithm* described in the following paper:

> Peter Lindstrom
> "Fixed-Rate Compressed Floating-Point Arrays"
> IEEE Transactions on Visualization and Computer Graphics
> 20(12):2674-2683, December 2014
> doi:10.1109/TVCG.2014.2346458

zfp was originally designed for floating-point arrays only, but has been extended to also support integer data, and could for instance be used to compress images and quantized volumetric data. To achieve high compression ratios, zfp uses lossy but optionally error-bounded compression. Although bit-for-bit lossless compression of floating-point data is not always possible, zfp is usually accurate to within machine epsilon in near-lossless mode.

zfp works best for 2D and 3D arrays that exhibit spatial correlation, such as continuous fields from physics simulations, images, regularly sampled terrain surfaces, etc. Although zfp also provides a 1D array class that can be used for 1D signals such as audio, or even unstructured floating-point streams, the compression scheme has not been well optimized for this use case, and rate and quality may not be competitive with floating-point compressors designed specifically for 1D streams.

zfp is freely available as open source under a *BSD license*. For more information on zfp and comparisons with other compressors, please see the zfp website. For questions, comments, requests, and bug reports, please contact Peter Lindstrom.

CHAPTER 2

Overview

zfp is a compressor for integer and floating-point data stored in multidimensional arrays. The compressor is primarily *lossy*, meaning that the numerical values are usually only approximately represented, though the user may specify error tolerances to limit the amount of loss. *Lossless compression*, where values are represented exactly, is possible in some circumstances.

The zfp software consists of three main components: a C library for compressing whole arrays (or smaller pieces of arrays); C++ classes that implement compressed arrays; and a command-line compression tool and other code examples. zfp has also been incorporated into several independently developed plugins for interfacing zfp with popular I/O libraries and visualization tools such as ADIOS, HDF5, and VTK.

The typical user will interact with zfp via one or more of those components, specifically

- Via the *C API* when doing I/O in an application or otherwise performing data (de)compression online.
- Via zfp's C++ in-memory *compressed array classes* when performing computations on very large arrays that demand random access to array elements, e.g. in visualization, data analysis, or even in numerical simulation.
- Via the zfp *command-line tool* when compressing binary files offline.
- Via one of the I/O libraries or visualization tools that support zfp, e.g.
  - ADIOS plugin
  - HDF5 plugin
  - VTK plugin

In all cases, it is important to know how to use zfp's *compression modes* as well as what the *limitations* of zfp are. Although it is not critical to understand the *compression algorithm* itself, having some familiarity with its major components may help understand what to expect and how zfp's parameters influence the result.

zfp compresses d-dimensional (1D, 2D, and 3D) arrays of integer or floating-point values by partitioning the array into blocks of 4^d values, i.e., 4, 16, or 64 values for 1D, 2D, and 3D arrays, respectively. Each such block is (de)compressed independently into a fixed- or variable-length bit string, and these bit strings are concatenated into a single stream of bits.

zfp usually truncates each bit string to a fixed number of bits to meet a storage budget or to some variable length needed to meet a given error tolerance, as dictated by the compressibility of the data. The bit string representing any given block may be truncated at any point and still yield a valid approximation. The early bits are most important; later bits progressively refine the approximation, similar to how the last few bits in a floating-point number have less significance than the first several bits and can often be discarded (zeroed) with limited impact on accuracy.

The next several sections cover information on the zfp algorithm and its parameters; the C API; the compressed array classes; examples of how to perform compression and work with the classes; how to use the binary file compressor; and code examples that further illustrate how to use zfp. The documentation concludes with frequently asked questions and troubleshooting, as well as current limitations and future development directions.

For questions not answered here, please contact Peter Lindstrom.

CHAPTER 3

License

Copyright (c) 2014-2017, Lawrence Livermore National Security, LLC.
Produced at the Lawrence Livermore National Laboratory.
Written by Peter Lindstrom.
LLNL-CODE-663824.
All rights reserved.

This file is part of the zfp library. For details, see http://computation.llnl.gov/casc/zfp/.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the disclaimer below.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the disclaimer (as noted below) in the documentation and/or other materials provided with the distribution.

3. Neither the name of the LLNS/LLNL nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL LAWRENCE LIVERMORE NATIONAL SECURITY, LLC, THE U.S. DEPARTMENT OF ENERGY OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Additional BSD Notice

1. This notice is required to be provided under our contract with the U.S. Department of Energy (DOE). This work was produced at Lawrence Livermore National Laboratory under Contract No. DE-AC52-07NA27344 with the DOE.

2. Neither the United States Government nor Lawrence Livermore National Security, LLC nor any of their employees, makes any warranty, express or implied, or assumes any liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately-owned rights.

3. Also, reference herein to any specific commercial products, process, or services by trade name, trademark, manufacturer or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

CHAPTER 4

Installation

zfp consists of three distinct parts: a compression library written in C, a set of C++ header files that implement compressed arrays, and a set of C and C++ examples. The main compression codec is written in C and should conform to both the ISO C89 and C99 standards. The C++ array classes are implemented entirely in header files and can be included as is, but since they call the compression library, applications must link with `libzfp`.

On Linux, macOS, and MinGW, zfp is easiest compiled using gcc and gmake. CMake support is also available, e.g. for Windows builds. See below for instructions on GNU and CMake builds.

zfp has successfully been built and tested using these compilers:

- gcc versions 4.4.7, 4.7.2, 4.8.2, 4.9.2, 5.4.1, 6.3.0
- icc versions 12.0.5, 12.1.5, 15.0.4, 16.0.1, 17.0.0, 18.0.0
- clang version 3.6.0
- xlc version 12.1
- MinGW version 5.3.0
- Visual Studio versions 14.0 (2015), 14.1 (2017)

NOTE: zfp requires 64-bit compiler and operating system support.

4.1 GNU Builds

To compile zfp using gcc, type:

```
make
```

from the zfp root directory. This builds `libzfp` as a static library as well as utilities and example programs. To optionally create a shared library, type:

```
make shared
```

and set `LD_LIBRARY_PATH` to point to `lib`. To test the compressor, type:

```
make test
```

If the compilation or regression tests fail, it is possible that some of the macros in the file `Config` have to be adjusted. Also, the tests may fail due to minute differences in the computed floating-point fields being compressed (as indicated by checksum errors). It is surprisingly difficult to portably generate a floating-point array that agrees bit-for-bit across platforms. If most tests succeed and the failures result in byte sizes and error values reasonably close to the expected values, then it is likely that the compressor is working correctly.

4.2 CMake Builds

To build zfp using CMake on Linux or macOS, start a Unix shell and type:

```
mkdir build
cd build
cmake ..
make
```

To also build the examples, replace the cmake line with:

```
cmake -DBUILD_EXAMPLES=ON ..
```

To build zfp using Visual Studio on Windows, start an MSBuild shell and type:

```
mkdir build
cd build
cmake ..
msbuild /p:Configuration=Release zfp.sln
msbuild /p:Configuration=Debug   zfp.sln
```

This builds zfp in both debug and release mode. See the instructions for Linux on how to change the cmake line to also build the example programs.

4.3 Compile-Time Macros

The behavior of zfp can be configured at compile time via a set of macros. For GNU builds, these macros are set in the file `Config`. For CMake builds, use the `-D` option on the cmake line as in the example above.

`ZFP_INT64`

`ZFP_INT64_SUFFIX`

`ZFP_UINT64`

`ZFP_UINT64_SUFFIX`

64-bit signed and unsigned integer types and their literal suffixes. Platforms on which `long int` is 32 bits wide may require `long long int` as type and `ll` as suffix. These macros are relevant **only** when compiling in C89 mode. When compiling in C99 mode, integer types are taken from `stdint.h`. Defaults: `long int`, `l`, `unsigned long int`, and `ul`, respectively.

`ZFP_WITH_ALIGNED_ALLOC`

Use aligned memory allocation in an attempt to align compressed blocks on hardware cache lines. Default: undefined/off.

ZFP_WITH_CACHE_TWOWAY

Use a two-way skew-associative rather than direct-mapped cache. This incurs some overhead that may be offset by better cache utilization. Default: undefined/off.

ZFP_WITH_CACHE_FAST_HASH

Use a simpler hash function for cache line lookup. This is faster but may lead to more collisions. Default: undefined/off.

ZFP_WITH_CACHE_PROFILE

Enable cache profiling to gather and print statistics on cache hit and miss rates. Default: undefined/off.

BIT_STREAM_WORD_TYPE

Unsigned integer type used for buffering bits. Wider types tend to give higher performance at the expense of lower bit rate granularity. For portability of compressed files between little and big endian platforms, *BIT_STREAM_WORD_TYPE* should be set to `uint8`. Default: `uint64`.

ZFP_BIT_STREAM_WORD_SIZE

CMake macro for indirectly setting *BIT_STREAM_WORD_TYPE*. Valid values are 8, 16, 32, 64. Default: 64.

BIT_STREAM_STRIDED

Enable support for strided bit streams that allow for non-contiguous memory layouts, e.g., to enable progressive access. Default: undefined/off.

CHAPTER 5

Algorithm

The zfp lossy compression scheme is based on the idea of breaking a d-dimensional array into independent blocks of 4^d values each, e.g. $4 \times 4 \times 4$ values in three dimensions. Each block is compressed/decompressed entirely independently from all other blocks. In this sense, zfp is similar to current hardware texture compression schemes for image coding implemented on graphics cards and mobile devices.

The compression scheme implemented in this version of zfp has evolved from the method described in the *original paper*, and can conceptually be thought of as consisting of eight sequential steps (in practice some steps are consolidated or exist only for illustrative purposes):

1. The d-dimensional array is partitioned into blocks of dimensions 4^d. If the array dimensions are not multiples of four, then blocks near the boundary are padded to the next multiple of four. This padding is invisible to the application.
2. The independent floating-point values in a block are converted to what is known as a block-floating-point representation, which uses a single, common floating-point exponent for all 4^d values. The effect of this conversion is to turn each floating-point value into a 31- or 63-bit signed integer. If the values in the block are all zero or are smaller in magnitude than the fixed-accuracy tolerance (see below), then only a single bit is stored with the block to indicate that it is "empty" and expands to all zeros. Note that the block-floating-point conversion and empty-block encoding are not performed if the input data is represented as integers rather than floating-point numbers.
3. The integers are decorrelated using a custom, high-speed, near orthogonal transform similar to the discrete cosine transform used in JPEG image coding. The transform exploits separability and is implemented efficiently in-place using the lifting scheme, requiring only 2.5 d integer additions and 1.5 d bit shifts by one per integer in d dimensions. If the data is "smooth," then this transform will turn most integers into small signed values clustered around zero.
4. The signed integer coefficients are reordered in a manner similar to JPEG zig-zag ordering so that statistically they appear in a roughly monotonically decreasing order. Coefficients corresponding to low frequencies tend to have larger magnitude, and are listed first. In 3D, coefficients corresponding to frequencies i, j, k in the three dimensions are ordered by $i + j + k$ first, and then by $i^2 + j^2 + k^2$.
5. The two's complement signed integers are converted to their negabinary (base negative two) representation using one addition and one bit-wise exclusive or per integer. Because negabinary has no dedicated single sign bit, these integers are subsequently treated as unsigned.

6. The bits that represent the list of 4^d integers are transposed so that instead of being ordered by coefficient they are ordered by bit plane, from most to least significant bit. Viewing each bit plane as an unsigned integer, with the lowest bit corresponding to the lowest frequency coefficient, the anticipation is that the first several of these transposed integers are small, because the coefficients are assumed to be ordered by magnitude.
7. The transform coefficients are compressed losslessly using embedded coding by exploiting the property that the coefficients tend to have many leading zeros that need not be encoded explicitly. Each bit plane is encoded in two parts, from lowest to highest bit. First the n lowest bits are emitted verbatim, where n depends on previous bit planes and is initially zero. Then a variable-length representation of the remaining $4^d - n$ bits, x, is encoded. For such an integer x, a single bit is emitted to indicate if $x = 0$, in which case we are done with the current bit plane. If not, then bits of x are emitted, starting from the lowest bit, until a one-bit is emitted. This triggers another test whether this is the highest set bit of x, and the result of this test is output as a single bit. If not, then the procedure repeats until all m of x's value bits have been output, where $2^{m-1} \leq x < 2^m$. This can be thought of as a run-length encoding of the zeros of x, where the run lengths are expressed in unary. The total number of value bits, n, in this bit plane is then incremented by m before being passed to the next bit plane, which is encoded by first emitting its n lowest bits. The assumption is that these bits correspond to n coefficients whose most significant bits have already been output, i.e. these n bits are essentially random and not compressible. Following this, the remaining $4^d - n$ bits of the bit plane are run-length encoded as described above, which potentially results in n being increased.
8. The embedded coder emits one bit at a time, with each successive bit potentially improving the quality of the reconstructed signal. The early bits are most important and have the greatest impact on signal quality, with the last few bits providing very small changes. The resulting compressed bit stream can be truncated at any point and still allow for a valid approximate reconstruction of the original signal. The final step truncates the bit stream in one of three ways: to a fixed number of bits (the fixed-rate mode); after some fixed number of bit planes have been encoded (the fixed-precision mode); or until a lowest bit plane number has been encoded, as expressed in relation to the common floating-point exponent within the block (the fixed-accuracy mode).

Various parameters are exposed for controlling the quality and compressed size of a block, and can be specified by the user at a very fine granularity. These parameters are discussed *here*.

CHAPTER 6

Compression Modes

zfp accepts one or more parameters for specifying how the data is to be compressed to meet various constraints on accuracy or size. At a high level, there are four different compression modes that are mutually exclusive: *expert*, *fixed-rate*, *fixed-precision*, and *fixed-accuracy* mode. The user has to select one of these modes and its corresponding parameters. In streaming I/O applications, the *fixed-accuracy mode* is preferred, as it provides the highest quality (in the absolute error sense) per bit of compressed storage.

The `zfp_stream` struct encapsulates the compression parameters and other information about the compressed stream. Its members should not be manipulated directly. Instead, use the access functions (see the *C API* section) for setting and querying them. The members that govern the compression parameters are described below.

6.1 Expert Mode

The most general mode is the 'expert mode,' which takes four integer parameters. Although most users will not directly select this mode, we discuss it first since the other modes can be expressed in terms of setting expert mode parameters.

The four parameters denote constraints that are applied to each block. Compression is terminated as soon as one of these constraints is not met, which has the effect of truncating the compressed bit stream that encodes the block. The four constraints are as follows:

uint **`zfp_stream.minbits`**

The minimum number of compressed bits used to represent a block. Usually this parameter is zero, unless each and every block is to be stored using a fixed number of bits to facilitate random access, in which case it should be set to the same value as `zfp_stream.maxbits`.

uint **`zfp_stream.maxbits`**

The maximum number of bits used to represent a block. This parameter sets a hard upper bound on compressed block size, and governs the rate in *fixed-rate mode*. It may also be used as an upper storage limit to guard against buffer overruns in combination with the accuracy constraints given by `zfp_stream.maxprec` and `zfp_stream.minexp`.

uint **`zfp_stream.maxprec`**

The maximum number of bit planes encoded. This parameter governs the number of most significant uncompressed bits encoded per transform coefficient. It does not directly correspond to the number of uncompressed mantissa bits for the floating-point or integer values being compressed, but is closely related. This is the parameter that specifies the precision in *fixed-precision mode*, and it provides a mechanism for controlling the *relative error*. Note that this parameter selects how many bits planes to encode regardless of the magnitude of the common floating-point exponent within the block.

int **`zfp_stream.minexp`**

The smallest absolute bit plane number encoded. The place value of each transform coefficient bit depends on the common floating-point exponent, *e*, that scales the integer coefficients. If the most significant coefficient bit has place value 2^e, then the number of bit planes encoded is (one plus) the difference between *e* and `zfp_stream.minexp`. As an analogy, consider representing currency in decimal. Setting `zfp_stream.minexp` to -2 would, if generalized to base-10, ensure that amounts are represented to cent accuracy, i.e. in units of 10^{-2} = $0.01. This parameter governs the *absolute error* in *fixed-accuracy mode*. Note that to achieve a certain accuracy in the decompressed values, the `zfp_stream.minexp` value has to be conservatively lowered since zfp's inverse transform may magnify the error (see also *FAQs #20-22*).

Care must be taken to allow all constraints to be met, as encoding terminates as soon as a single constraint is violated (except `zfp_stream.minbits`, which is satisfied at the end of encoding by padding zeros).

As mentioned above, other combinations of constraints can be used. For example, to ensure that the compressed stream is not larger than the uncompressed one, or that it fits within the amount of memory allocated, one may in conjunction with other constraints set

```
maxbits = 4^d * CHAR_BIT * sizeof(Type)
```

where Type is either float or double. The minbits parameter is useful only in fixed-rate mode–when minbits = maxbits, zero-bits are padded to blocks that compress to fewer than maxbits bits.

The effects of the above four parameters are best explained in terms of the three main compression modes supported by zfp, described below.

6.2 Fixed-Rate Mode

In fixed-rate mode, each *d*-dimensional compressed block of 4^d values is stored using a fixed number of bits given by the parameter `zfp_stream.maxbits`. This number of compressed bits per *block* is amortized over the 4^d values to give a *rate* in bits per *value*:

```
rate = maxbits / 4^d
```

This rate is specified in the *zfp executable* via the -r option, and programmatically via `zfp_stream_set_rate()`, as a floating-point value. Fixed-rate mode can also be achieved via the expert mode interface by setting

```
minbits = maxbits = (1 << (2 * d)) * rate
maxprec = ZFP_MAX_PREC
minexp = ZFP_MIN_EXP
```

Note that each block stores a bit to indicate whether the block is empty, plus a common exponent. Hence `zfp_stream.maxbits` must be at least 9 for single precision and 12 for double precision.

Fixed-rate mode is needed to support random access to blocks, and also is the mode used in the implementation of zfp's *compressed arrays*. Fixed-rate mode also ensures a predictable memory/storage footprint, but usually results in far worse accuracy per bit than the variable-rate fixed-precision and fixed-accuracy modes. **Use fixed-rate mode only if you have to bound the compressed size or need random access to blocks**.

6.3 Fixed Precision

In fixed-precision mode, the number of bits used to encode a block may vary, but the number of bit planes (i.e. the precision) encoded for the transform coefficients is fixed. To achieve the desired precision, use option -p with the *zfp executable* or call `zfp_stream_set_precision()`. In expert mode, fixed precision is achieved by specifying the precision in `zfp_stream.maxprec` and fully relaxing the size constraints, i.e.,

```
minbits = ZFP_MIN_BITS
maxbits = ZFP_MAX_BITS
maxprec = precision
minexp = ZFP_MIN_EXP
```

Fixed-precision mode is preferable when relative rather than absolute errors matter.

6.4 Fixed Accuracy

In fixed-accuracy mode, all transform coefficient bit planes up to a minimum bit plane number are encoded. (The actual minimum bit plane is not necessarily `zfp_stream.minexp`, but depends on the dimensionality, d, of the data. The reason for this is that the inverse transform incurs range expansion, and the amount of expansion depends on the number of dimensions.) Thus, `zfp_stream.minexp` should be interpreted as the base-2 logarithm of an absolute error tolerance. In other words, given an uncompressed value, f, and a reconstructed value, g, the absolute difference $| f - g |$ is at most 2^{minexp}. (Note that it is not possible to guarantee error tolerances smaller than machine epsilon relative to the largest value within a block.) This error tolerance is not always tight (especially for 3D arrays), but can conservatively be set so that even for worst-case inputs the error tolerance is respected. To achieve fixed accuracy to within ‘tolerance’, use option -a with the *zfp executable* or call `zfp_stream_set_accuracy()`. The corresponding expert mode parameters are:

```
minbits = ZFP_MIN_BITS
maxbits = ZFP_MAX_BITS
maxprec = ZFP_MAX_PREC
minexp = floor(log2(tolerance))
```

As in fixed-precision mode, the number of bits used per block is not fixed but is dictated by the data. Use tolerance = 0 to achieve near-lossless compression. Fixed-accuracy mode gives the highest quality (in terms of absolute error) for a given compression rate, and is preferable when random access is not needed.

CHAPTER 7

High-Level C API

The C API is broken down into a *high-level API*, which handles compression of entire arrays, and a *low-level-api* for processing individual blocks and managing the underlying bit stream.

The high-level API should be the API of choice for applications that compress and decompress entire arrays. A *low-level API* exists for processing individual, possibly partial blocks as well as reduced-precision integer data less than 32 bits wide.

The following sections are available:

- *Macros*
- *Types*
- *Constants*
- *Functions*
  - *Compressed Stream*
  - *Compression Parameters*
  - *Array Metadata*
  - *Compression and Decompression*

7.1 Macros

ZFP_VERSION_MAJOR

ZFP_VERSION_MINOR

ZFP_VERSION_PATCH

ZFP_VERSION

ZFP_VERSION_STRING

Macros identifying the zfp library version. *ZFP_VERSION* is a single integer constructed from the previous three macros. *ZFP_VERSION_STRING* is a string literal. See also *zfp_library_version* and *zfp_version_string*.

ZFP_CODEC

Macro identifying the version of the compression CODEC. See also *zfp_codec_version*.

ZFP_MIN_BITS

ZFP_MAX_BITS

ZFP_MAX_PREC

ZFP_MIN_EXP

Default compression parameter settings that impose no constraints. The largest possible compressed block size, corresponding to 3D blocks of doubles, is given by *ZFP_MAX_BITS*. See also *zfp_stream*.

ZFP_HEADER_MAGIC

ZFP_HEADER_META

ZFP_HEADER_MODE

ZFP_HEADER_FULL

Bit masks for specifying which portions of a header to output (if any). These constants should be bitwise ORed together. Use *ZFP_HEADER_FULL* to output all header information available. The compressor and decompressor must agree on which parts of the header to read/write.

ZFP_HEADER_META in essence encodes the information stored in the *zfp_field* struct, while *ZFP_HEADER_MODE* encodes the compression parameters stored in the *zfp_stream* struct. The magic can be used to uniquely identify the stream as a zfp stream, and includes the CODEC version.

See *zfp_read_header()* and *zfp_write_header()* for how to read and write header information.

ZFP_MAGIC_BITS

ZFP_META_BITS

ZFP_MODE_SHORT_BITS

ZFP_MODE_LONG_BITS

ZFP_HEADER_MAX_BITS

ZFP_MODE_SHORT_MAX

Number of bits used by each portion of the header. These macros are primarily informational and should not be accessed by the user through the high-level API. For most common compression parameter settings, only *ZFP_MODE_SHORT_BITS* bits of header information are stored to encode the mode (see *zfp_stream_mode()*).

7.2 Types

zfp_stream

The *zfp_stream* struct encapsulates all information about the compressed stream for a single block or a collection of blocks that represent an array. See the section on *compression modes* for a description of the members of this struct.

```
typedef struct {
  uint minbits;      // minimum number of bits to store per block
  uint maxbits;      // maximum number of bits to store per block
  uint maxprec;      // maximum number of bit planes to store
  int minexp;        // minimum floating point bit plane number to store
  bitstream* stream; // compressed bit stream
} zfp_stream;
```

zfp_type

Enumerates the scalar types supported by the compressor, and is used to describe the uncompressed array. The compressor and decompressor must use the same *zfp_type*, e.g. one cannot compress doubles and decompress to floats or integers.

```
typedef enum {
  zfp_type_none   = 0, // unspecified type
  zfp_type_int32  = 1, // 32-bit signed integer
  zfp_type_int64  = 2, // 64-bit signed integer
  zfp_type_float  = 3, // single precision floating point
  zfp_type_double = 4  // double precision floating point
} zfp_type;
```

zfp_field

The uncompressed array is described by the *zfp_field* struct, which encodes the array's scalar type, dimensions, and memory layout.

```
typedef struct {
  zfp_type type;   // scalar type (e.g. int32, double)
  uint nx, ny, nz; // sizes (zero for unused dimensions)
  int sx, sy, sz;  // strides (zero for contiguous array a[nz][ny][nx])
  void* data;      // pointer to array data
} zfp_field;
```

For example, a static multidimensional C array declared as

```
double array[n1][n2][n3];
```

would be described by a *zfp_field* with members

```
type = zfp_type_double;
nx = n3; ny = n2; nz = n1;
sx = 1; sy = n3; sz = n2 * n3;
data = &array[0][0][0];
```

7.3 Constants

const uint **zfp_codec_version**

The version of the compression CODEC implemented by this version of the zfp library. The library can decompress files generated by the same CODEC only. To ensure that the `zfp.h` header matches the binary library linked to, *zfp_codec_version* should match *ZFP_CODEC*.

const uint **zfp_library_version**

The library version. The binary library and headers are compatible if *zfp_library_version* matches *ZFP_VERSION*.

const char* const **`zfp_version_string`**

A constant string representing the zfp library version and release date. One can search for this string in executables and libraries that use zfp to determine which version of the library the application was compiled against.

7.4 Functions

size_t **`zfp_type_size`** (*zfp_type type*)

Return byte size of the given scalar type, e.g. `zfp_type_size(zfp_type_float) = 4`.

7.4.1 Compressed Stream

*zfp_stream** **`zfp_stream_open`** (*bitstream* stream*)

Allocate compressed stream and associate it with bit stream for reading and writing bits to/from memory. *stream* may be `NULL` and attached later via `zfp_stream_set_bit_stream()`.

void **`zfp_stream_close`** (*zfp_stream* stream*)

Close and deallocate compressed stream. This does not affect the attached bit stream.

*bitstream** **`zfp_stream_bit_stream`** (const *zfp_stream* stream*)

Return bit stream associated with compressed stream.

uint64 **`zfp_stream_mode`** (const *zfp_stream* zfp*)

Return compact encoding of compression parameters. If the return value is no larger than `ZFP_MODE_SHORT_MAX`, then the least significant `ZFP_MODE_SHORT_BITS` (12 in the current version) suffice to encode the parameters. Otherwise all 64 bits are needed, and the low `ZFP_MODE_SHORT_BITS` bits will be all ones. Thus, this variable-length encoding can be used to economically encode and decode the compression parameters, which is especially important if the parameters are to vary spatially over small regions. Such spatially adaptive coding would have to be implemented via the low-level API.

void **`zfp_stream_params`** (const *zfp_stream* stream*, uint* *minbits*, uint* *maxbits*, uint* *maxprec*, int* *minexp*)

Query *compression parameters*. For any parameter not needed, pass `NULL` for the corresponding pointer.

size_t **`zfp_stream_compressed_size`** (const *zfp_stream* stream*)

Number of bytes of compressed storage. This function returns the current byte offset within the bit stream from the beginning of the bit stream memory buffer. To ensure all buffered compressed data has been output call `zfp_stream_flush()` first.

size_t **`zfp_stream_maximum_size`** (const *zfp_stream* stream*, const *zfp_field* field*)

Conservative estimate of the compressed byte size for the compression parameters stored in *stream* and the array whose scalar type and dimensions are given by *field*. This function may be used to determine how large a memory buffer to allocate to safely hold the entire compressed array.

void **`zfp_stream_set_bit_stream`** (*zfp_stream* stream*, *bitstream* bs*)

Associate bit stream with compressed stream.

void **`zfp_stream_rewind`** (*zfp_stream* stream*)

Rewind bit stream to beginning for compression or decompression.

7.4.2 Compression Parameters

double **`zfp_stream_set_rate`** (*zfp_stream* stream*, double *rate*, *zfp_type type*, uint *dims*, int *wra*)

Set *rate* for *fixed-rate mode* in compressed bits per value. The target scalar *type* and array *dimensionality* are needed to correctly translate the rate to the number of bits per block. The parameter *wra* should be nonzero if random access writes of blocks into the compressed bit stream is needed, for example for implementing zfp's *compressed arrays*. This requires blocks to be aligned on *bit stream word* boundaries, and therefore constrains the rate. The closest supported rate is returned, which may differ from the desired rate.

uint **`zfp_stream_set_precision`** (*zfp_stream** *stream*, uint *precision*)
Set *precision* for *fixed-precision mode*. The precision specifies how many uncompressed bits per value to store, and indirectly governs the relative error. The actual precision is returned, e.g. in case the desired precision is out of range. To preserve a certain floating-point mantissa or integer precision in the decompressed data, see *FAQ #21*.

double **`zfp_stream_set_accuracy`** (*zfp_stream** *stream*, double *tolerance*)
Set absolute error *tolerance* for *fixed-accuracy mode*. The tolerance ensures that values in the decompressed array differ from the input array by no more than this tolerance (in all but exceptional circumstances; see *FAQ #17*). This compression mode should be used only with floating-point (not integer) data.

int **`zfp_stream_set_mode`** (*zfp_stream** *stream*, uint64 *mode*)
Set all compression parameters from compact integer representation. See `zfp_stream_mode()` for how to encode the parameters. The return value is nonzero upon success.

int **`zfp_stream_set_params`** (*zfp_stream** *stream*, uint *minbits*, uint *maxbits*, uint *maxprec*, int *minexp*)
Set all compression parameters directly. See the section on *expert mode* for a discussion of the parameters. The return value is nonzero upon success.

7.4.3 Array Metadata

*zfp_field** **`zfp_field_alloc`** ()
Allocates and returns a default initialized `zfp_field` struct. The caller must free this struct using `zfp_field_free()`.

*zfp_field** **`zfp_field_1d`** (void* *pointer*, *zfp_type* *type*, uint *nx*)
Allocate and return a field struct that describes an existing 1D array, `a[nx]`, of *nx* uncompressed scalars of given *type* stored at *pointer*, which may be `NULL` and specified later.

*zfp_field** **`zfp_field_2d`** (void* *pointer*, *zfp_type* *type*, uint *nx*, uint *ny*)
Allocate and return a field struct that describes an existing 2D array, `a[ny][nx]`, of $nx \times ny$ uncompressed scalars of given *type* stored at *pointer*, which may be `NULL` and specified later.

*zfp_field** **`zfp_field_3d`** (void* *pointer*, *zfp_type* *type*, uint *nx*, uint *ny*, uint *nz*)
Allocate and return a field struct that describes an existing 3D array, `a[nz][ny][nx]`, of $nx \times ny \times nz$ uncompressed scalars of given *type* stored at *pointer*, which may be `NULL` and specified later.

void **`zfp_field_free`** (*zfp_field** *field*)
Free `zfp_field` struct previously allocated by one of the functions above.

void* **`zfp_field_pointer`** (const *zfp_field** *field*)
Return pointer to the first scalar in the array.

zfp_type **`zfp_field_type`** (const *zfp_field** *field*)
Return array scalar type.

uint **`zfp_field_precision`** (const *zfp_field** *field*)
Return scalar precision in number of bits, e.g. 32 for `zfp_type_float`.

uint **`zfp_field_dimensionality`** (const *zfp_field** *field*)
Return array dimensionality (1, 2, or 3).

size_t **`zfp_field_size`** (const *zfp_field** *field*, uint* *size*)
Return total number of scalars stored in the array, e.g. $nx \times ny \times nz$ for a 3D array. If *size* is not `NULL`, then store the number of scalars for each dimension, e.g. `size[0] = nx; size[1] = ny; size[2] = nz` for a 3D array.

int **`zfp_field_stride`** (const *zfp_field** *field*, int* *stride*)
Return zero if array is stored contiguously; nonzero if it is strided. If *stride* is not `NULL`, then store the stride for each dimension, e.g. `stride[0] = sx; stride[1] = sy; stride[2] = sz` for a 3D array. See below for more information on strides.

uint64 **`zfp_field_metadata`** (const *zfp_field** *field*)
Return 52-bit compact encoding of the scalar type and array dimensions.

void **`zfp_field_set_pointer`** (*zfp_field** *field*, void* *pointer*)
Set pointer to first scalar in the array.

zfp_type **`zfp_field_set_type`** (*zfp_field** *field*, *zfp_type* *type*)
Set array scalar type.

void **`zfp_field_set_size_1d`** (*zfp_field** *field*, uint *nx*)
Specify dimensions of 1D array `a[nx]`.

void **`zfp_field_set_size_2d`** (*zfp_field** *field*, uint *nx*, uint *ny*)
Specify dimensions of 2D array `a[ny][nx]`.

void **`zfp_field_set_size_3d`** (*zfp_field** *field*, uint *nx*, uint *ny*, uint *nz*)
Specify dimensions of 3D array `a[nz][ny][nx]`.

void **`zfp_field_set_stride_1d`** (*zfp_field** *field*, int *sx*)
Specify stride for 1D array: `sx = &a[1] - &a[0]`.

void **`zfp_field_set_stride_2d`** (*zfp_field** *field*, int *sx*, int *sy*)
Specify strides for 2D array: `sx = &a[0][1] - &a[0][0]; sy = &a[1][0] - &a[0][0]`.

void **`zfp_field_set_stride_3d`** (*zfp_field** *field*, int *sx*, int *sy*, int *sz*)
Specify strides for 3D array: `sx = &a[0][0][1] - &a[0][0][0]; sy = &a[0][1][0] - &a[0][0][0]; sz = &a[1][0][0] - &a[0][0][0]`.

int **`zfp_field_set_metadata`** (*zfp_field** *field*, uint64 *meta*)
Specify array scalar type and dimensions from compact 52-bit representation. Returns nonzero upon success. See `zfp_field_metadata()` for how to encode *meta*.

7.4.4 Compression and Decompression

size_t **`zfp_compress`** (*zfp_stream** *stream*, const *zfp_field** *field*)
Compress the whole array described by *field* using parameters given by *stream* and then flush the stream. The number of bytes of compressed storage is returned, if the stream were rewound before compression, and otherwise the current byte offset within the bit stream. Zero is returned if compression failed.

int **`zfp_decompress`** (*zfp_stream** *stream*, *zfp_field** *field*)
Decompress from *stream* to array described by *field* and align the stream on the next word boundary. Nonzero is returned upon success.

size_t **`zfp_write_header`** (*zfp_stream** *stream*, const *zfp_field** *field*, uint *mask*)
Write an optional header to the stream that encodes compression parameters, array metadata, etc. The header information written is determined by the bit *mask* (see `macros`). The return value is the number of bits written, or zero upon failure.

size_t **`zfp_read_header`** (*zfp_stream** *stream*, *zfp_field** *field*, uint *mask*)
Read header if one was previously written using `zfp_write_header()`. The return value is the number of bits read, or zero upon failure. The caller must ensure that the bit *mask* agrees between header read and write calls.

CHAPTER 8

Low-Level C API

The low-level C API provides functionality for compressing individual *d*-dimensional blocks of up to 4^d values. If a block is not complete, i.e. contains fewer than 4^d values, then zfp's partial block support should be favored over padding the block with, say, zeros or other fill values. The blocks (de)compressed need not be contiguous, but can be gathered from or scatter to a larger array by setting appropriate strides.

The following topics are available:

- *Stream Manipulation*
- *Encoder*
  - *1D Data*
  - *2D Data*
  - *3D Data*
- *Decoder*
  - *1D Data*
  - *2D Data*
  - *3D Data*
- *Utility Functions*

8.1 Stream Manipulation

size_t **`zfp_stream_flush`**(*zfp_stream* stream*)

Flush bit stream to write out any buffered bits. This function must be must be called after the last encode call. The bit stream is aligned on a stream word boundary following this call. The number of zero-bits written, if any, is returned.

size_t **`zfp_stream_align`**(*zfp_stream* stream*)

Align bit stream on next word boundary. This function is analogous to `zfp_stream_flush()`, but for decoding. That is, wherever the encoder flushes the stream, the decoder should align it to ensure synchronization between encoder and decoder. The number of bits skipped, if any, is returned.

8.2 Encoder

A function is available for encoding whole or partial blocks of each scalar type and dimensionality. These functions return the number of bits of compressed storage for the block being encoded, or zero upon failure.

8.2.1 1D Data

uint **zfp_encode_block_int32_1** (*zfp_stream* stream*, const int32* *block*)

uint **zfp_encode_block_int64_1** (*zfp_stream* stream*, const int64* *block*)

uint **zfp_encode_block_float_1** (*zfp_stream* stream*, const float* *block*)

uint **zfp_encode_block_double_1** (*zfp_stream* stream*, const double* *block*)
Encode 1D contiguous block of 4 values.

uint **zfp_encode_block_strided_int32_1** (*zfp_stream* stream*, const int32* *p*, int *sx*)

uint **zfp_encode_block_strided_int64_1** (*zfp_stream* stream*, const int64* *p*, int *sx*)

uint **zfp_encode_block_strided_float_1** (*zfp_stream* stream*, const float* *p*, int *sx*)

uint **zfp_encode_block_strided_double_1** (*zfp_stream* stream*, const double* *p*, int *sx*)
Encode 1D complete block from strided array with stride *sx*.

uint **zfp_encode_partial_block_strided_int32_1** (*zfp_stream* stream*, const int32* *p*, uint *nx*, int *sx*)

uint **zfp_encode_partial_block_strided_int64_1** (*zfp_stream* stream*, const int64* *p*, uint *nx*, int *sx*)

uint **zfp_encode_partial_block_strided_float_1** (*zfp_stream* stream*, const float* *p*, uint *nx*, int *sx*)

uint **zfp_encode_partial_block_strided_double_1** (*zfp_stream* stream*, const double* *p*, uint *nx*, int *sx*)
Encode 1D partial block of size *nx* from strided array with stride *sx*.

8.2.2 2D Data

uint **zfp_encode_block_int32_2** (*zfp_stream* stream*, const int32* *block*)

uint **zfp_encode_block_int64_2** (*zfp_stream* stream*, const int64* *block*)

uint **zfp_encode_block_float_2** (*zfp_stream* stream*, const float* *block*)

uint **zfp_encode_block_double_2** (*zfp_stream* stream*, const double* *block*)
Encode 2D contiguous block of 4 × 4 values.

uint **zfp_encode_block_strided_int32_2** (*zfp_stream* stream*, const int32* *p*, int *sx*, int *sy*)

uint **zfp_encode_block_strided_int64_2** (*zfp_stream* stream*, const int64* *p*, int *sx*, int *sy*)

uint **zfp_encode_block_strided_float_2** (*zfp_stream* stream*, const float* *p*, int *sx*, int *sy*)

uint **zfp_encode_block_strided_double_2** (*zfp_stream* stream*, const double* *p*, int *sx*, int *sy*)
Encode 2D complete block from strided array with strides *sx* and *sy*.

uint **`zfp_encode_partial_block_strided_int32_2`** (*zfp_stream** *stream*, const int32* *p*, uint *nx*, uint *ny*, int *sx*, int *sy*)

uint **`zfp_encode_partial_block_strided_int64_2`** (*zfp_stream** *stream*, const int64* *p*, uint *nx*, uint *ny*, int *sx*, int *sy*)

uint **`zfp_encode_partial_block_strided_float_2`** (*zfp_stream** *stream*, const float* *p*, uint *nx*, uint *ny*, int *sx*, int *sy*)

uint **`zfp_encode_partial_block_strided_double_2`** (*zfp_stream** *stream*, const double* *p*, uint *nx*, uint *ny*, int *sx*, int *sy*)

Encode 2D partial block of size $nx \times ny$ from strided array with strides *sx* and *sy*.

8.2.3 3D Data

uint **`zfp_encode_block_int32_3`** (*zfp_stream** *stream*, const int32* *block*)

uint **`zfp_encode_block_int64_3`** (*zfp_stream** *stream*, const int64* *block*)

uint **`zfp_encode_block_float_3`** (*zfp_stream** *stream*, const float* *block*)

uint **`zfp_encode_block_double_3`** (*zfp_stream** *stream*, const double* *block*)

Encode 3D contiguous block of $4 \times 4 \times 4$ values.

uint **`zfp_encode_block_strided_int32_3`** (*zfp_stream** *stream*, const int32* *p*, int *sx*, int *sy*, int *sz*)

uint **`zfp_encode_block_strided_int64_3`** (*zfp_stream** *stream*, const int64* *p*, int *sx*, int *sy*, int *sz*)

uint **`zfp_encode_block_strided_float_3`** (*zfp_stream** *stream*, const float* *p*, int *sx*, int *sy*, int *sz*)

uint **`zfp_encode_block_strided_double_3`** (*zfp_stream** *stream*, const double* *p*, int *sx*, int *sy*, int *sz*)

Encode 3D complete block from strided array with strides *sx*, *sy*, and *sz*.

uint **`zfp_encode_partial_block_strided_int32_3`** (*zfp_stream** *stream*, const int32* *p*, uint *nx*, uint *ny*, uint *nz*, int *sx*, int *sy*, int *sz*)

uint **`zfp_encode_partial_block_strided_int64_3`** (*zfp_stream** *stream*, const int64* *p*, uint *nx*, uint *ny*, uint *nz*, int *sx*, int *sy*, int *sz*)

uint **`zfp_encode_partial_block_strided_float_3`** (*zfp_stream** *stream*, const float* *p*, uint *nx*, uint *ny*, uint *nz*, int *sx*, int *sy*, int *sz*)

uint **`zfp_encode_partial_block_strided_double_3`** (*zfp_stream** *stream*, const double* *p*, uint *nx*, uint *ny*, uint *nz*, int *sx*, int *sy*, int *sz*)

Encode 3D partial block of size $nx \times ny \times nz$ from strided array with strides *sx*, *sy*, and *sz*.

8.3 Decoder

Each function below decompresses a single block and returns the number of bits of compressed storage consumed. See corresponding encoder functions above for further details.

8.3.1 1D Data

uint **`zfp_decode_block_int32_1`** (*zfp_stream** *stream*, int32* *block*)

uint **`zfp_decode_block_int64_1`** (*zfp_stream** *stream*, int64* *block*)

uint **`zfp_decode_block_float_1`** (*zfp_stream** *stream*, float* *block*) uint **zfp_decode_block_double_1** (*zfp_stream** *stream*, double* *block*)

Decode 1D contiguous block of 4 values.

uint **zfp_decode_block_strided_int32_1** (*zfp_stream** *stream*, int32* *p*, int *sx*)

uint **zfp_decode_block_strided_int64_1** (*zfp_stream** *stream*, int64* *p*, int *sx*)

uint **zfp_decode_block_strided_float_1** (*zfp_stream** *stream*, float* *p*, int *sx*)

uint **zfp_decode_block_strided_double_1** (*zfp_stream** *stream*, double* *p*, int *sx*)

Decode 1D complete block to strided array with stride *sx*.

`uint zfp_decode_partial_block_strided_int32_1(zfp_stream* stream, int32* p, uint nx, int s`

`uint zfp_decode_partial_block_strided_int64_1(zfp_stream* stream, int64* p, uint nx, int s`

`uint zfp_decode_partial_block_strided_float_1(zfp_stream* stream, float* p, uint nx, int s`

`uint zfp_decode_partial_block_strided_double_1(zfp_stream* stream, double* p, uint nx, int`

Decode 1D partial block of size *nx* to strided array with stride *sx*.

8.3.2 2D Data

`uint zfp_decode_block_int32_2(zfp_stream* stream, int32* block);`

`uint zfp_decode_block_int64_2(zfp_stream* stream, int64* block);`

`uint zfp_decode_block_float_2(zfp_stream* stream, float* block);`

uint **zfp_decode_block_double_2** (*zfp_stream** *stream*, double* *block*)

Decode 2D contiguous block of 4×4 values.

`uint zfp_decode_block_strided_int32_2(zfp_stream* stream, int32* p, int sx, int sy);`

`uint zfp_decode_block_strided_int64_2(zfp_stream* stream, int64* p, int sx, int sy);`

`uint zfp_decode_block_strided_float_2(zfp_stream* stream, float* p, int sx, int sy);`

`uint zfp_decode_block_strided_double_2(zfp_stream* stream, double* p, int sx, int sy);`

Decode 2D complete block to strided array with strides *sx* and *sy*.

`uint zfp_decode_partial_block_strided_int32_2(zfp_stream* stream, int32* p, uint nx, uint`

`uint zfp_decode_partial_block_strided_int64_2(zfp_stream* stream, int64* p, uint nx, uint`

`uint zfp_decode_partial_block_strided_float_2(zfp_stream* stream, float* p, uint nx, uint`

`uint zfp_decode_partial_block_strided_double_2(zfp_stream* stream, double* p, uint nx, uin`

Decode 2D partial block of size $nx \times ny$ to strided array with strides *sx* and *sy*.

8.3.3 3D Data

uint **zfp_decode_block_int32_3** (*zfp_stream** *stream*, int32* *block*)

uint **zfp_decode_block_int64_3** (*zfp_stream** *stream*, int64* *block*)

uint **zfp_decode_block_float_3** (*zfp_stream** *stream*, float* *block*)

uint **zfp_decode_block_double_3** (*zfp_stream** *stream*, double* *block*)

Decode 3D contiguous block of $4 \times 4 \times 4$ values.

uint **zfp_decode_block_strided_int32_3** (*zfp_stream** *stream*, int32* *p*, int *sx*, int *sy*, int *sz*)

uint **zfp_decode_block_strided_int64_3** (*zfp_stream** *stream*, int64* *p*, int *sx*, int *sy*, int *sz*) uint **`zfp_decode_block_strided_float_3`** (*zfp_stream** *stream*, float* *p*, int *sx*, int *sy*, int *sz*)

uint **`zfp_decode_block_strided_double_3`** (*zfp_stream** *stream*, double* *p*, int *sx*, int *sy*, int *sz*)
Decode 3D complete block to strided array with strides *sx*, *sy*, and *sz*.

uint **`zfp_decode_partial_block_strided_int32_3`** (*zfp_stream** *stream*, int32* *p*, uint *nx*, uint *ny*, uint *nz*, int *sx*, int *sy*, int *sz*)

uint **`zfp_decode_partial_block_strided_int64_3`** (*zfp_stream** *stream*, int64* *p*, uint *nx*, uint *ny*, uint *nz*, int *sx*, int *sy*, int *sz*)

uint **`zfp_decode_partial_block_strided_float_3`** (*zfp_stream** *stream*, float* *p*, uint *nx*, uint *ny*, uint *nz*, int *sx*, int *sy*, int *sz*)

uint **`zfp_decode_partial_block_strided_double_3`** (*zfp_stream** *stream*, double* *p*, uint *nx*, uint *ny*, uint *nz*, int *sx*, int *sy*, int *sz*)
Decode 3D partial block of size $nx \times ny \times nz$ to strided array with strides *sx*, *sy*, and *sz*.

8.4 Utility Functions

These functions convert 8- and 16-bit signed and unsigned integer data to (by promoting) and from (by demoting) 32-bit integers that can be (de)compressed by zfp's `int32` functions. These conversion functions are preferred over simple casting since they eliminate the redundant leading zeros that would otherwise have to be compressed, and they apply the appropriate bias for unsigned integer data.

void **`zfp_promote_int8_to_int32`** (int32* *oblock*, const int8* *iblock*, uint *dims*)

void **`zfp_promote_uint8_to_int32`** (int32* *oblock*, const uint8* *iblock*, uint *dims*)

void **`zfp_promote_int16_to_int32`** (int32* *oblock*, const int16* *iblock*, uint *dims*)

void **`zfp_promote_uint16_to_int32`** (int32* *oblock*, const uint16* *iblock*, uint *dims*)
Convert *dims*-dimensional contiguous block to 32-bit integer type.

void **`zfp_demote_int32_to_int8`** (int8* *oblock*, const int32* *iblock*, uint *dims*)

void **`zfp_demote_int32_to_uint8`** (uint8* *oblock*, const int32* *iblock*, uint *dims*)

void **`zfp_demote_int32_to_int16`** (int16* *oblock*, const int32* *iblock*, uint *dims*)

void **`zfp_demote_int32_to_uint16`** (uint16* *oblock*, const int32* *iblock*, uint *dims*)
Convert *dims*-dimensional contiguous block from 32-bit integer type.

CHAPTER 9

Bit Stream API

zfp relies on low-level functions for bit stream I/O, e.g. for reading/writing single bits or groups of bits. zfp's bit streams support random access (with some caveats) and, optionally, strided access. The functions read from and write to main memory allocated by the user. Buffer overruns are for performance reasons not guarded against.

From an implementation standpoint, bit streams are read from and written to memory in increments of *words* of bits. The constant power-of-two word size is configured at compile time, and is limited to 8, 16, 32, or 64 bits.

The bit stream API is publicly exposed and may be used to write additional information such as metadata into the zfp compressed stream, as well as to manipulate whole or partial bit streams. Moreover, we envision releasing the bit stream functions as a separate library in the future that may be used, for example, in other compressors.

Stream readers and writers are synchronized by making corresponding calls. For each write call, there is a corresponding read call. This ensures that reader and writer agree on the position within the stream and the number of bits buffered, if any. The API below reflects this duality.

A bit stream is either in read or write mode, or either, if rewound to the beginning. When in read mode, only read calls should be made, and similarly for write mode.

9.1 Strided Streams

Bit streams may be strided by sequentially reading/writing a few words at a time and then skipping over some user-specified number of words. This allows, for instance, zfp to interleave the first few bits of all compressed blocks in order to support progressive access. To enable strided access, which does carry a small performance penalty, the macro `BIT_STREAM_STRIDED` must be defined during compilation.

Strides are specified in terms of a *block size*–a power-of-two number of contiguous words–and a *delta*, which specifies how many words to advance the stream by to get to the next contiguous block. These bit stream blocks are entirely independent of the 4^d blocks used for compression in zfp. Setting *delta* to zero ensures a non-strided, sequential layout.

9.2 Macros

Two compile-time macros are used to influence the behavior: *BIT_STREAM_WORD_TYPE* and *BIT_STREAM_STRIDED*. These are documented in the *installation* section.

9.3 Types

word

Bits are buffered and read/written in units of words. By default, the bit stream word type is 64 bits, but may be set to 8, 16, or 32 bits by setting the macro *BIT_STREAM_WORD_TYPE* to `uint8`, `uint16`, or `uint32`, respectively. Larger words tend to give higher throughput, while 8-bit words are needed to ensure endian independence (see FAQ *#11*).

bitstream

The bit stream struct maintains all the state associated with a bit stream. This struct is passed to all bit stream functions. Its members should not be accessed directly.

```
struct bitstream {
  uint bits;        // number of buffered bits (0 <= bits < word size)
  word buffer;      // buffer for incoming/outgoing bits (buffer < 2^bits)
  word* ptr;        // pointer to next word to be read/written
  word* begin;      // beginning of stream
  word* end;        // end of stream (currently unused)
  size_t mask;      // one less the block size in number of words (if BIT_STREAM_
↪STRIDED)
  ptrdiff_t delta; // number of words between consecutive blocks (if BIT_STREAM_
↪STRIDED)
};
```

9.4 Constants

const size_t **stream_word_bits**

The number of bits in a word. The size of a flushed bit stream will be a multiple of this number of bits. See *BIT_STREAM_WORD_TYPE*.

9.5 Functions

*bitstream** **stream_open** (void* *buffer*, size_t *bytes*)

Allocate a `bitstream` struct and associate it with the memory buffer allocated by the caller.

void **stream_close** (*bitstream** *stream*)

Close the bit stream and deallocate *stream*.

void* **stream_data** (const *bitstream** *stream*)

Return pointer to the beginning of bit stream *stream*.

size_t **stream_size** (const *bitstream** *stream*)

Return position of stream pointer in number of bytes, which equals the end of stream if no seeks have been made. Note that additional bits may be buffered and not reported unless the stream has been flushed.

size_t **stream_capacity** (const *bitstream** *stream*)

Return byte size of memory buffer associated with *stream*.

uint **`stream_read_bit`** (*bitstream* stream*)

Read a single bit from *stream*.

uint **`stream_write_bit`** (*bitstream* stream*, uint *bit*)

Write the least significant bit of *bit* to *stream*. *bit* should be one of 0 or 1.

uint64 **`stream_read_bits`** (*bitstream* stream*, uint *n*)

Read and return $0 \leq n \leq 64$ bits from *stream*.

uint64 **`stream_write_bits`** (*bitstream* stream*, uint64 *value*, uint *n*)

Write $0 \leq n \leq 64$ low bits of *value* to *stream*. Return any remaining bits from *value*, i.e. *value* >> *n*.

size_t **`stream_rtell`** (const *bitstream* stream*)

Return bit offset to next bit to be read.

size_t **`stream_wtell`** (const *bitstream* stream*)

Return bit offset to next bit to be written.

void **`stream_rewind`** (*bitstream* stream*)

Rewind stream to beginning of memory buffer. Following this call, the stream may either be read or written.

void **`stream_rseek`** (*bitstream* stream*, size_t *offset*)

Position stream for reading at given bit offset. This places the stream in read mode.

void **`stream_wseek`** (*bitstream* stream*, size_t *offset*)

Position stream for writing at given bit offset. This places the stream in write mode.

void **`stream_skip`** (*bitstream* stream*, uint *n*)

Skip over the next *n* bits, i.e. without reading them.

void **`stream_pad`** (*bitstream* stream*, uint *n*)

Append *n* zero-bits to *stream*.

size_t **`stream_align`** (*bitstream* stream*)

Align stream on next word boundary by skipping bits.

size_t **`stream_flush`** (*bitstream* stream*)

Write out any remaining buffered bits.

size_t **`stream_stride_block`** (const *bitstream* stream*)

Return stream block size in number of words. The block size is always one word unless strided streams are enabled. See *Strided Streams* for more information.

ptrdiff_t **`stream_stride_delta`** (const *bitstream* stream*)

Return stream delta in number of words between blocks. See *Strided Streams* for more information.

int **`stream_set_stride`** (*bitstream* stream*, size_t *block*, ptrdiff_t *delta*)

Set block size in number of words and spacing in number of blocks for strided access. Requires `BIT_STREAM_STRIDED`.

CHAPTER 10

Compressed Arrays

zfp's compressed arrays are C++ classes that implement random-accessible single- and multi-dimensional floating-point arrays whose storage size, specified in number of bits per array element, is set by the user. Such arbitrary storage is achieved via zfp's lossy *fixed-rate compression* mode, by partitioning each d-dimensional array into blocks of 4^d values and compressing each block to a fixed number of bits. The more smoothly the array values vary along each dimension, the more accurately zfp can represent them. In other words, these arrays are not suitable for representing data where adjacent elements are not correlated. Rather, the expectation is that the array represents a regularly sampled and predominantly continuous function, such as a temperature field in a physics simulation.

The *rate*, measured in number of bits per array element, can be specified in fractions of a bit (but see FAQs *#12* and *#18* for limitations). Note that array dimensions need not be multiples of four; zfp transparently handles partial blocks on array boundaries.

The C++ templated array classes are implemented entirely as header files that call the zfp C library to perform compression and decompression. These arrays cache decompressed blocks to reduce the number of compression and decompression calls. Whenever an array value is read, the corresponding block is first looked up in the cache, and if found the uncompressed value is returned. Otherwise the block is first decompressed and stored in the cache. Whenever an array element is written (whether actually modified or not), a "dirty bit" is set with its cached block to indicate that the block must be compressed back to persistent storage when evicted from the cache.

This section documents the public interface to the array classes, including base classes and member accessor classes like proxy references/pointers and iterators.

10.1 Array Classes

Currently there are six array classes for 1D, 2D, and 3D arrays, each of which can represent single- or double-precision values. Although these arrays store values in a form different from conventional single- and double-precision floating point, the user interacts with the arrays via floats and doubles.

The array classes can often serve as direct substitutes for C/C++ single- and multi-dimensional floating-point arrays and STL vectors, but have the benefit of allowing fine control over storage size. All classes below belong to the `zfp` namespace.

10.1.1 Base Class

class array

Virtual base class for common array functionality.

array::**array**(**const** *array* &*a*)

Copy constructor. Performs a deep copy. **NOTE: Deep copies are not yet fully implemented**.

array::**~array**()

Destructor.

array &*array*::**operator=**(**const** *array* &*a*)

Assignment operator–performs a deep copy of the whole array.

double *array*::**rate**() **const**

Return rate in bits per value.

double *array*::**set_rate**(double *rate*)

Set desired compression rate in bits per value. Return the closest rate supported. See *FAQ #12* and *FAQ #18* for discussions of the rate granularity. This method destroys the previous contents of the array.

virtual void *array*::**clear_cache**() **const**

Empty cache without compressing modified cached blocks, i.e. discard any cached updates to the array.

virtual void *array*::**flush_cache**() **const**

Flush cache by compressing all modified cached blocks back to persistent storage and emptying the cache. This method should be called before writing the compressed representation of the array to disk, for instance.

size_t *array*::**compressed_size**() **const**

Return number of bytes of storage for the compressed data. This amount does not include the small overhead of other class members or the size of the cache. Rather, it reflects the size of the memory buffer returned by `compressed_data()`.

uchar **array*::**compressed_data**() **const**

Return pointer to compressed data for read or write access. The size of the buffer is given by `compressed_size()`.

10.1.2 Common Methods

The following methods are common to 1D, 2D, and 3D arrays, but are implemented in the array class specific to each dimensionality rather than in the base class.

size_t *array*::**size**() **const**

Total number of elements in array, e.g. $nx \times ny \times nz$ for 3D arrays.

size_t *array*::**cache_size**() **const**

Return the cache size in number of bytes.

void *array*::**set_cache_size**(size_t *csize*)

Set minimum cache size in bytes. The actual size is always a power of two bytes and consists of at least one block. If *csize* is zero, then a default cache size is used, which requires the array dimensions to be known.

void *array*::**get**(Scalar **p*) **const**

Decompress entire array and store at p, for which sufficient storage must have been allocated. The uncompressed array is assumed to be contiguous (with default strides) and stored in the usual "row-major" order, i.e. with x varying faster than y and y varying faster than z.

void *array*::**set**(**const** Scalar **p*)

Initialize array by copying and compressing data stored at p. The uncompressed data is assumed to be stored as in the `get()` method.

const Scalar &*array*::**operator[]**(uint *index*) **const**

Return scalar stored at given flat index (inspector). For a 3D array, `index = x + nx * (y + ny * z)`.

reference *array*::**operator[]**(uint *index*)

Return *proxy reference* to scalar stored at given flat index (mutator). For a 3D array, `index = x + nx * (y + ny * z)`.

iterator *array*::**begin**()

Return iterator to beginning of array.

iterator *array*::**end**()

Return iterator to end of array. As with STL iterators, the end points to a virtual element just past the last valid array element.

10.1.3 1D, 2D, and 3D Arrays

Below are classes and methods specific to each array dimensionality. Since the classes and methods share obvious similarities regardless of dimensionality, only one generic description for all dimensionalities is provided.

template<typename **Scalar**>
class array1 : **public** *array*

template<typename **Scalar**>
class array2 : **public** *array*

template<typename **Scalar**>
class array3 : **public** *array*

This is a 1D/2D/3D array that inherits basic functionality from the generic `array` base class. The template argument, `Scalar`, specifies the floating type returned for array elements. The suffixes `f` and `d` can also be appended to each class to indicate float or double type, e.g. `array1f` is a synonym for `array1<float>`.

array1::**array1**()

array2::**array2**()

array3::**array3**()

Default constructor. Creates an empty array.

array1::**array1**(uint *n*, double *rate*, **const** Scalar **p* = 0, size_t *csize* = 0)

array2::**array2**(uint *nx*, uint *ny*, double *rate*, **const** Scalar **p* = 0, size_t *csize* = 0)

array3::**array3**(uint *nx*, uint *ny*, uint *nz*, double *rate*, **const** Scalar **p* = 0, size_t *csize* = 0)

Constructor of array with dimensions *n* (1D), $nx \times ny$ (2D), or $nx \times ny \times nz$ (3D) using *rate* bits per value, at least *csize* bytes of cache, and optionally initialized from flat, uncompressed array *p*. If *csize* is zero, a default cache size is chosen.

uint `array2`::**size_x**() **const**

uint `array2`::**size_y**() **const**

uint `array3`::**size_x**() **const**

uint `array3`::**size_y**() **const**

uint `array3`::**size_z**() **const**

Return array dimensions.

void `array1`::**resize**(uint *n*, bool *clear* = true)

void `array2`::**resize**(uint *nx*, uint *ny*, bool *clear* = true) void `array3::`**`resize`**(uint *nx*, uint *ny*, uint *nz*, bool *clear* = true)

Resize the array (all previously stored data will be lost). If *clear* is true, then the array elements are all initialized to zero.

`const` Scalar &`array1::`**`operator()`**(uint *i*) **`const`**

`const` Scalar &`array2::`**`operator()`**(uint *i*, uint *j*) **`const`**

`const` Scalar &`array3::`**`operator()`**(uint *i*, uint *j*, uint *k*) **`const`**

Return scalar stored at multi-dimensional index given by *i*, *j*, and *k* (inspector).

reference `array1::`**`operator()`**(uint *i*)

reference `array2::`**`operator()`**(uint *i*, uint *j*)

reference `array3::`**`operator()`**(uint *i*, uint *j*, uint *k*)

Return *proxy reference* to scalar stored at multi-dimensional index given by *i*, *j*, and *k* (mutator).

10.2 Caching

As mentioned above, the array classes maintain a software write-back cache of at least one uncompressed block. When a block in this cache is evicted (e.g. due to a conflict), it is compressed back to permanent storage only if it was modified while stored in the cache.

The size cache to use is specified by the user, and is an important parameter that needs careful consideration in order to balance the extra memory usage, performance, and quality (recall that data loss is incurred only when a block is evicted from the cache and compressed). Although the best choice varies from one application to another, we suggest allocating at least two "layers" of blocks, e.g. 2 × (*nx* / 4) × (*ny* / 4) blocks for 3D arrays, for applications that stream through the array and perform stencil computations such as gathering data from neighboring elements. This allows limiting the cache misses to compulsory ones. If the *csize* parameter provided to the constructor is set to zero bytes, then this default of two layers is used.

The cache size can be set during construction, or can be set at a later time via *`array::set_cache_size()`*. Note that if *csize* = 0, then the array dimensions must have already been specified for the default size to be computed correctly. When the cache is resized, it is first flushed if not already empty. The cache can also be flushed explicitly if desired by calling *`array::flush_cache()`*. To empty the cache without compressing any cached data, call `clear_cache()`. To query the byte size of the cache, use *`array::cache_size()`*.

By default a direct-mapped cache is used with a hash function that maps block indices to cache lines. A faster but more collision prone hash can be enabled by defining the preprocessor macro *`ZFP_WITH_CACHE_FAST_HASH`*. A two-way skew-associative cache is enabled by defining the preprocessor macro *`ZFP_WITH_CACHE_TWOWAY`*.

10.3 References

template<typename **`Scalar`**>
`class` `array1::`**`reference`**

template<typename **`Scalar`**>
`class` `array2::`**`reference`**

template<typename **`Scalar`**>
`class` `array3::`**`reference`**

Array *indexing operators* must return lvalue references that alias array elements and serve as vehicles for assigning values to those elements. Unfortunately, zfp cannot simply return a standard C++ reference (e.g. `float&`) to an uncompressed array element since the element in question may exist only in compressed form or as a transient cached entry that may be invalidated (evicted) at any point.

To address this, zfp provides *proxies* for references and pointers that act much like regular references and pointers, but which refer to elements by array and index rather than by memory address. When assigning to an array element through such a proxy reference or pointer, the corresponding element is decompressed to cache (if not already cached) and immediately updated.

zfp references may be freely passed to other functions and they remain valid during the lifetime of the corresponding array element. One may also take the address of a reference, which yields a *proxy pointer*. When a reference appears as an rvalue in an expression, it is implicitly converted to a value.

The following operators are defined for zfp references. They act on the referenced array element in the same manner as operators defined for conventional C++ references.

reference `reference::`**`operator=`**`(`**`const`** reference *&ref*`)`

> Assignment (copy) operator. The referenced element, *elem*, is assigned the value stored at the element referenced by *ref*. Return `*this`.

reference `reference::`**`operator=`**`(`Scalar *val*`)`

reference `reference::`**`operator+=`**`(`Scalar *val*`)`

reference `reference::`**`operator-=`**`(`Scalar *val*`)`

reference `reference::`**`operator*=`**`(`Scalar *val*`)`

reference `reference::`**`operator/=`**`(`Scalar *val*`)`

> Assignment and compound assignment operators. For a given operator `op`, update the referenced element, *elem*, via *elem* `op` *val*. Return `*this`.

pointer `reference::`**`operator&`**`()`

> Return pointer to the referenced array element.

Finally, zfp proxy references serve as a building block for implementing proxy *pointers* and *iterators*.

10.4 Pointers

template<typename **`Scalar`**>
`class` `array1::`**`pointer`**

template<typename **`Scalar`**>
`class` `array2::`**`pointer`**

template<typename **`Scalar`**>
`class` `array3::`**`pointer`**

Similar to *references*, zfp supports proxies for pointers to individual array elements. From the user's perspective, such pointers behave much like regular pointers to uncompressed data, e.g. instead of

```
float a[ny][nx];      // uncompressed 2D array of floats
float* p = &a[0][0]; // point to first array element
p[nx] = 1;            // set a[1][0] = 1
*++p = 2;             // set a[0][1] = 2
```

one would write

```
zfp::array2<float> a(nx, ny, rate);        // compressed 2D array of floats
zfp::array2<float>::pointer p = &a(0, 0); // point to first array element
p[nx] = 1;                                 // set a(0, 1) = 1
*++p = 2;                                  // set a(1, 0) = 2
```

However, even though zfp's proxy pointers point to individual scalars, they are associated with the array that those scalars are stored in, including the array's dimensionality. Pointers into arrays of different dimensionality have incompatible type. Moreover, pointers to elements in different arrays are incompatible. For example, one cannot take the difference between pointers into two different arrays.

Unlike zfp's proxy references, its proxy pointers support traversing arrays using conventional pointer arithmetic. In particular, unlike the *iterators* below, zfp's pointers are oblivious to the fact that the compressed arrays are partitioned into blocks, and the pointers traverse arrays element by element as though the arrays were flattened in standard C row-major order. That is, if `p` points to the first element of a 3D array `a(nx, ny, nz)`, then `a(i, j, k) == p[i + nx * (j + ny * k)]`. In other words, pointer indexing follows the same order as flat array indexing (see `array::operator[]()`).

A pointer remains valid during the lifetime of the scalar that it points to. Like conventional pointers, proxy pointers can be passed to other functions and manipulated there, for instance by passing the pointer by reference via `pointer&`.

The following operators are defined for proxy pointers. Below *p* refers to the pointer being acted upon.

pointer `pointer::`**`operator=`**`(`**`const`** pointer &*q*`)`

Assignment operator. Assigns *q* to *p*.

reference `pointer::`**`operator*`**`()` **`const`**

Dereference operator. Return proxy reference to the value pointed to by *p*.

reference `pointer::`**`operator[]`**`(`ptrdiff_t *d*`)` **`const`**

Index operator. Return reference to the value stored at `p[d]`.

pointer &`pointer::`**`operator++`**`()`

pointer &`pointer::`**`operator--`**`()`

Pre increment (decrement) pointer, e.g. `++p`. Return reference to the incremented (decremented) pointer.

pointer `pointer::`**`operator++`**`(`int`)`

pointer `pointer::`**`operator--`**`(`int`)`

Post increment (decrement) pointer, e.g. `p++`. Return a copy of the pointer before it was incremented (decremented).

pointer `pointer::`**`operator+=`**`(`ptrdiff_t *d*`)`

pointer `pointer::`**`operator-=`**`(`ptrdiff_t *d*`)`

Increment (decrement) pointer by *d*. Return a copy of the incremented (decremented) pointer.

pointer `pointer::`**`operator+`**`(`ptrdiff_t *d*`)` **`const`**

pointer `pointer::`**`operator-`**`(`ptrdiff_t *d*`)` **`const`**

Return a copy of the pointer incremented (decremented) by *d*.

ptrdiff_t `pointer::`**`operator-`**`(`**`const`** pointer &*q*`)` **`const`**

Return difference *p* - *q*. Defined only for pointers within the same array.

bool `pointer::`**`operator==`**`(`**`const`** pointer &*q*`)` **`const`**

bool `pointer::`**`operator!=`**`(`**`const`** pointer &*q*`)` **`const`**

Pointer comparison. Return true (false) if *p* and *q* do (do not) point to the same array element.

10.5 Iterators

template<typename **`Scalar`**>
`class` `array1::`**`iterator`**

template<typename **`Scalar`**>

class `array2::`**iterator**

template<typename **Scalar**>
class `array3::`**iterator**

Iterators provide a mechanism for sequentially traversing a possibly multi-dimensional array without having to track array indices or bounds. They are also the preferred mechanism, compared to nested index loops, for initializing arrays, because they are guaranteed to visit the array one block at a time. This allows all elements of a block to be initialized together and ensures that the block is not compressed to memory before it has been fully initialized, which might otherwise result in poor compression and, consequently, larger errors than when the entire block is initialized as a whole. Note that the iterator traversal order differs in this respect from traversal by *pointers*.

The order of blocks visited is row-major (as in C), and the elements within a block are also visited in row-major order, i.e. first by x, then by y, and finally by z. All 4^d values in a block are visited before moving on to the next block.

The iterators provided by zfp are sequential forward iterators, except for 1D array iterators, which are random access iterators. The reason why higher dimensional iterators do not support random access is that this would require very complicated index computations, especially for arrays with partial blocks. zfp iterators are STL compliant and can be used in STL algorithms that support forward and random access iterators.

10.5.1 All Iterators

Per STL mandate, the iterators define several types:

type `iterator::`**value_type**
The scalar type associated with the array that the iterator points into.

type `iterator::`**difference_type**
Difference between two iterators in number of array elements.

type `iterator::`**reference**

type `iterator::`**pointer**
The reference and pointer type associated with the iterator's parent array class.

type `iterator::`**iterator_category**
Type of iterator: `std::random_access_iterator_tag` for 1D arrays; `std::forward_iterator_tag` for all other arrays.

The following operations are defined on iterators:

iterator `iterator::`**operator=**(**const** iterator &*it*)
Assignment (copy) operator. Make the iterator point to the same element as *it*.

reference `iterator::`**operator***() **const**
Dereference operator. Return reference to the value pointed to by the iterator.

iterator &`iterator::`**operator++**()
Pre increment. Return a reference to the incremented iterator.

iterator `iterator::`**operator++**(int)
Post increment. Return the value of the iterator before being incremented.

bool `iterator::`**operator==**(**const** iterator &*it*) **const**

bool `iterator::`**operator!=**(**const** iterator &*it*) **const**
Return true (false) if the two iterators do (do not) point to the same element.

uint `iterator::`**i**() **const**

uint `iterator::`**j**() **const**

uint `iterator::`**`k`**`()` **`const`**

Return array index of element pointed to by the iterator. *`iterator::i()`* is defined for all arrays. *`iterator::j()`* is defined only for 2D and 3D arrays. *`iterator::k()`* is defined only for 3D arrays.

10.5.2 1D Array Iterators

The following operators are defined **only for 1D arrays**:

reference `iterator::`**`operator[]`**`(`difference_type *d*`)` **`const`**

Random access index operator.

iterator &`iterator::`**`operator--`**`()`

Pre decrement. Return a reference to the decremented iterator.

iterator `iterator::`**`operator--`**`(`int`)`

Post decrement. Return the value of the iterator before being decremented.

iterator `iterator::`**`operator+=`**`(`difference_type *d*`)`

iterator `iterator::`**`operator-=`**`(`difference_type *d*`)`

Increment (decrement) iterator *d* times. Return value of incremented (decremented) iterator.

iterator `iterator::`**`operator+`**`(`difference_type *d*`)` **`const`**

iterator `iterator::`**`operator-`**`(`difference_type *d*`)` **`const`**

Return a new iterator that has been incremented (decremented) by *d*.

difference_type `iterator::`**`operator-`**`(`**`const`** iterator &*it*`)` **`const`**

Return difference between this iterator and *it* in number of elements. The iterators must refer to elements in the same array.

bool `iterator::`**`operator<=`**`(`**`const`** iterator &*it*`)` **`const`**

bool `iterator::`**`operator>=`**`(`**`const`** iterator &*it*`)` **`const`**

bool `iterator::`**`operator<`**`(`**`const`** iterator &*it*`)` **`const`**

bool `iterator::`**`operator>`**`(`**`const`** iterator &*it*`)` **`const`**

Return true if the two iterators satisfy the given relationship. For two iterators, *p* and *q*, within the same array, *p* < *q* if and only if *q* can be reached by incrementing *p* one or more times.

CHAPTER 11

Tutorial

This tutorial provides examples that illustrate how to use the zfp library and compressed arrays, and includes code snippets that show the order of declarations and function calls needed to use the compressor.

This tutorial is divided into three parts: the high-level libzfp *library*; the low-level *compression codecs*; and the *compressed array classes* (in that order). Users interested only in the compressed arrays, which do not directly expose anything related to compression other than compression *rate control*, may safely skip the next two sections.

All code examples below are for 3D arrays of doubles, but it should be clear how to modify the function calls for single precision and for 1D or 2D arrays.

11.1 High-Level C Interface

Users concerned only with storing their floating-point data compressed may use zfp as a black box that maps a possibly non-contiguous floating-point array to a compressed bit stream. The intent of libzfp is to provide both a high- and low-level interface to the compressor that can be called from both C and C++ (and possibly other languages). libzfp supports strided access, e.g. for compressing vector fields one scalar at a time, or for compressing arrays of structs.

Consider compressing the 3D C/C++ array

```
// define an uncompressed array
double a[nz][ny][nx];
```

where *nx*, *ny*, and *nz* can be any positive dimensions. To invoke the libzfp compressor, the dimensions and type must first be specified in a `zfp_field` parameter object that encapsulates the type, size, and memory layout of the array:

```
// allocate metadata for the 3D array a[nz][ny][nx]
uint dims = 3;
zfp_type type = zfp_type_double;
zfp_field* field = zfp_field_3d(&a[0][0][0], type, nx, ny, nz);
```

For single-precision data, use `zfp_type_float`. As of version 0.5.1, the the high-level API also supports integer arrays (`zfp_type_int32` and `zfp_type_int64`). See FAQs *#8* and *#9* regarding integer compression.

Functions similar to `zfp_field_3d()` exist for declaring 1D and 2D arrays. If the dimensionality of the array is unknown at this point, then a generic `zfp_field_alloc()` call can be made to just allocate a `zfp_field` struct, which can be filled in later using the *set* functions. If the array is non-contiguous, then `zfp_field_set_stride_3d()` should be called.

The `zfp_field` parameter object holds information about the uncompressed array. To specify the compressed array, a `zfp_stream` object must be allocated:

```
// allocate metadata for a compressed stream
zfp_stream* zfp = zfp_stream_open(NULL);
```

We may now specify the rate, precision, or accuracy (see *Compression Modes* for more details on the meaning of these parameters):

```
// set compression mode and parameters
zfp_stream_set_rate(zfp, rate, type, dims, 0);
zfp_stream_set_precision(zfp, precision);
zfp_stream_set_accuracy(zfp, tolerance);
```

Note that only one of these three functions should be called. The return value from these functions gives the actual rate, precision, or tolerance, and may differ slightly from the argument passed due to constraints imposed by the compressor, e.g. each block must be stored using a whole number of bits at least as large as the number of bits in the floating-point exponent; the precision cannot exceed the number of bits in a floating-point value (i.e. 32 for single and 64 for double precision); and the tolerance must be a (possibly negative) power of two.

The compression parameters have now been specified, but before compression can occur a buffer large enough to hold the compressed bit stream must be allocated. Another utility function exists for estimating how many bytes are needed:

```
// allocate buffer for compressed data
size_t bufsize = zfp_stream_maximum_size(zfp, field);
uchar* buffer = new uchar[bufsize];
```

Note that `zfp_stream_maximum_size()` returns the smallest buffer size necessary to safely compress the data–the *actual* compressed size may be smaller. If the members of `zfp` and `field` are for whatever reason not initialized correctly, then `zfp_stream_maximum_size()` returns 0.

Before compression can commence, we must associate the allocated buffer with a bit stream used by the compressor to read and write bits:

```
// associate bit stream with allocated buffer
bitstream* stream = stream_open(buffer, bufsize);
zfp_stream_set_bit_stream(zfp, stream);
```

Finally, the array is compressed as follows:

```
// compress entire array
size_t size = zfp_compress(zfp, field);
```

The return value is the actual number of bytes of compressed storage, and as already mentioned, $size \leq bufsize$. If $size = 0$, then the compressor failed. Since zfp 0.5.0, the compressor does not rewind the bit stream before compressing, which allows multiple fields to be compressed one after the other. The return value from `zfp_compress()` is always the total number of bytes of compressed storage so far relative to the memory location pointed to by *buffer*.

To decompress the data, the field and compression parameters must be initialized with the same values as used for compression, either via the same sequence of function calls as above, or by recording these fields and setting them directly. Metadata such as array dimensions and compression parameters are by default not stored in the compressed stream. It is up to the caller to store this information, either separate from the compressed data, or via the `zfp_write_header()` and `zfp_read_header()` calls, which must precede the corresponding `zfp_compress()` and `zfp_decompress()` calls, respectively. These calls allow the user to specify what information to store in the header, including a 'magic' format identifier, the field type and dimensions, and the compression parameters (see the *ZFP_HEADER* macros).

In addition to this initialization, the bit stream has to be rewound to the beginning (before reading the header and decompressing the data):

```
// rewind compressed stream and decompress array
zfp_stream_rewind(zfp);
int success = zfp_decompress(zfp, field);
```

The return value is zero if the decompressor failed.

11.1.1 Simple Example

Tying it all together, the code example below (see also the *simple* program) shows how to compress a 3D array `double array[nz][ny][nx]`:

```
// input: (void* array, int nx, int ny, int nz, double tolerance)

// initialize metadata for the 3D array a[nz][ny][nx]
zfp_type type = zfp_type_double;                          // array scalar type
zfp_field* field = zfp_field_3d(array, type, nx, ny, nz); // array metadata

// initialize metadata for a compressed stream
zfp_stream* zfp = zfp_stream_open(NULL);                  // compressed stream and␣
↪parameters
zfp_stream_set_accuracy(zfp, tolerance);                  // set tolerance for fixed-
↪accuracy mode
//  zfp_stream_set_precision(zfp, precision);             // alternative: fixed-
↪precision mode
//  zfp_stream_set_rate(zfp, rate, type, 3, 0);           // alternative: fixed-rate␣
↪mode

// allocate buffer for compressed data
size_t bufsize = zfp_stream_maximum_size(zfp, field);     // capacity of compressed␣
↪buffer (conservative)
void* buffer = malloc(bufsize);                           // storage for compressed␣
↪stream

// associate bit stream with allocated buffer
bitstream* stream = stream_open(buffer, bufsize);         // bit stream to compress to
zfp_stream_set_bit_stream(zfp, stream);                   // associate with␣
↪compressed stream
zfp_stream_rewind(zfp);                                   // rewind stream to␣
↪beginning

// compress array
size_t zfpsize = zfp_compress(zfp, field);                // return value is byte␣
↪size of compressed stream
```

11.2 Low-Level C Interface

For applications that wish to compress or decompress portions of an array on demand, a low-level interface is available. Since this API is useful primarily for supporting random access, the user also needs to manipulate the *bit stream*, e.g. to position the bit pointer to where data is to be read or written. Please be advised that the bit stream functions have been optimized for speed, and do not check for buffer overruns or other types of programmer error.

Like the high-level API, the low-level API also makes use of the `zfp_stream` parameter object (see section above) to specify compression parameters and storage, but does not encapsulate array metadata in a `zfp_field` object. Functions exist for encoding and decoding complete or partial blocks, with or without strided access. In non-strided mode, the uncompressed block to be encoded or decoded is assumed to be stored contiguously. For example,

```
// compress a single contiguous block
double block[4 * 4 * 4] = { /* some set of values */ };
uint bits = zfp_encode_block_double_3(zfp, block);
```

The return value is the number of bits of compressed storage for the block. For fixed-rate streams, if random access is desired, then the stream should also be flushed after each block is encoded:

```
// flush any buffered bits
zfp_stream_flush(zfp);
```

This flushing should be done only after the last block has been compressed in fixed-precision and fixed-accuracy mode, or when random access is not needed in fixed-rate mode.

The block above could also have been compressed as follows using strides:

```
// compress a single contiguous block using strides
double block[4][4][4] = { /* some set of values */ };
int sx = &block[0][0][1] - &block[0][0][0]; // x stride =  1
int sy = &block[0][1][0] - &block[0][0][0]; // y stride =  4
int sz = &block[1][0][0] - &block[0][0][0]; // z stride = 16
uint bits = zfp_encode_block_strided_double_3(zfp, block, sx, sy, sz);
```

The strides are measured in number of scalars, not in bytes.

For partial blocks, e.g. near the boundaries of arrays whose dimensions are not multiples of four, there are corresponding functions that accept parameters nx, ny, and nz to specify the actual block dimensions, with $1 \leq nx, ny, nz \leq 4$. Corresponding functions exist for decompression. Such partial blocks typically do not compress as well as full blocks and should be avoided if possible.

To position a bit stream for reading (decompression), use

```
// position the stream at given bit offset for reading
stream_rseek(stream, offset);
```

where the offset is measured in number of bits from the beginning of the stream. For writing (compression), a corresponding call exists:

```
// position the stream at given bit offset for writing
stream_wseek(stream, offset);
```

Note that it is possible to decompress fewer bits than are stored with a compressed block to quickly obtain an approximation. This is done by setting `zfp->maxbits` to fewer bits than used during compression, e.g. to decompress only the first 256 bits of each block:

```
// modify decompression parameters to decode 256 bits per block
uint maxbits;
uint maxprec;
int minexp;
zfp_stream_params(zfp, NULL, &maxbits, &maxprec, &minexp);
assert(maxbits >= 256);
zfp_stream_set_params(zfp, 256, 256, maxprec, minexp);
```

This feature may be combined with progressive decompression, as discussed further in FAQ *#13*.

11.3 Compressed C++ Arrays

The zfp API has been designed to facilitate integration with existing applications. After initial array declaration, a zfp array can often be used in place of a regular C/C++ array or STL vector, e.g. using flat indexing via `a[index]` or using multidimensional indexing via `a(i)`, `a(i, j)`, or `a(i, j, k)`. There are, however, some important differences. For instance, applications that rely on addresses or references to array elements may have to be modified to use special proxy classes that implement pointers and references; see *Limitations*.

zfp does not support special floating-point values like infinities and NaNs, although denormalized numbers are handled correctly. Similarly, because the compressor assumes that the array values vary smoothly, using finite but large values like `HUGE_VAL` in place of infinities is not advised, as this will introduce large errors in smaller values within the same block. Future extensions will provide support for a bit mask to mark the presence of non-values.

The zfp C++ classes are implemented entirely as header files and make extensive use of C++ templates to reduce code redundancy. Most classes are wrapped in the `zfp` namespace.

Currently there are six array classes for 1D, 2D, and 3D arrays, each of which can represent single- or double-precision values. Although these arrays store values in a form different from conventional single- and double-precision floating point, the user interacts with the arrays via floats and doubles.

The description below is for 3D arrays of doubles–the necessary changes for other array types should be obvious. To declare and zero initialize an array, use

```
// declare nx * ny * nz array of compressed doubles
zfp::array3<double> a(nx, ny, nz, rate);
```

This declaration is conceptually equivalent to

```
double a[nz][ny][nx] = {};
```

or using STL vectors

```
std::vector<double> a(nx * ny * nz, 0.0);
```

but with the user specifying the amount of storage used via the *rate* parameter. (A predefined type `array3d` also exists, while the suffix 'f' is used for floats.) Note that the array dimensions can be arbitrary, and need not be multiples of four (see above for a discussion of incomplete blocks). The *rate* argument specifies how many bits per value (amortized) to store in the compressed representation. By default the block size is restricted to a multiple of 64 bits, and therefore the rate argument can be specified in increments of 64 / 4^d bits in d dimensions, i.e.

```
1D arrays: 16-bit granularity
2D arrays: 4-bit granularity
3D arrays: 1-bit granularity
```

For finer granularity, the *`BIT_STREAM_WORD_TYPE`* macro needs to be set to a type narrower than 64 bits during compilation of libzfp, e.g. if set to `uint8` the rate granularity becomes 8 / 4^d bits in d dimensions, or

```
1D arrays: 2-bit granularity
2D arrays: 1/2-bit granularity
3D arrays: 1/8-bit granularity
```

Note that finer granularity usually implies slightly lower performance. Also note that because the arrays are stored compressed, their effective precision is likely to be higher than the user-specified rate.

The array can also optionally be initialized from an existing contiguous floating-point array stored at *pointer* with an x stride of 1, y stride of nx, and z stride of $nx \times ny$:

```
// declare and initialize 3D array of doubles
zfp::array3d a(nx, ny, nz, precision, pointer, cache_size);
```

The optional *cache_size* argument specifies the minimum number of bytes to allocate for the cache of uncompressed blocks (see *Caching* below for more details).

If not already initialized, a function `array::set()` can be used to copy uncompressed data to the compressed array:

```
const double* pointer; // pointer to uncompressed, initialized data
a.set(pointer); // initialize compressed array with floating-point data
```

Similarly, an `array::get()` function exists for retrieving uncompressed data:

```
double* pointer; // pointer to where to write uncompressed data
a.get(pointer); // decompress and store the array at pointer
```

The compressed representation of an array can also be queried or initialized directly without having to convert to/from its floating-point representation:

```
size_t bytes = compressed_size(); // number of bytes of compressed storage
uchar* compressed_data(); // pointer to compressed data
```

The array can through this pointer be initialized from offline compressed storage, but only after its dimensions and rate have been specified (see above). For this to work properly, the cache must first be emptied via a `array::clear_cache()` call (see below).

Through operator overloading, the array can be accessed in one of two ways. For read accesses, use

```
double value = a[index]; // fetch value with given flat array index
double value = a(i, j, k); // fetch value with 3D index (i, j, k)
```

These access the same value if and only if `index = i + nx * (j + ny * k)`. Note that $0 \leq i < nx$, $0 \leq j < ny$, and $0 \leq k < nz$, and i varies faster than j, which varies faster than k.

Array values may be written and updated using the usual set of C++ assignment and compound assignment operators. For example:

```
a[index] = value; // set value at flat array index
a(i, j, k) += value; // increment value with 3D index (i, j, k)
```

Whereas one might expect these operators to return a (non-const) reference to an array element, this would allow seating a reference to a value that currently is cached but is transient, which could be unsafe. Moreover, this would preclude detecting when an array element is modified. Therefore, the return type of both operators `[]` and `()` is a proxy reference class, similar to `std::vector<bool>::reference` from the STL library. Because read accesses to a mutable object cannot call the const-qualified accessor, a proxy reference may be returned even for read calls, e.g. in

```
a[i - 1] = a[i];
```

the array `a` clearly must be mutable to allow assignment to `a[i - 1]`, and therefore the read access `a[i]` returns type `array::reference`. The value associated with the read access is obtained via an implicit conversion.

Array dimensions *nx*, *ny*, and *nz* can be queried using these functions:

```
size_t size(); // total number of elements nx * ny * nz
uint size_x(); // nx
uint size_y(); // ny
uint size_z(); // nz
```

The array dimensions can also be changed dynamically, e.g. if not known at time of construction, using

```
void resize(uint nx, uint ny, uint nz, bool clear = true);
```

When *clear* = true, the array is explicitly zeroed. In either case, all previous contents of the array are lost. If *nx* = *ny* = *nz* = 0, all storage is freed.

Finally, the rate supported by the array may be queried via

```
double rate(); // number of compressed bits per value
```

and changed using

```
void set_rate(rate); // change rate
```

This also destroys prior contents.

As of zfp 0.5.2, iterators and proxy objects for pointers and references are supported. Note that the decompressed value of an array element exists only intermittently, when the decompressed value is cached. It would not be safe to return a `double&` reference or `double*` pointer to the cached but transient value since it may be evicted from the cache at any point, thus invalidating the reference or pointer. Instead, zfp provides proxy objects for references and pointers that guarantee persistent access by referencing elements by array object and index. These classes perform decompression on demand, much like how Boolean vector references are implemented in the STL.

Iterators for 1D arrays support random access, while 2D and 3D array iterators are merely forward (sequential) iterators. All iterators ensure that array values are visited one block at a time, and are the preferred way of looping over array elements. Such block-by-block access is especially useful when performing write accesses since then complete blocks are updated one at a time, thus reducing the likelihood of a partially updated block being evicted from the cache and compressed, perhaps with some values in the block being uninitialized. Here is an example of initializing a 3D array:

```
for (zfp::array3d::iterator it = a.begin(); it != a.end(); it++) {
  int i = it.i();
  int j = it.j();
  int k = it.k();
  a(i, j, k) = some_function(i, j, k);
}
```

Pointers to array elements are available via a special pointer class. Such pointers may be a useful way of passing (flattened) zfp arrays to functions that expect uncompressed arrays, e.g. by using the pointer type as template argument. For example:

```
template <typename double_ptr>
void sum(double_ptr p, int count)
{
```

(continues on next page)

(continued from previous page)

```
  double s = 0;
  for (int i = 0; i < count; i++)
    s += p[i];
  return s;
}
```

Then the following are equivalent:

```
// sum of STL vector elements (double_ptr == double*)
std::vector<double> vec(nx * ny * nz, 0.0);
double vecsum = sum(&vec[0], nx * ny * nz);

// sum of zfp array elements (double_ptr == zfp::array3d::pointer)
zfp::array3<double> array(nx, ny, nz, rate);
double zfpsum = sum(&array[0], nx * ny * nz);
```

As another example,

```
for (zfp::array1d::pointer p = &a[0]; p - &a[0] < a.size(); p++)
  *p = 0.0;
```

initializes a 1D array to all-zeros. Pointers visit arrays in standard row-major order, i.e.

```
&a(i, j, k) == &a[0] + i + nx * (j + ny * k)
            == &a[i + nx * (j + ny * k)]
```

where `&a(i, j, k)` and `&a[0]` are both of type `array3d::pointer`. Thus, iterators and pointers do not visit arrays in the same order, except for the special case of 1D arrays. Unlike iterators, pointers support random access for arrays of all dimensions and behave very much like `float*` and `double*` built-in pointers.

Proxy objects for array element references have been supported since the first release of zfp, and may for instance be used in place of `double&`. Iterators and pointers are implemented in terms of references.

The following table shows the equivalent zfp type to standard types when working with 1D arrays:

```
double&                        zfp::array1d::reference
double*                        zfp::array1d::pointer
std::vector<double>::iterator  zfp::array1d::iterator
```

11.3.1 Caching

As mentioned above, the array class maintains a software write-back cache of at least one uncompressed block. When a block in this cache is evicted (e.g. due to a conflict), it is compressed back to permanent storage only if it was modified while stored in the cache.

The size cache to use is specified by the user, and is an important parameter that needs careful consideration in order to balance the extra memory usage, performance, and quality (recall that data loss is incurred only when a block is evicted from the cache and compressed). Although the best choice varies from one application to another, we suggest allocating at least two layers of blocks ($2 \times (nx / 4) \times (ny / 4)$ blocks) for applications that stream through the array and perform stencil computations such as gathering data from neighboring elements. This allows limiting the cache misses to compulsory ones. If the *cache_size* parameter is set to zero bytes, then this default of two layers is used.

The cache size can be set during construction, or can be set at a later time via

```
void set_cache_size(bytes); // change cache size
```

Note that if *bytes* = 0, then the array dimensions must have already been specified for the default size to be computed correctly. When the cache is resized, it is first flushed if not already empty. The cache can also be flushed explicitly if desired by calling

```
void flush_cache(); // empty cache by first compressing any modified blocks
```

To empty the cache without compressing any cached data, call

```
void clear_cache(); // empty cache without compression
```

To query the byte size of the cache, use

```
size_t cache_size(); // actual cache size in bytes
```

CHAPTER 12

File Compressor

The **`zfp`** executable in the `bin` directory is primarily intended for evaluating the rate-distortion (compression ratio and quality) provided by the compressor, but since version 0.5.0 also allows reading and writing compressed data sets. **`zfp`** takes as input a raw, binary array of floats, doubles, or integers in native byte order and optionally outputs a compressed or reconstructed array obtained after lossy compression followed by decompression. Various statistics on compression ratio and error are also displayed.

The uncompressed input and output files should be a flattened, contiguous sequence of scalars without any header information, generated for instance by

```
double* data = new double[nx * ny * nz];
// populate data
FILE* file = fopen("data.bin", "wb");
fwrite(data, sizeof(*data), nx * ny * nz, file);
fclose(file);
```

`zfp` requires a set of command-line options, the most important being the `-i` option that specifies that the input is uncompressed. When present, `-i` tells **`zfp`** to read an uncompressed input file and compress it to memory. If desired, the compressed stream can be written to an output file using `-z`. When `-i` is absent, on the other hand, `-z` names the compressed input (not output) file, which is then decompressed. In either case, `-o` can be used to output the reconstructed array resulting from lossy compression and decompression.

So, to compress a file, use `-i file.in -z file.zfp`. To later decompress the file, use `-z file.zfp -o file.out`. A single dash "-" can be used in place of a file name to denote standard input or output.

When reading uncompressed input, the floating-point precision (single or double) must be specified using either `-f` (float) or `-d` (double). In addition, the array dimensions must be specified using `-1` (for 1D arrays), `-2` (for 2D arrays), or `-3` (for 3D arrays). For multidimensional arrays, x varies faster than y, which in turn varies faster than z. That is, a 3D input file corresponding to a flattened C array `a[nz][ny][nx]` is specified as `-3 nx ny nz`.

Note that `-2 nx ny` is not equivalent to `-3 nx ny 1`, even though the same number of values are compressed. One invokes the 2D codec, while the other uses the 3D codec, which in this example has to pad the input to an $nx \times ny \times 4$ array since arrays are partitioned into blocks of dimensions 4^d. Such padding usually negatively impacts compression.

Moreover, `-2 nx ny` is not equivalent to `-2 ny nx`, i.e., with the dimensions transposed. It is crucial for accuracy and compression ratio that the array dimensions are listed in the order expected by **zfp** so that the array layout is correctly interpreted. See this *discussion* for more details.

Using *-h*, the array dimensions and type are stored in a header of the compressed stream so that they do not have to be specified on the command line during decompression. The header also stores compression parameters, which are described below. The compressor and decompressor must agree on whether headers are used, and it is up to the user to enforce this.

zfp accepts several options for specifying how the data is to be compressed. The most general of these, the *-c* option, takes four constraint parameters that together can be used to achieve various effects. These constraints are:

```
minbits: the minimum number of bits used to represent a block
maxbits: the maximum number of bits used to represent a block
maxprec: the maximum number of bit planes encoded
minexp:  the smallest bit plane number encoded
```

These parameters are discussed in detail in the section on *compression modes*. Options *-r*, *-p*, and *-a* provide a simpler interface to setting all of the above parameters by invoking *fixed-rate* (*-r*), *-precision* (*-p*), and *-accuracy* (*-a*).

12.1 Usage

Below is a description of each command-line option accepted by **zfp**.

12.1.1 General options

-h

Read/write array and compression parameters from/to compressed header.

-q

Quiet mode; suppress diagnostic output.

-s

Evaluate and print the following error statistics:

- rmse: The root mean square error.
- nrmse: The root mean square error normalized to the range.
- maxe: The maximum absolute pointwise error.
- psnr: The peak signal to noise ratio in decibels.

12.1.2 Input and output

-i <path>

Name of uncompressed binary input file. Use "-" for standard input.

-o <path>

Name of decompressed binary output file. Use "-" for standard output. May be used with either *-i*, *-z*, or both.

-z <path>

Name of compressed input (without *-i*) or output file (with *-i*). Use "-" for standard input or output.

When *-i* is specified, data is read from the corresponding uncompressed file, compressed, and written to the compressed file specified by *-z* (when present). Without *-i*, compressed data is read from the file specified by *-z* and decompressed. In either case, the reconstructed data can be written to the file specified by *-o*.

12.1.3 Array type and dimensions

-f
Single precision (float type). Shorthand for `-t f32`.

-d
Double precision (double type). Shorthand for `-t f64`.

-t `<type>`
Specify scalar type as one of i32, i64, f32, f64 for 32- or 64-bit integer or floating scalar type.

-1 `<nx>`
Dimensions of 1D C array `a[nx]`.

-2 `<nx> <ny>`
Dimensions of 2D C array `a[ny][nx]`.

-3 `<nx> <ny> <nz>`
Dimensions of 3D C array `a[nz][ny][nx]`.

When *-i* is used, the scalar type and array dimensions must be specified. One of *-f*, *-d*, or *-t* specifies the input scalar type. *-1*, *-2*, or *-3* specifies the array dimensions. The same parameters must be given when decompressing data (without *-i*), unless a header was stored using *-h* during compression.

12.1.4 Compression parameters

-r `<rate>`
Specify fixed rate in terms of number of compressed bits per floating-point value.

-p `<precision>`
Specify fixed precision in terms of number of uncompressed bits per value.

-a `<tolerance>`
Specify fixed accuracy in terms of absolute error tolerance.

-c `<minbits> <maxbits> <maxprec> <minexp>`
Specify expert mode parameters.

When *-i* is used, the compression parameters must be specified. The same parameters must be given when decompressing data (without *-i*), unless a header was stored using *-h* when compressing. See the section on *compression modes* for a discussion of these parameters.

12.1.5 Examples

- `-i file` : read uncompressed file and compress to memory
- `-z file` : read compressed file and decompress to memory
- `-i ifile -z zfile` : read uncompressed ifile, write compressed zfile
- `-z zfile -o ofile` : read compressed zfile, write decompressed ofile
- `-i ifile -o ofile` : read ifile, compress, decompress, write ofile
- `-i file -s` : read uncompressed file, compress to memory, print stats
- `-i - -o - -s` : read stdin, compress, decompress, write stdout, print stats
- `-f -3 100 100 100 -r 16` : 2x fixed-rate compression of 100 × 100 × 100 floats
- `-d -1 1000000 -r 32` : 2x fixed-rate compression of 1,000,000 doubles

- `-d -2 1000 1000 -p 32` : 32-bit precision compression of 1000 × 1000 doubles
- `-d -1 1000000 -a 1e-9` : compression of 1,000,000 doubles with $< 10^{-9}$ max error
- `-d -1 1000000 -c 64 64 0 -1074` : 4x fixed-rate compression of 1,000,000 doubles

CHAPTER 13

Code Examples

The `examples` directory includes five programs that make use of the compressor.

13.1 Simple Compressor

The **`simple`** program is a minimal example that shows how to call the compressor and decompressor on a double-precision 3D array. Without the `-d` option, it will compress the array and write the compressed stream to standard output. With the `-d` option, it will instead read the compressed stream from standard input and decompress the array:

```
simple > compressed.zfp
simple -d < compressed.zfp
```

For a more elaborate use of the compressor, see the *zfp utility*.

13.2 Diffusion Solver

The **`diffusion`** example is a simple forward Euler solver for the heat equation on a 2D regular grid, and is intended to show how to declare and work with zfp's compressed arrays, as well as give an idea of how changing the compression rate and cache size affects the error in the solution and solution time. The usage is:

```
diffusion [-i] [-n nx ny] [-t nt] [-r rate] [-c blocks]
```

where *rate* specifies the exact number of compressed bits to store per double-precision floating-point value (default = 64); *nx* and *ny* specify the grid size (default = 100×100); *nt* specifies the number of time steps to take (the default is to run until time $t = 1$); and *blocks* is the number of uncompressed blocks to cache (default = nx / 2). The `-i` option enables array traversal via iterators instead of indices.

Running diffusion with the following arguments:

```
diffusion -r 8
diffusion -r 12
diffusion -r 20
diffusion -r 64
```

should result in this output:

```
rate=8 sum=0.996442 error=4.813938e-07
rate=12 sum=0.998338 error=1.967777e-07
rate=20 sum=0.998326 error=1.967952e-07
rate=64 sum=0.998326 error=1.967957e-07
```

For speed and quality comparison, the solver solves the same problem using uncompressed double-precision arrays when `-r` is omitted.

13.3 Speed Benchmark

The **speed** program takes two optional parameters:

```
speed [rate] [blocks]
```

It measures the throughput of compression and decompression of 3D double-precision data (in megabytes of uncompressed data per second). By default, a rate of 1 bit/value and two million blocks are processed.

13.4 PGM Image Compression

The **pgm** program illustrates how zfp can be used to compress grayscale images in the pgm format. The usage is:

```
pgm <param> <input.pgm >output.pgm
```

If `param` is positive, it is interpreted as the rate in bits per pixel, which ensures that each block of 4 × 4 pixels is compressed to a fixed number of bits, as in texture compression codecs. If `param` is negative, then fixed-precision mode is used with precision `-param`, which tends to give higher quality for the same rate. This use of zfp is not intended to compete with existing texture and image compression formats, but exists merely to demonstrate how to compress 8-bit integer data with zfp. See FAQs *#20* and *#21* for information on the effects of setting the precision.

13.5 In-place Compression

The **inplace** example shows how one might use zfp to perform in-place compression and decompression when memory is at a premium. Here the floating-point array is overwritten with compressed data, which is later decompressed back in place. This example also shows how to make use of some of the low-level features of zfp, such as its low-level, block-based compression API and bit stream functions that perform seeks on the bit stream. The program takes one optional argument:

```
inplace [tolerance]
```

which specifies the fixed-accuracy absolute tolerance to use during compression. Please see *FAQ #19* for more on the limitations of in-place compression.

13.6 Iterators

The **`iterator`** example illustrates how to use zfp's compressed-array iterators and pointers for traversing arrays. For instance, it gives an example of sorting a 1D compressed array using `std::sort()`. This example takes no command-line options.

CHAPTER 14

Regression Tests

The **`testzfp`** program in the `tests` directory performs regression testing that exercises most of the functionality of `libzfp` and the array classes. The tests assume the default compiler settings, i.e. with none of the macros in `Config` defined. By default, small, pregenerated floating-point arrays are used in the test, since they tend to have the same binary representation across platforms, whereas it can be difficult to computationally generate bit-for-bit identical arrays. To test larger arrays, use the `medium` or `large` options. When large arrays are used, the (de)compression throughput is also measured and reported in number of uncompressed bytes per second.

CHAPTER 15

FAQ

The following is a list of answers to frequently asked questions. For questions not answered here or elsewhere in the documentation, please e-mail Peter Lindstrom.

Questions answered in this FAQ:

1. *Can zfp compress vector fields?*
2. *Should I declare a 2D array as zfp::array1d a(nx * ny, rate)?*
3. *How can I initialize a zfp compressed array from disk?*
4. *Can I use zfp to represent dense linear algebra matrices?*
5. *Can zfp compress logically regular but geometrically irregular data?*
6. *Does zfp handle infinities, NaNs,and subnormal floating-point numbers?*
7. *Can zfp handle data with some missing values?*
8. *Can I use zfp to store integer data?*
9. *Can I compress 32-bit integers using zfp?*
10. *Why does zfp corrupt memory if my allocated buffer is too small?*
11. *Are zfp compressed streams portable across platforms?*
12. *How can I achieve finer rate granularity?*
13. *Can I generate progressive zfp streams?*
14. *How do I initialize the decompressor?*
15. *Must I use the same parameters during compression and decompression?*
16. *Do strides have to match during compression and decompression?*
17. *Why does zfp sometimes not respect my error tolerance?*
18. *Why is the actual rate sometimes not what I requested?*
19. *Can zfp perform compression in place?*

20. *How should I set the precision to bound the relative error?*
21. *Does zfp support lossless compression?*
22. *Why is my actual, measured error so much smaller than the tolerance?*

Q1: *Can zfp compress vector fields?*

I have a 2D vector field

```
double velocity[ny][nx][2];
```

of dimensions *nx* × *ny*. Can I use a 3D zfp array to store this as:

```
array3d velocity(2, nx, ny, rate);
```

A: Although this could be done, zfp assumes that consecutive values are related. The two velocity components (*vx*, *vy*) are almost suredly independent and would not be correlated. This will severely hurt the compression rate or quality. Instead, consider storing *vx* and *vy* as two separate 2D scalar arrays:

```
array2d vx(nx, ny, rate);
array2d vy(nx, ny, rate);
```

or as

```
array2d velocity[2] = {array2d(nx, ny, rate), array2d(nx, ny, rate)};
```

Q2: *Should I declare a 2D array as zfp::array1d a(nx * ny, rate)?*

I have a 2D scalar field of dimensions *nx* × *ny* that I allocate as

```
double* a = new double[nx * ny];
```

and index as

```
a[x + nx * y]
```

Should I use a corresponding zfp array

```
array1d a(nx * ny, rate);
```

to store my data in compressed form?

A: Although this is certainly possible, if the scalar field exhibits coherence in both spatial dimensions, then far better results can be achieved by using a 2D array:

```
array2d a(nx, ny, rate);
```

Although both compressed arrays can be indexed as above, the 2D array can exploit smoothness in both dimensions and improve the quality dramatically for the same rate.

Since zfp 0.5.2, proxy pointers are also available that act much like the flat `double*`.

Q3: *How can I initialize a zfp compressed array from disk?*

I have a large, uncompressed, 3D data set:

```
double a[nz][ny][nx];
```

stored on disk that I would like to read into a compressed array. This data set will not fit in memory uncompressed. What is the best way of doing this?

A: Using a zfp array:

```
array3d a(nx, ny, nz, rate);
```

the most straightforward (but perhaps not best) way is to read one floating-point value at a time and copy it into the array:

```
for (uint z = 0; z < nz; z++)
  for (uint y = 0; y < ny; y++)
    for (uint x = 0; x < nx; x++) {
      double f;
      if (fread(&f, sizeof(f), 1, file) == 1)
        a(x, y, z) = f;
      else {
        // handle I/O error
      }
    }
```

Note, however, that if the array cache is not large enough, then this may compress blocks before they have been completely filled. Therefore it is recommended that the cache holds at least one complete layer of blocks, i.e. (*nx* / 4) × (*ny* / 4) blocks in the example above.

To avoid inadvertent evictions of partially initialized blocks, it is better to buffer four layers of *nx* × *ny* values each at a time, when practical, and to completely initialize one block after another, which is facilitated using zfp's iterators:

```
double* buffer = new double[nx * ny * 4];
int zmin = -4;
for (zfp::array3d::iterator it = a.begin(); it != a.end(); it++) {
  int x = it.i();
  int y = it.j();
  int z = it.k();
  if (z > zmin + 3) {
    // read another layer of blocks
    if (fread(buffer, sizeof(*buffer), nx * ny * 4, file) != nx * ny * 4) {
      // handle I/O error
    }
    zmin += 4;
  }
  a(x, y, z) = buffer[x + nx * (y + ny * (z - zmin))];
}
```

Iterators have been available since zfp 0.5.2.

Q4: *Can I use zfp to represent dense linear algebra matrices?*

A: Yes, but your mileage may vary. Dense matrices, unlike smooth scalar fields, rarely exhibit correlation between adjacent rows and columns. Thus, the quality or compression ratio may suffer.

Q5: *Can zfp compress logically regular but geometrically irregular data?*

My data is logically structured but irregularly sampled, e.g. it is rectilinear, curvilinear, or Lagrangian, or uses an irregular spacing of quadrature points. Can I still use zfp to compress it?

A: Yes, as long as the data is (or can be) represented as a logical multidimensional array, though your mileage may vary. zfp has been designed for uniformly sampled data, and compression will in general suffer the more irregular the sampling is.

Q6: *Does zfp handle infinities, NaNs,and subnormal floating-point numbers?*

A: Only finite, valid floating-point values are currently supported. If a block contains a NaN or an infinity, undefined behavior is invoked due to the C math function `frexp()` being undefined for non-numbers. Subnormal numbers are, however, handled correctly.

Q7: *Can zfp handle data with some missing values?*

My data has some missing values that are flagged by very large numbers, e.g. 1e30. Is that OK?

A: Although all finite numbers are "correctly" handled, such large sentinel values are likely to pollute nearby values, because all values within a block are expressed with respect to a common largest exponent. The presence of very large values may result in complete loss of precision of nearby, valid numbers. Currently no solution to this problem is available, but future versions of zfp will likely support a bit mask to tag values that should be excluded from compression.

Q8: *Can I use zfp to store integer data?*

Can I use zfp to store integer data such as 8-bit quantized images or 16-bit digital elevation models?

A: Yes (as of version 0.4.0), but the data has to be promoted to 32-bit signed integers first. This should be done one block at a time using an appropriate `zfp_promote_*_to_int32()` function call (see `zfp.h`). Future versions of zfp may provide a high-level interface that automatically performs promotion and demotion.

Note that the promotion functions shift the low-precision integers into the most significant bits of 31-bit (not 32-bit) integers and also convert unsigned to signed integers. Do use these functions rather than simply casting 8-bit integers to 32 bits to avoid wasting compressed bits to encode leading zeros. Moreover, in fixed-precision mode, set the precision relative to the precision of the (unpromoted) source data.

As of version 0.5.1, integer data is supported both by the low-level API and high-level calls *`zfp_compress()`* and *`zfp_decompress()`*.

Q9: *Can I compress 32-bit integers using zfp?*

I have some 32-bit integer data. Can I compress it using zfp's 32-bit integer support?

A: Maybe. zfp compression of 32-bit and 64-bit integers requires that each integer f have magnitude $|f| < 2^{30}$ and $|f| < 2^{62}$, respectively. To handle signed integers that span the entire range $-2^{31} \leq x < 2^{31}$, or unsigned integers $0 \leq x < 2^{32}$, the data has to be promoted to 64 bits first.

As with floating-point data, the integers should ideally represent a quantized continuous function rather than, say, categorical data or set of indices. Depending on compression settings and data range, the integers may or may not be losslessly compressed. If fixed-precision mode is used, the integers may be stored at less precision than requested. See *Q21* for more details on precision and lossless compression.

Q10: *Why does zfp corrupt memory if my allocated buffer is too small?*

Why does zfp corrupt memory rather than return an error code if not enough memory is allocated for the compressed data?

A: This is for performance reasons. zfp was primarily designed for fast random access to fixed-rate compressed arrays, where checking for buffer overruns is unnecessary. Adding a test for every compressed byte output would significantly compromise performance.

One way around this problem (when not in fixed-rate mode) is to use the `maxbits` parameter in conjunction with the maximum precision or maximum absolute error parameters to limit the size of compressed blocks. Finally, the function `zfp_stream_maximum_size()` returns a conservative buffer size that is guaranteed to be large enough to hold the compressed data and the optional header.

Q11: *Are zfp compressed streams portable across platforms?*

Are zfp compressed streams portable across platforms? Are there, for example, endianness issues?

A: Yes, zfp can write portable compressed streams. To ensure portability across different endian platforms, the bit stream must however be written in increments of single bytes on big endian processors (e.g. PowerPC, SPARC), which is achieved by compiling zfp with an 8-bit (single-byte) word size:

```
-DBIT_STREAM_WORD_TYPE=uint8
```

See `BIT_STREAM_WORD_TYPE`. Note that on little endian processors (e.g. Intel x86-64 and AMD64), the word size does not affect the bit stream produced, and thus the default word size may be used. By default, zfp uses a word size of 64 bits, which results in the coarsest rate granularity but fastest (de)compression. If cross-platform portability is not needed, then the maximum word size is recommended (but see also *Q12*).

When using 8-bit words, zfp produces a compressed stream that is byte order independent, i.e. the exact same compressed sequence of bytes is generated on little and big endian platforms. When decompressing such streams, floating-point and integer values are recovered in the native byte order of the machine performing decompression. The decompressed values can be used immediately without the need for byte swapping and without having to worry about the byte order of the computer that generated the compressed stream.

Finally, zfp assumes that the floating-point format conforms to IEEE 754. Issues may arise on architectures that do not support IEEE floating point.

Q12: *How can I achieve finer rate granularity?*

A: For *d*-dimensional arrays, zfp supports a granularity of 8 / 4^d bits, i.e. the rate can be specified in increments of a fraction of a bit for 2D and 3D arrays. Such fine rate selection is always available for sequential compression (e.g. when calling `zfp_compress()`).

Unlike in sequential compression, zfp's compressed arrays require random access writes, which are supported only at the granularity of whole words. By default, a word is 64 bits, which gives a rate granularity of 64 / 4^d in *d* dimensions, i.e. 16 bits in 1D, 4 bits in 2D, and 1 bit in 3D.

To achieve finer granularity, recompile zfp with a smaller (but as large as possible) stream word size, e.g.:

```
-DBIT_STREAM_WORD_TYPE=uint8
```

gives the finest possible granularity, but at the expense of (de)compression speed. See `BIT_STREAM_WORD_TYPE`.

Q13: *Can I generate progressive zfp streams?*

A: Yes, but it requires some coding effort. There is currently no high-level support for progressive zfp streams. To implement progressive fixed-rate streams, the fixed-length bit streams should be interleaved among the blocks that make up an array. For instance, if a 3D array uses 1024 bits per block, then those 1024 bits could be broken down into, say, 16 pieces of 64 bits each, resulting in 16 discrete quality settings. By storing the blocks interleaved such that the first 64 bits of all blocks are contiguous, followed by the next 64 bits of all blocks, etc., one can achieve progressive decompression by setting the `zfp_stream.maxbits` parameter (see `zfp_stream_set_params()`) to the number of bits per block received so far.

To enable interleaving of blocks, zfp must first be compiled with:

```
-DBIT_STREAM_STRIDED
```

to enable strided bit stream access. In the example above, if the stream word size is 64 bits and there are *n* blocks, then:

```
stream_set_stride(stream, m, n);
```

implies that after every *m* 64-bit words have been decoded, the bit stream is advanced by $m \times n$ words to the next set of m 64-bit words associated with the block.

Q14: *How do I initialize the decompressor?*

A: The `zfp_stream` and `zfp_field` objects usually need to be initialized with the same values as they had during compression (but see *Q15* for exceptions). These objects hold the compression mode and parameters, and field data like the scalar type and dimensions. By default, these parameters are not stored with the compressed stream (the "codestream") and prior to zfp 0.5.0 had to be maintained separately by the application.

Since version 0.5.0, functions exist for reading and writing a 12- to 19-byte header that encodes compression and field parameters. For applications that wish to embed only the compression parameters, e.g. when the field dimensions are already known, there are separate functions that encode and decode this information independently.

Q15: *Must I use the same parameters during compression and decompression?*

A: Not necessarily. It is possible to use more tightly constrained `zfp_stream` parameters during decompression than were used during compression. For instance, one may use a larger `zfp_stream.minbits`, smaller `zfp_stream.maxbits`, smaller `zfp_stream.maxprec`, or larger `zfp_stream.minexp` during decompression to process fewer compressed bits than are stored, and to decompress the array more quickly at a lower precision. This may be useful in situations where the precision and accuracy requirements are not known a priori, thus forcing conservative settings during compression, or when the compressed stream is used for multiple purposes. For instance, visualization usually has less stringent precision requirements than quantitative data analysis. This feature of decompressing to a lower precision is particularly useful when the stream is stored progressively (see *Q13*).

Note that one may not use less constrained parameters during decompression, e.g. one cannot ask for more than `zfp_stream.maxprec` bits of precision when decompressing.

Currently float arrays have a different compressed representation from compressed double arrays due to differences in exponent width. It is not possible to compress a double array and then decompress (demote) the result to floats, for instance. Future versions of the zfp codec may use a unified representation that does allow this.

Q16: *Do strides have to match during compression and decompression?*

A: No. For instance, a 2D vector field:

```
float in[ny][nx][2];
```

could be compressed as two scalar fields with strides $sx = 2$, $sy = 2 \times nx$, and with pointers `&in[0][0][0]` and `&in[0][0][1]` to the first value of each scalar field. These two scalar fields can later be decompressed as non-interleaved fields:

```
float out[2][ny][nx];
```

using strides *sx* = 1, *sy* = *nx* and pointers `&out[0][0][0]` and `&out[1][0][0]`.

Q17: *Why does zfp sometimes not respect my error tolerance?*

A: zfp does not store each floating-point value independently, but represents a group of values (4, 16, or 64 values, depending on dimensionality) as linear combinations like averages by evaluating arithmetic expressions. Just like in uncompressed IEEE floating-point arithmetic, both representation error and roundoff error in the least significant bit(s) often occur.

To illustrate this, consider compressing the following 1D array of four floats:

```
float f[4] = { 1, 1e-1, 1e-2, 1e-3 };
```

using the zfp command-line tool:

```
zfp -f -1 4 -a 0 -i input.dat -o output.dat
```

In spite of an error tolerance of zero, the reconstructed values are:

```
float g[4] = { 1, 1e-1, 9.999998e-03, 9.999946e-04 };
```

with a (computed) maximum error of 5.472e-9. Because f[3] = 1e-3 can only be approximately represented in radix-2 floating-point, the actual error is even smaller: 5.424e-9. This reconstruction error is primarily due to zfp's block-floating-point representation, which expresses the four values in a block relative to a single, common binary exponent. Such exponent alignment occurs also in regular IEEE floating-point operations like addition. For instance,:

```
float x = (f[0] + f[3]) - 1;
```

should of course result in `x = f[3] = 1e-3`, but due to exponent alignment a few of the least significant bits of f[3] are lost in the addition, giving a result of `x = 1.0000467e-3` and a roundoff error of 4.668e-8. Similarly,:

```
float sum = f[0] + f[1] + f[2] + f[3];
```

should return `sum = 1.111`, but is computed as 1.1110000610. Moreover, the value 1.111 cannot even be represented exactly in (radix-2) floating-point; the closest float is 1.1109999. Thus the computed error:

```
float error = sum - 1.111f;
```

which itself has some roundoff error, is 1.192e-7.

Phew! Note how the error introduced by zfp (5.472e-9) is in fact one to two orders of magnitude smaller than the roundoff errors (4.668e-8 and 1.192e-7) introduced by IEEE floating-point in these computations. This lower error is in part due to zfp's use of 30-bit significands compared to IEEE's 24-bit single-precision significands. Note that data sets with a large dynamic range, e.g. where adjacent values differ a lot in magnitude, are more susceptible to representation errors.

The moral of the story is that error tolerances smaller than machine epsilon (relative to the data range) cannot always be satisfied by zfp. Nor are such tolerances necessarily meaningful for representing floating-point data that originated in floating-point arithmetic expressions, since accumulated roundoff errors are likely to swamp compression errors. Because such roundoff errors occur frequently in floating-point arithmetic, insisting on lossless compression on the grounds of accuracy is tenuous at best.

Q18: *Why is the actual rate sometimes not what I requested?*

A: In principle, zfp allows specifying the size of a compressed block in increments of single bits, thus allowing very fine-grained tuning of the bit rate. There are, however, cases when the desired rate does not exactly agree with the effective rate, and users are encouraged to check the return value of `zfp_stream_set_rate()`, which gives the actual rate.

There are several reasons why the requested rate may not be honored. First, the rate is specified in bits/value, while zfp always represents a block of 4^d values in d dimensions, i.e. using $N = 4^d \times rate$ bits. N must be an integer number of bits, which constrains the actual rate to be a multiple of $1 / 4^d$. The actual rate is computed by rounding 4^d times the desired rate.

Second, if the array dimensions are not multiples of four, then zfp pads the dimensions to the next higher multiple of four. Thus, the total number of bits for a 2D array of dimensions $nx \times ny$ is computed in terms of the number of blocks $bx \times by$:

```
bitsize = (4 * bx) * (4 * by) * rate
```

where $nx \leq 4 \times \mathrm{bx} < nx + 4$ and $ny \leq 4 \times by < ny + 4$. When amortizing bitsize over the $nx \times ny$ values, a slightly higher rate than requested may result.

Third, to support updating compressed blocks, as is needed by zfp's compressed array classes, the user may request write random access to the fixed-rate stream. To support this, each block must be aligned on a stream word boundary (see *Q12*), and therefore the rate when write random access is requested must be a multiple of $wordsize / 4^d$ bits. By default $wordsize = 64$ bits.

Fourth, for floating-point data, each block must hold at least the common exponent and one additional bit, which places a lower bound on the rate.

Finally, the user may optionally include a header with each array. Although the header is small, it must be accounted for in the rate. The function `zfp_stream_maximum_size()` conservatively includes space for a header, for instance.

Q19: *Can zfp perform compression in place?*

Because the compressed data tends to be far smaller than the uncompressed data, it is natural to ask if the compressed stream can overwrite the uncompressed array to avoid having to allocate separate storage for the compressed stream. zfp does allow for the possibility of such in-place compression, but with several caveats and restrictions:

1. A bitstream must be created whose buffer points to the beginning of uncompressed (and to be compressed) storage.
2. The array must be compressed using zfp's low-level API. In particular, the data must already be partitioned and organized into contiguous blocks so that all values of a block can be pulled out once and then replaced with the corresponding shorter compressed representation.
3. No one compressed block can occupy more space than its corresponding uncompressed block so that the not-yet compressed data is not overwritten. This is usually easily accomplished in fixed-rate mode, although the expert interface also allows guarding against this in all modes using the `zfp_stream.maxbits` parameter. This parameter should be set to `maxbits = 4^d * 8 * sizeof(type)`, where *d* is the array dimensionality (1, 2, or 3) and where *type* is the scalar type of the uncompressed data.
4. No header information may be stored in the compressed stream.

In-place decompression can also be achieved, but in addition to the above constraints requires even more care:

1. The data must be decompressed in reverse block order, so that the last block is decompressed first to the end of the block array. This requires the user to maintain a pointer to uncompressed storage and to seek via `stream_rseek()` to the proper location in the compressed stream where the block is stored.
2. The space allocated to the compressed stream must be large enough to also hold the uncompressed data.

An *example* is provided that shows how in-place compression can be done.

Q20: *How should I set the precision to bound the relative error?*

In general, zfp cannot bound the point-wise relative error due to its use of a block-floating-point representation, in which all values within a block are represented in relation to a single common exponent. For a high enough dynamic range within a block there may simply not be enough precision available to guard against loss. For instance, a block containing the values $2^0 = 1$ and 2^{-n} would require a precision of $n + 3$ bits to represent losslessly, and zfp uses at most 64-bit integers to represent values. Thus, if $n \geq 62$, then 2^{-n} is replaced with 0, which is a 100% relative error. Note that such loss also occurs when, for instance, 2^0 and 2^{-n} are added using floating-point arithmetic (see also *Q17*).

It is, however, possible to bound the error relative to the largest (in magnitude) value, *fmax*, within a block, which if the magnitude of values does not change too rapidly may serve as a reasonable proxy for point-wise relative errors.

One might then ask if using zfp's fixed-precision mode with *p* bits of precision ensures that the block-wise relative error is at most $2^{-p} \times fmax$. This is, unfortunately, not the case, because the requested precision, *p*, is ensured only for the transform coefficients. During the inverse transform of these quantized coefficients the quantization error may amplify. That being said, it is possible to derive a bound on the error in terms of *p* that would allow choosing an appropriate precision. Such a bound is derived below.

Let

```
emax = floor(log2(fmax))
```

be the largest base-2 exponent within a block. For transform coefficient precision, *p*, one can show that the maximum absolute error, *err*, is bounded by:

```
err <= k(d) * (2^emax / 2^p) <= k(d) * (fmax / 2^p)
```

Here *k*(*d*) is a constant that depends on the data dimensionality *d*:

```
k(d) = 20 * (15/4)^(d-1)
```

so that in 1D, 2D, and 3D we have:

```
k(1) = 20
k(2) = 125
k(3) = 1125/4
```

Thus, to guarantee *n* bits of accuracy in the decompressed data, we need to choose a higher precision, *p*, for the transform coefficients:

```
p(n, d) = n + ceil(log2(k(d))) = n + 2 * d + 3
```

so that

```
p(n, 1) = n + 5
p(n, 2) = n + 7
p(n, 3) = n + 9
```

This *p* value should be used in the call to `zfp_stream_set_precision()`.

Note, again, that some values in the block may have leading zeros when expressed relative to 2^{emax}, and these leading zeros are counted toward the *n*-bit precision. Using decimal to illustrate this, suppose we used 4-digit precision for a 1D block containing these four values:

```
-1.41421e+1 ~ -1.414e+1 = -1414 * (10^1 / 1000)
+2.71828e-1 ~ +0.027e+1 =   +27 * (10^1 / 1000)
+3.14159e-6 ~ +0.000e+1 =     0 * (10^1 / 1000)
+1.00000e+0 ~ +0.100e+1 =  +100 * (10^1 / 1000)
```

with the values in the middle column aligned to the common base-10 exponent +1, and with the values on the right expressed as scaled integers. These are all represented using four digits of precision, but some of those digits are leading zeros.

Q21: *Does zfp support lossless compression?*

Yes, and no. For integer data, zfp can with few exceptions ensure lossless compression. For a given n-bit integer type (n = 32 or n = 64), consider compressing p-bit signed integer data, with the sign bit counting toward the precision. In other words, there are exactly 2^p possible signed integers. If the integers are unsigned, then subtract 2^{p-1} first so that they range from -2^{p-1} to 2^{p-1} - 1.

Lossless compression is achieved by first promoting the p-bit integers to n - 1 bits (see *Q8*) such that all integer values fall in $[-2^{30}, +2^{30})$, when n = 32, or in $[-2^{62}, +2^{62})$, when n = 64. In other words, the p-bit integers first need to be shifted left by n - p - 1 bits. After promotion, the data should be compressed in zfp's fixed-precision mode using:

```
q = p + 4 * d + 1
```

bits of precision to ensure no loss, where d is the data dimensionality ($1 \leq d \leq 3$). Consequently, the p-bit data can be losslessly compressed as long as $p \leq n - 4 \times d - 1$. The table below lists the maximum precision p that can be losslessly compressed using 32- and 64-bit integer types.

d	n=32	n=64
1	27	59
2	23	55
3	19	51

Although lossless compression is possible as long as the precision constraint is met, the precision needed to guarantee no loss is generally much higher than the precision intrinsic in the uncompressed data, making lossless compression via zfp not competitive with compressors designed for lossless compression. Lossy integer compression with zfp can, on the other hand, work fairly well by using fewer than q bits of precision.

Furthermore, the minimum precision, q, given above is often larger than what is necessary in practice. There are worst-case inputs that do require such large q values, but they are quite rare.

The reason for expanded precision, i.e., why $q > p$, is that zfp's decorrelating transform computes averages of integers, and this transform is applied d times in d dimensions. Each average of two p-bit numbers requires p + 1 bits to avoid loss, and each transform can be thought of involving up to four such averaging operations.

For floating-point data, fully lossless compression with zfp is unlikely, albeit possible. If the dynamic range is low or varies slowly such that values within a 4^d block have the same or similar exponent, then the precision gained by discarding the 8 or 11 bits of the common floating-point exponents can offset the precision lost in the decorrelating transform. For instance, if all values in a block have the same exponent, then lossless compression is obtained using $q = 26 + 4 \times d \leq 32$ bits of precision for single-precision data and $q = 55 + 4 \times d \leq 64$ bits of precision for double-precision data. Of course, the constraint imposed by the available integer precision n implies that lossless compression of such data is possible only in 1D for single-precision data and only in 1D and 2D for double-precision data.

Q22: *Why is my actual, measured error so much smaller than the tolerance?*

For two reasons. The way zfp bounds the absolute error in *fixed-accuracy mode* is by keeping all transform coefficient bits whose place value exceeds the tolerance while discarding the less significant bits. Each such bit has a place value that is a power of two, and therefore the tolerance must first be rounded down to the next smaller power of two, which itself will introduce some slack. This possibly lower, effective tolerance is returned by the `zfp_stream_set_accuracy()` call.

Second, the quantized coefficients are then put through an inverse transform. This linear transform will combine signed quantization errors that, in the worst case, may cause them to add up and increase the error, even though the average (RMS) error remains the same, i.e. some errors cancel while others compound. For d-dimensional data, d such inverse transforms are applied, with the possibility of errors cascading across transforms. To account for the worst possible case, zfp has to conservatively lower its internal error tolerance further, once for each of the d transform passes.

Unless the data is highly oscillatory or noisy, the error is not likely to be magnified much, leaving an observed error in the decompressed data that is much lower than the prescribed tolerance. In practice, the observed maximum error tends to be about 4-8 times lower than the error tolerance for 3D data, while the difference is smaller for 2D and 1D data.

We recommend experimenting with tolerances and evaluating what error levels are appropriate for each application, e.g. by starting with a low, conservative tolerance and successively doubling it. The distribution of errors produced by zfp is approximately Gaussian, so even if the maximum error may seem large at an individual grid point, most errors tend to be much smaller and tightly clustered around zero.

CHAPTER 16

Troubleshooting

This section is intended for troubleshooting problems with zfp, in case any arise, and primarily focuses on how to correctly make use of zfp. If the decompressed data looks nothing like the original data, or if the compression ratios obtained seem not so impressive, then it is very likely that array dimensions or compression parameters have not been set correctly, in which case this troubleshooting guide could help.

The problems addressed in this section include:

1. *Is the data dimensionality correct?*
2. *Do the compressor and decompressor agree on the dimensionality?*
3. *Have the "smooth" dimensions been identified?*
4. *Are the array dimensions correct?*
5. *Are the array dimensions large enough?*
6. *Is the data logically structured?*
7. *Is the data set embedded in a regular grid?*
8. *Is the data provided to the zfp executable a raw binary array?*
9. *Is the byte order correct?*
10. *Is the floating-point precision correct?*
11. *Is the integer precision correct?*
12. *Is the data provided to the zfp executable a raw binary array?*
13. *Has the appropriate compression mode been set?*

P1: *Is the data dimensionality correct?*

This is one of the most common problems. First, make sure that zfp is given the correct dimensionality of the data. For instance, an audio stream is a 1D array, an image is a 2D array, and a volume grid is a 3D array. Sometimes a data set is a discrete collection of lower-dimensional objects. For instance, a stack of unrelated images (of the same size) could be represented in C as a 3D array:

```
imstack[count][ny][nx]
```

but since in this case the images are unrelated, no correlation would be expected along the third dimension–the underlying dimensionality of the data is here two. In this case, the images could be compressed one at a time, or they could be compressed together by treating the array dimensions as:

```
imstack[count * ny][nx]
```

Note that zfp partitions *d*-dimensional arrays into blocks of 4^d values. If *ny* above is not a multiple of four, then some blocks of 4 × 4 pixels will contain pixels from different images, which could hurt compression and/or quality. Still, this way of creating a single image by stacking multiple images is far preferable over linearizing each image into a 1D signal, and then compressing the images as:

```
imstack[count][ny * nx]
```

This loses the correlation along the *y* dimension, and further introduces discontinuities unless *nx* is a multiple of four.

Similarly to the example above, a 2D vector field

```
vfield[ny][nx][2]
```

could be declared as a 3D array, but the *x*- and *y*-components of the 2D vectors are likely entirely unrelated. In this case, each component needs to be compressed independently, either by rearranging the data as two scalar fields:

```
vfield[2][ny][nx]
```

or by using strides (see also *FAQ #1*). Note that in all these cases zfp will still compress the data, but if the dimensionality is not correct then the compression ratio will suffer.

P2: *Do the compressor and decompressor agree on the dimensionality?*

Consider compressing a 3D array:

```
double a[1][1][100]
```

with *nx* = 100, *ny* = 1, *nz* = 1, then decompressing the result to a 1D array:

```
double b[100]
```

with *nx* = 100. Although the arrays *a* and *b* occupy the same amount of memory and are in C laid out similarly, these arrays are not equivalent to zfp because their dimensionalities differ. zfp uses different CODECs to (de)compress 1D, 2D, and 3D arrays, and the 1D decompressor expects a compressed bit stream that corresponds to a 1D array.

What happens in practice in this case is that the array *a* is compressed using zfp's 3D CODEC, which first pads the array to

```
double padded[4][4][100]
```

When this array is correctly decompressed using the 3D CODEC, the padded values are generated but discarded. zfp's 1D decompressor, on the other hand, expects 100 values, not 100 × 4 × 4 = 1600 values, and therefore likely returns garbage.

P3: *Have the "smooth" dimensions been identified?*

Closely related to *P1* above, some fields simply do not vary smoothly along all dimensions, and zfp can do a good job compressing only those dimensions that exhibit some coherence. For instance, consider a table of stock prices indexed by date and stock:

```
price[stocks][dates]
```

One could be tempted to compress this as a 2D array, but there is likely little to no correlation in prices between different stocks. Each such time series should be compressed independently as a 1D signal.

What about time-varying images like a video sequence? In this case, it is likely that there is correlation over time, and that the value of a single pixel varies smoothly in time. It is also likely that each image exhibits smoothness along its two spatial dimensions. So this can be treated as a single, 3D data set.

How about time-varying volumes, such as

```
field[nt][nz][ny][nx]
```

zfp currently supports only 1D, 2D, and 3D arrays, whereas a time-varying volume is 4D. Here the data should ideally be organized by the three "smoothest" dimensions. Given the organization above, this could be compressed as a 3D array:

```
field[nt * nz][ny][nx]
```

Again, do **not** compress this as a 3D array:

```
field[nt][nz][ny * nx]
```

P4: *Are the array dimensions correct?*

This is another common problem that seems obvious, but often the dimensions are accidentally transposed. Assuming that the smooth dimensions have been identified, it is important that the dimensions are listed in the correct order. For instance, if the data (in C notation) is organized as:

```
field[d1][d2][d3]
```

then the data is organized in memory (or on disk) with the d3 dimension varying fastest, and hence $nx = d3$, $ny = d2$, $nz = d1$ using the zfp naming conventions for the dimensions, e.g. the *zfp executable* should be invoked with:

```
zfp -3 d3 d2 d1
```

in this case. Things will go horribly wrong if zfp in this case is called with $nx = d1$, $ny = d2$, $nz = d3$. The entire data set will still compress and decompress, but compression ratio and quality will suffer greatly.

P5: *Are the array dimensions large enough?*

zfp partitions *d*-dimensional data sets into blocks of 4^d values, e.g. in 3D a block consists of 4 × 4 × 4 values. If the dimensions are not multiples of four, then zfp will "pad" the array to the next larger multiple of four. Such padding can hurt compression. In particular, if one or more of the array dimensions are small, then the overhead of such padding could be significant.

Consider compressing a collection of 1000 small 3D arrays:

```
field[1000][5][14][2]
```

zfp would first logically pad this to a larger array:

```
field[1000][8][16][4]
```

which is (8 × 16 × 4) / (5 × 14 × 2) ~ 3.66 times larger. Although such padding often compresses well, this still represents a significant overhead.

If a large array has been partitioned into smaller pieces, it may be best to reassemble the larger array. Or, when possible, ensure that the sub-arrays have dimensions that are multiples of four.

P6: *Is the data logically structured?*

zfp was designed for logically structured data, i.e. Cartesian grids. It works much like an image compressor does, which assumes that the data set is a structured array of pixels, and it assumes that values vary reasonably smoothly on average, just like natural images tend to contain large regions of uniform color or smooth color gradients, like a blue sky, smoothly varying skin tones of a human's face, etc. Many data sets are not represented on a regular grid. For instance, an array of particle *xyz* positions:

```
points[count][3]
```

is a 2D array, but does not vary smoothly in either dimension. Furthermore, such unstructured data sets need not be organized in any particular order; the particles could be listed in any arbitrary order. One could attempt to sort the particles, for example by the *x* coordinate, to promote smoothness, but this would still leave the other two dimensions non-smooth.

Sometimes the underlying dimensions are not even known, and only the total number of floating-point values is known. For example, suppose we only knew that the data set contained $n = count \times 3$ values. One might be tempted to compress this using zfp's 1-dimensional compressor, but once again this would not work well. Such abuse of zfp is much akin to trying to compress an image using an audio compressor like mp3, or like compressing an *n*-sample piece of music as an *n*-by-one sized image using an image compressor like JPEG. The results would likely not be very good.

Some data sets are logically structured but geometrically irregular. Examples include fields stored on Lagrangian meshes that have been warped, or on spectral element grids, which use a non-uniform grid spacing. zfp assumes that the data has been regularly sampled in each dimension, and the more the geometry of the sampling deviates from uniform, the worse compression gets. Note that rectilinear grids with different but uniform grid spacing in each dimension are fine. If your application uses very non-uniform sampling, then resampling onto a uniform grid (if possible) may be advisable.

Other data sets are "block structured" and consist of piecewise structured grids that are "glued" together. Rather than treating such data as unstructured 1D streams, consider partitioning the data set into independent (possibly overlapping) regular grids.

P7: *Is the data set embedded in a regular grid?*

Some applications represent irregular geometry on a Cartesian grid, and leave portions of the domain unspecified. Consider, for instance, sampling the density of the Earth onto a Cartesian grid. Here the density for grid points outside the Earth is unspecified.

In this case, zfp does best by initializing the "background field" to all zeros. In zfp's *fixed-accuracy mode*, any "empty" block that consists of all zeros is represented using a single bit, and therefore the overhead of representing empty space can be kept low.

P8: *Have fill values, NaNs, and infinities been removed?*

It is common to signal unspecified values using what is commonly called a "fill value," which is a special constant value that tends to be far out of range of normal values. For instance, in climate modeling the ocean temperature over land is meaningless, and it is common to use a very large temperature value such as 1e30 to signal that the temperature is undefined for such grid points.

Very large fill values do not play well with zfp, because they both introduce artificial discontinuities and pollute nearby values by expressing them all with respect to the common largest exponent within their block. Assuming a fill value of 1e30, the value pi in the same block would be represented as:

```
0.00000000000000000000000000000314159... * 1e30
```

Given finite precision, the small fraction would likely be replaced with zero, resulting in complete loss of the actual value being stored.

Other applications use NaNs (special not-a-number values) or infinities as fill values. These are even more problematic, because they do not have a defined exponent. zfp relies on the C function `frexp()` to compute the exponent of the largest (in magnitude) value within a block, but produces unspecified behavior if that value is not finite.

zfp currently has no independent mechanism for handling fill values. Ideally such special values would be signalled separately, e.g. using a bit mask, and then replaced with zeros to ensure that they both compress well and do not pollute actual data.

P9: *Is the byte order correct?*

zfp generally works with the native byte order (e.g. little or big endian) of the machine it is compiled on. One needs only be concerned with byte order when reading raw, binary data into the zfp executable, when exchanging compressed files across platforms, and when varying the bit stream word size on big endian machines (not common). For instance, to compress a binary double-precision floating-point file stored in big endian byte order on a little endian machine, byte swapping must first be done. For example, on Linux and macOS, 8-byte doubles can be byte swapped using:

```
objcopy -I binary -O binary --reverse-bytes=8 big.bin little.bin
```

See also FAQ *#11* for more discussion of byte order.

P10: *Is the floating-point precision correct?*

Another obvious problem: Please make sure that zfp is told whether the data to compress is an array of single- (32-bit) or double-precision (64-bit) values, e.g. by specifying the *-f* or *-d* options to the **`zfp`** executable or by passing the appropriate *`zfp_type`* to the C functions.

P11: *Is the integer precision correct?*

zfp currently supports compression of 31- or 63-bit signed integers. Shorter integers (e.g., bytes, shorts) can be compressed but must first be promoted to one of the longer types. This should always be done using zfp's functions for *promotion and demotion*, which both perform bit shifting and biasing to handle both signed and unsigned types. It is not sufficient to simply cast short integers to longer integers. See also FAQs *#8* and *#9*.

P12: *Is the data provided to the zfp executable a raw binary array?*

zfp expects that the input file is a raw binary array of integers or floating-point values in the IEEE format, e.g. written to file using `fwrite()`. Do not hand zfp a text file containing ASCII floating-point numbers. Strip the file of any header information. Languages like Fortran tend to store with the array its size. No such metadata may be embedded in the file.

P13: *Has the appropriate compression mode been set?*

zfp provides three different *modes of compression* that trade storage and accuracy. In fixed-rate mode, the user specifies the exact number of bits (often in increments of a fraction of a bit) of compressed storage per value (but see FAQ *#18* for caveats). From the user's perspective, this seems a very desirable feature, since it provides for a direct mechanism for specifying how much storage to use. However, there is often a large quality penalty associated with the fixed-rate mode, because each block of 4^d values is allocated the same number of bits. In practice, the information content over the data set varies significantly, which means that easy-to-compress regions are assigned too many bits, while too few bits are available to faithfully represent the more challenging-to-compress regions. Although one of the unique features of zfp, its fixed-rate mode should primarily be used only when random access to the data is needed.

zfp also provides a fixed-precision mode, where the user specifies how many uncompressed significant bits to use to represent the floating-point fraction. This precision may not be exactly what people might normally think of. For instance, the C float type is commonly referred to as 32-bit precision. However, the sign bit and exponent account for nine of those bits and do not contribute to the number of significant bits of precision. Furthermore, for normal numbers, IEEE uses a hidden implicit one bit, so most float values actually have 24 bits of precision. Furthermore, zfp uses a block-floating-point representation with a single exponent per block, which may cause some small values to have several leading zero bits and therefore less precision than requested. Thus, the effective precision returned by zfp in its fixed-precision mode may in fact vary. In practice, the precision requested is only an upper bound, though typically at least one value within a block has the requested precision.

Finally, zfp supports a fixed-accuracy mode, which except in rare circumstances (see FAQ *#17*) ensures that the absolute error is bounded, i.e. the difference between any decompressed and original value is at most the tolerance specified by the user (but usually several times smaller). Whenever possible, we recommend using this compression mode, which depending on how easy the data is to compress results in the smallest compressed stream that respects the error tolerance.

There is also an expert mode that allows the user to combine the constraints of fixed rate, precision, and accuracy. See the section on *compression modes* for more details.

CHAPTER 17

Limitations

zfp has evolved from a research prototype to a library that is approaching production readiness. However, the API and even the compression codec are still undergoing changes as new important features are added.

Below is a list of known limitations of the current version of zfp. See the section on *Future Directions* for a discussion of planned features that will address some of these limitations.

- The current version of zfp allows for near lossless compression through suitable parameter choices, but no guarantees are made that bit-for-bit lossless compression is achieved. We envision supporting lossless compression in a future version.
- Special values like infinity and NaN are not supported. Subnormal floating-point numbers are, however, correctly handled. There is an implicit assumption that floating point conforms to IEEE, though extensions to other floating-point formats should be possible with minor effort.
- Conventional pointers and references to individual array elements are not available. That is, constructions like `double* ptr = &a[i];` are not possible when `a` is a zfp array. However, as of zfp 0.5.2, *proxy pointers* are available that act much like pointers to uncompressed data. Similarly, operators `[]` and `()` do not return regular C++ references. Instead, a *proxy reference* class is used (similar to how STL bit vectors are implemented). These proxy references and pointers can, however, safely be passed to functions and used where regular references and pointers can.
- Although the current version of zfp supports *iterators*, *pointers*, and *references* to array elements, 'const' versions of these accessors are not yet available for read-only access.
- There currently is no way to make a complete copy of a compressed array, i.e. a = b; does not work for arrays a and b. Copy constructors and assignment operators will be added in the near future.
- zfp can potentially provide higher precision than conventional float and double arrays, but the interface currently does not expose this. For example, such added precision could be useful in finite difference computations, where catastrophic cancellation can be an issue when insufficient precision is available.
- Only single and double precision types are supported. Generalizations to IEEE half and quad precision would be useful. For instance, compressed 64-bit-per-value storage of 128-bit quad-precision numbers could greatly improve the accuracy of double-precision floating-point computations using the same amount of storage.

- Complex-valued arrays are not directly supported. Real and imaginary components must be stored as separate arrays, which may result in lost opportunities for compression, e.g. if the complex magnitude is constant and only the phase varies.
- zfp arrays are not thread-safe. We are considering options for supporting multi-threaded access, e.g. for OpenMP parallelization.
- This version of zfp does not run on the GPU. Some work has been done to port zfp to CUDA, and an experimental version is available.

CHAPTER 18

Future Directions

zfp is actively being developed and plans have been made to add a number of important features, including:

- **Support for 4D arrays**, e.g., for compressing time-varying 3D fields. Although the zfp compression algorithm trivially generalizes to higher dimensions, *d*, the current implementation is hampered by the lack of integer types large enough to hold 4^d bits for $d > 3$. For now, higher-dimensional data should be compressed as collections of independent 3D fields.
- **Tagging of missing values**. zfp currently assumes that arrays are dense, i.e., each array element stores a valid numerical value. In many science applications this is not the case. For instance, in climate modeling, ocean temperature is not defined over land. In other applications, the domain is not rectangular but irregular and embedded in a rectangular array. Such examples of sparse arrays demand a mechanism to tag values as missing or indeterminate. Current solutions often rely on tagging missing values as NaNs or special, often very large sentinel values outside the normal range, which can lead to poor compression and complete loss of accuracy in nearby valid values. See *FAQ #7*.
- **Support for NaNs and infinities**. Similar to missing values, some applications store special IEEE floating-point values that are not yet supported by zfp. In fact, the presence of such values will currently result in undefined behavior and loss of data for all values within a block that contains non-finite values.
- **Lossless compression**. Although zfp can usually limit compression errors to within floating-point roundoff error, some applications demand bit-for-bit accurate reconstruction. Strategies for lossless compression are currently being evaluated.
- **Progressive decompression**. Streaming large data sets from remote storage for visualization can be time consuming, even when the data is compressed. Progressive streaming allows the data to be reconstructed at reduced precision over the entire domain, with quality increasing progressively as more data arrives. The low-level bit stream interface already supports progressive access by interleaving bits across blocks (see *FAQ #13*), but zfp lacks a high-level API for generating and accessing progressive streams.
- **Parallel compression**. zfp's data partitioning into blocks invites opportunities for data parallelism on multithreaded platforms by dividing the blocks among threads. An OpenMP implementation of parallel compression is under development that produces compressed streams that are identical to serially compressed streams. An experimental CUDA implementation for parallel compression and decompression on the GPU is also under development.

- **Thread-safe arrays**. zfp's compressed arrays are not thread-safe, even when performing read accesses only. The primary reason is that the arrays employ caching, which requires special protection to avoid race conditions. Work is planned to support both read-only and read-write accessible arrays that are thread-safe, most likely by using thread-local caches for read-only access and disjoint sub-arrays for read-write access, where each thread has exclusive ownership of a portion of the array.
- **Variable-rate arrays**. zfp currently supports only fixed-rate compressed arrays, which wastes bits in smooth regions with little information content while too few bits may be allocated to accurately preserve sharp features such as shocks and material interfaces, which tend to drive the physics in numerical simulations. Two candidate solutions have been identified for read-only and read-write access to variable-rate arrays with very modest storage overhead. These arrays will support both fixed precision and accuracy.
- **Array operations**. zfp's compressed arrays currently support basic indexing and initialization, but lack essential features such as shallow and deep copies, slicing, views, etc. Work is underway to address these deficiencies.
- **Language bindings**. The main compression codec is written in C89 to facilitate calls from other languages, but would benefit from language wrappers to ease integration. zfp's compressed arrays exploit the operator overloading provided by C++, and therefore can currently not be used in other languages, including C. Work is planned to add complete language bindings for C, C++, Fortran, and Python.

Please contact Peter Lindstrom with requests for features not listed above.

CHAPTER 19

Contributors

- LLNL zfp team
  - Peter Lindstrom (Project Lead)
  - Matt Larsen (CUDA port)
  - Mark Miller (HDF5 plugin)
  - Markus Salasoo (regression tests, software engineering)
- External contributors
  - Chuck Atkins, Kitware (CMake support)
  - Stephen Hamilton, Johns Hopkins University (VTK plugin)
  - Mark Kim, ORNL (original CUDA port)
  - Amik St-Cyr, Shell (OpenMP compressor)
  - Eric Suchyta, ORNL (ADIOS plugin)

CHAPTER 20

Release Notes

zfp 0.5.2, September 28, 2017

- Added iterators and proxy objects for pointers and references.
- Added example illustrating how to use iterators and pointers.
- Modified diffusion example to optionally use iterators.
- Moved internal headers under array to array/zfp.
- Modified 64-bit integer typedefs to avoid the C89 non-compliant long long and allow for user-supplied types and literal suffixes.
- Renamed compile-time macros that did not have a ZFP prefix.
- Fixed issue with setting stream word type via CMake.
- Rewrote documentation in reStructuredText and added complete documentation of all public functions, classes, types, and macros. Removed ASCII documentation.

zfp 0.5.1, March 28, 2017

- This release primarily fixes a few minor issues but also includes changes in anticipation of a large number of planned future additions to the library. No changes have been made to the compressed format, which is backwards compatible with version 0.5.0.
- Added high-level API support for integer types.
- Separated library version from CODEC version and added version string.
- Added example that illustrates in-place compression.
- Added support for CMake builds.
- Corrected inconsistent naming of BIT_STREAM macros in code and documentation.
- Renamed some of the header bit mask macros.
- Added return values to stream_skip and stream_flush to indicate the number of bits skipped or output.

- Renamed stream_block and stream_delta to make it clear that they refer to strided streams. Added missing definition of stream_stride_block.
- Changed int/uint types in places to use ptrdiff_t/size_t where appropriate.
- Changed API for zfp_set_precision and zfp_set_accuracy to not require the scalar type.
- Added missing static keyword in decode_block.
- Changed testzfp to allow specifying which tests to perform on the command line.
- Fixed bug that prevented defining uninitialized arrays.
- Fixed incorrect computation of array sizes in zfp_field_size.
- Fixed minor issues that prevented code from compiling on Windows.
- Fixed issue with fixed-accuracy headers that caused unnecessary storage.
- Modified directory structure.
- Added documentation that discusses common issues with using zfp.

zfp 0.5.0, February 29, 2016

- Modified CODEC to more efficiently encode blocks whose values are all zero or are smaller in magnitude than the absolute error tolerance. This allows representing "empty" blocks using only one bit each. This version is not backwards compatible with prior zfp versions.
- Changed behavior of zfp_compress and zfp_decompress to not automatically rewind the bit stream. This makes it easier to concatenate multiple compressed bit streams, e.g. when compressing vector fields or multiple scalars together.
- Added functions for compactly encoding the compression parameters and field meta data, e.g. for producing self-contained compressed streams. Also added functions for reading and writing a header containing these parameters.
- Changed the zfp example program interface to allow reading and writing compressed streams, optionally with a header. The zfp tool can now be used to compress and decompress files as a stand alone utility.

zfp 0.4.1, December 28, 2015

- Fixed bug that caused segmentation fault when compressing 3D arrays whose dimensions are not multiples of four. Specifically, arrays of dimensions nx * ny * nz, with ny not a multiple of four, were not handled correctly.
- Modified examples/fields.h to ensure standard compliance. Previously, C99 support was needed to handle the hex float constants, which are not supported in C++98.
- Added simple.c as a minimal example of how to call the compressor.
- Changed compilation of diffusion example to output two executables: one with and one without compression.

zfp 0.4.0, December 5, 2015

- Substantial changes to the compression algorithm that improve PSNR by about 6 dB and speed by a factor of 2-3. These changes are not backward compatible with previous versions of zfp.
- Added support for 31-bit and 63-bit integer data, as well as shorter integer types.
- Rewrote compression codec entirely in C to make linking and calling easier from other programming languages, and to expose the low-level interface through C instead of C++. This necessitated significant changes to the API as well.
- Minor changes to the C++ compressed array API, as well as major implementation changes to support the C library. The namespace and public types are now all in lower case.
- Deprecated support for general fixed-point decorrelating transforms and slimmed down implementation.

- Added new examples for evaluating the throughput of the (de)compressor and for compressing grayscale images in the pgm format.
- Added FAQ.

zfp 0.3.2, December 3, 2015

- Fixed bug in Array::get() that caused the wrong cached block to be looked up, thus occasionally copying incorrect values back to parts of the array.

zfp 0.3.1, May 6, 2015

- Fixed rare bug caused by exponent underflow in blocks with no normal and some subnormal numbers.

zfp 0.3.0, March 3, 2015

- Modified the default decorrelating transform to one that uses only additions and bit shifts. This new transform, in addition to being faster, also has some theoretical optimality properties and tends to improve rate distortion.
- Added compile-time support for parameterized transforms, e.g. to support other popular transforms like DCT, HCT, and Walsh-Hadamard.
- Made forward transform range preserving: (-1, 1) is mapped to (-1, 1). Consequently Q1.62 fixed point can be used throughout.
- Changed the order in which bits are emitted within each bit plane to be more intelligent. Group tests are now deferred until they are needed, i.e. just before the value bits for the group being tested. This improves the quality of fixed-rate encodings, but has no impact on compressed size.
- Made several optimizations to improve performance.
- Added floating-point traits to reduce the number of template parameters. It is now possible to declare a 3D array as Array3<float>, for example.
- Added functions for setting the array scalar type and dimensions.
- Consolidated several header files.
- Added testzfp for regression testing.

zfp 0.2.1, December 12, 2014

- Added Win64 support via Microsoft Visual Studio compiler.
- Fixed broken support for IBM's xlc compiler.
- Made several minor changes to suppress compiler warnings.
- Documented expected output for the diffusion example.

zfp 0.2.0, December 2, 2014

- The compression interface from zfpcompress was relocated to a separate library, called libzfp, and modified to be callable from C. This API now uses a parameter object (zfp_params) to specify array type and dimensions as well as compression parameters.
- Several utility functions were added to simplify libzfp usage:
  - Functions for setting the rate, precision, and accuracy. Corresponding functions were also added to the Codec class.
  - A function for estimating the buffer size needed for compression.
- The Array class functionality was expanded:
  - Support for accessing the compressed bit stream stored with an array, e.g. for offline compressed storage and for initializing an already compressed array.

  - Functions for dynamically specifying the cache size.
  - The default cache is now direct-mapped instead of two-way associative.
- Minor bug fixes:
  - Corrected the value of the lowest possible bit plane to account for both the smallest exponent and the number of bits in the significand.
  - Corrected inconsistent use of rate and precision. The rate refers to the number of compressed bits per floating-point value, while the precision refers to the number of uncompressed bits. The Array API was changed accordingly.

zfp 0.1.0, November 12, 2014

- Initial beta release.

Index

Symbols

-1 <nx>
  - command line option, 53

-2 <nx> <ny>
  - command line option, 53

-3 <nx> <ny> <nz>
  - command line option, 53

-a <tolerance>
  - command line option, 53

-c <minbits> <maxbits> <maxprec> <minexp>
  - command line option, 53

-d
  - command line option, 53

-f
  - command line option, 53

-h
  - command line option, 52

-i <path>
  - command line option, 52

-o <path>
  - command line option, 52

-p <precision>
  - command line option, 53

-q
  - command line option, 52

-r <rate>
  - command line option, 53

-s
  - command line option, 52

-t <type>
  - command line option, 53

-z <path>
  - command line option, 52

B

BIT_STREAM_STRIDED (C macro), 9
BIT_STREAM_WORD_TYPE (C macro), 9
bitstream (C type), 30

C

command line option
  - -1 <nx>, 53
  - -2 <nx> <ny>, 53
  - -3 <nx> <ny> <nz>, 53
  - -a <tolerance>, 53
  - -c <minbits> <maxbits> <maxprec> <minexp>, 53
  - -d, 53
  - -f, 53
  - -h, 52
  - -i <path>, 52
  - -o <path>, 52
  - -p <precision>, 53
  - -q, 52
  - -r <rate>, 53
  - -s, 52
  - -t <type>, 53
  - -z <path>, 52

Compression mode, 12
  - Expert mode, 13
  - Fixed-accuracy mode, 15
  - Fixed-precision mode, 14
  - Fixed-rate mode, 14

Configuration, 8

E

environment variable
  - LD_LIBRARY_PATH, 7

I

Iterators, 38

L

LD_LIBRARY_PATH, 7

P

Pointers, 37

R

Rate, 14

References, 36

S

stream_align (C function), 31
stream_capacity (C function), 30
stream_close (C function), 30
stream_data (C function), 30
stream_flush (C function), 31
stream_open (C function), 30
stream_pad (C function), 31
stream_read_bit (C function), 31
stream_read_bits (C function), 31
stream_rewind (C function), 31
stream_rseek (C function), 31
stream_rtell (C function), 31
stream_set_stride (C function), 31
stream_size (C function), 30
stream_skip (C function), 31
stream_stride_block (C function), 31
stream_stride_delta (C function), 31
stream_word_bits (C variable), 30
stream_write_bit (C function), 31
stream_write_bits (C function), 31
stream_wseek (C function), 31
stream_wtell (C function), 31

W

word (C type), 30

Z

zfp::array (C++ class), 34
zfp::array1 (C++ class), 35
zfp::array1::array1 (C++ function), 35
zfp::array1::iterator (C++ class), 38
zfp::array1::iterator::operator+ (C++ function), 40
zfp::array1::iterator::operator+= (C++ function), 40
zfp::array1::iterator::operator- (C++ function), 40
zfp::array1::iterator::operator– (C++ function), 40
zfp::array1::iterator::operator-= (C++ function), 40
zfp::array1::iterator::operator> (C++ function), 40
zfp::array1::iterator::operator>= (C++ function), 40
zfp::array1::iterator::operator< (C++ function), 40
zfp::array1::iterator::operator<= (C++ function), 40
zfp::array1::iterator::operator[] (C++ function), 40
zfp::array1::operator() (C++ function), 36
zfp::array1::pointer (C++ class), 37
zfp::array1::reference (C++ class), 36
zfp::array1::resize (C++ function), 35
zfp::array2 (C++ class), 35
zfp::array2::array2 (C++ function), 35
zfp::array2::iterator (C++ class), 38
zfp::array2::operator() (C++ function), 36
zfp::array2::pointer (C++ class), 37
zfp::array2::reference (C++ class), 36
zfp::array2::resize (C++ function), 35
zfp::array2::size_x (C++ function), 35
zfp::array2::size_y (C++ function), 35
zfp::array3 (C++ class), 35
zfp::array3::array3 (C++ function), 35
zfp::array3::iterator (C++ class), 39
zfp::array3::operator() (C++ function), 36
zfp::array3::pointer (C++ class), 37
zfp::array3::reference (C++ class), 36
zfp::array3::resize (C++ function), 35
zfp::array3::size_x (C++ function), 35
zfp::array3::size_y (C++ function), 35
zfp::array3::size_z (C++ function), 35
zfp::array::~array (C++ function), 34
zfp::array::array (C++ function), 34
zfp::array::begin (C++ function), 35
zfp::array::cache_size (C++ function), 34
zfp::array::clear_cache (C++ function), 34
zfp::array::compressed_data (C++ function), 34
zfp::array::compressed_size (C++ function), 34
zfp::array::end (C++ function), 35
zfp::array::flush_cache (C++ function), 34
zfp::array::get (C++ function), 34
zfp::array::operator= (C++ function), 34
zfp::array::operator[] (C++ function), 34, 35
zfp::array::rate (C++ function), 34
zfp::array::set (C++ function), 34
zfp::array::set_cache_size (C++ function), 34
zfp::array::set_rate (C++ function), 34
zfp::array::size (C++ function), 34
zfp::arrayANY::iterator::difference_type (C++ type), 39
zfp::arrayANY::iterator::i (C++ function), 39
zfp::arrayANY::iterator::iterator_category (C++ type), 39
zfp::arrayANY::iterator::j (C++ function), 39
zfp::arrayANY::iterator::k (C++ function), 39
zfp::arrayANY::iterator::operator!= (C++ function), 39
zfp::arrayANY::iterator::operator* (C++ function), 39
zfp::arrayANY::iterator::operator++ (C++ function), 39
zfp::arrayANY::iterator::operator= (C++ function), 39
zfp::arrayANY::iterator::operator== (C++ function), 39
zfp::arrayANY::iterator::pointer (C++ type), 39
zfp::arrayANY::iterator::reference (C++ type), 39
zfp::arrayANY::iterator::value_type (C++ type), 39
zfp::arrayANY::pointer::operator!= (C++ function), 38
zfp::arrayANY::pointer::operator* (C++ function), 38
zfp::arrayANY::pointer::operator+ (C++ function), 38
zfp::arrayANY::pointer::operator++ (C++ function), 38
zfp::arrayANY::pointer::operator+= (C++ function), 38
zfp::arrayANY::pointer::operator- (C++ function), 38
zfp::arrayANY::pointer::operator– (C++ function), 38
zfp::arrayANY::pointer::operator-= (C++ function), 38
zfp::arrayANY::pointer::operator= (C++ function), 38
zfp::arrayANY::pointer::operator== (C++ function), 38
zfp::arrayANY::pointer::operator[] (C++ function), 38

zfp::arrayANY::reference::operator*= (C++ function), 37
zfp::arrayANY::reference::operator+= (C++ function), 37
zfp::arrayANY::reference::operator-= (C++ function), 37
zfp::arrayANY::reference::operator/= (C++ function), 37
zfp::arrayANY::reference::operator= (C++ function), 37
zfp::arrayANY::reference::operator& (C++ function), 37
ZFP_BIT_STREAM_WORD_SIZE (C macro), 9
ZFP_CODEC (C macro), 18
zfp_codec_version (C variable), 19
zfp_compress (C function), 22
zfp_decode_block_double_1 (C function), 25
zfp_decode_block_double_2 (C function), 26
zfp_decode_block_double_3 (C function), 26
zfp_decode_block_float_1 (C function), 25
zfp_decode_block_float_3 (C function), 26
zfp_decode_block_int32_1 (C function), 25
zfp_decode_block_int32_3 (C function), 26
zfp_decode_block_int64_1 (C function), 25
zfp_decode_block_int64_3 (C function), 26
zfp_decode_block_strided_double_1 (C function), 26
zfp_decode_block_strided_double_3 (C function), 27
zfp_decode_block_strided_float_1 (C function), 26
zfp_decode_block_strided_float_3 (C function), 26
zfp_decode_block_strided_int32_1 (C function), 26
zfp_decode_block_strided_int32_3 (C function), 26
zfp_decode_block_strided_int64_1 (C function), 26
zfp_decode_block_strided_int64_3 (C function), 26
zfp_decode_partial_block_strided_double_3 (C function), 27
zfp_decode_partial_block_strided_float_3 (C function), 27
zfp_decode_partial_block_strided_int32_3 (C function), 27
zfp_decode_partial_block_strided_int64_3 (C function), 27
zfp_decompress (C function), 22
zfp_demote_int32_to_int16 (C function), 27
zfp_demote_int32_to_int8 (C function), 27
zfp_demote_int32_to_uint16 (C function), 27
zfp_demote_int32_to_uint8 (C function), 27
zfp_encode_block_double_1 (C function), 24
zfp_encode_block_double_2 (C function), 24
zfp_encode_block_double_3 (C function), 25
zfp_encode_block_float_1 (C function), 24
zfp_encode_block_float_2 (C function), 24
zfp_encode_block_float_3 (C function), 25
zfp_encode_block_int32_1 (C function), 24
zfp_encode_block_int32_2 (C function), 24
zfp_encode_block_int32_3 (C function), 25
zfp_encode_block_int64_1 (C function), 24
zfp_encode_block_int64_2 (C function), 24
zfp_encode_block_int64_3 (C function), 25
zfp_encode_block_strided_double_1 (C function), 24
zfp_encode_block_strided_double_2 (C function), 24
zfp_encode_block_strided_double_3 (C function), 25
zfp_encode_block_strided_float_1 (C function), 24
zfp_encode_block_strided_float_2 (C function), 24
zfp_encode_block_strided_float_3 (C function), 25
zfp_encode_block_strided_int32_1 (C function), 24
zfp_encode_block_strided_int32_2 (C function), 24
zfp_encode_block_strided_int32_3 (C function), 25
zfp_encode_block_strided_int64_1 (C function), 24
zfp_encode_block_strided_int64_2 (C function), 24
zfp_encode_block_strided_int64_3 (C function), 25
zfp_encode_partial_block_strided_double_1 (C function), 24
zfp_encode_partial_block_strided_double_2 (C function), 25
zfp_encode_partial_block_strided_double_3 (C function), 25
zfp_encode_partial_block_strided_float_1 (C function), 24
zfp_encode_partial_block_strided_float_2 (C function), 25
zfp_encode_partial_block_strided_float_3 (C function), 25
zfp_encode_partial_block_strided_int32_1 (C function), 24
zfp_encode_partial_block_strided_int32_2 (C function), 24
zfp_encode_partial_block_strided_int32_3 (C function), 25
zfp_encode_partial_block_strided_int64_1 (C function), 24
zfp_encode_partial_block_strided_int64_2 (C function), 25
zfp_encode_partial_block_strided_int64_3 (C function), 25
zfp_field (C type), 19
zfp_field_1d (C function), 21
zfp_field_2d (C function), 21
zfp_field_3d (C function), 21
zfp_field_alloc (C function), 21
zfp_field_dimensionality (C function), 21
zfp_field_free (C function), 21
zfp_field_metadata (C function), 22
zfp_field_pointer (C function), 21
zfp_field_precision (C function), 21
zfp_field_set_metadata (C function), 22
zfp_field_set_pointer (C function), 22
zfp_field_set_size_1d (C function), 22
zfp_field_set_size_2d (C function), 22
zfp_field_set_size_3d (C function), 22
zfp_field_set_stride_1d (C function), 22
zfp_field_set_stride_2d (C function), 22
zfp_field_set_stride_3d (C function), 22
zfp_field_set_type (C function), 22
zfp_field_size (C function), 21

zfp_field_stride (C function), 21
zfp_field_type (C function), 21
ZFP_HEADER_FULL (C macro), 18
ZFP_HEADER_MAGIC (C macro), 18
ZFP_HEADER_MAX_BITS (C macro), 18
ZFP_HEADER_META (C macro), 18
ZFP_HEADER_MODE (C macro), 18
ZFP_INT64 (C macro), 8
ZFP_INT64_SUFFIX (C macro), 8
zfp_library_version (C variable), 19
ZFP_MAGIC_BITS (C macro), 18
ZFP_MAX_BITS (C macro), 18
ZFP_MAX_PREC (C macro), 18
ZFP_META_BITS (C macro), 18
ZFP_MIN_BITS (C macro), 18
ZFP_MIN_EXP (C macro), 18
ZFP_MODE_LONG_BITS (C macro), 18
ZFP_MODE_SHORT_BITS (C macro), 18
ZFP_MODE_SHORT_MAX (C macro), 18
zfp_promote_int16_to_int32 (C function), 27
zfp_promote_int8_to_int32 (C function), 27
zfp_promote_uint16_to_int32 (C function), 27
zfp_promote_uint8_to_int32 (C function), 27
zfp_read_header (C function), 22
zfp_stream (C type), 18
zfp_stream.maxbits (C member), 13
zfp_stream.maxprec (C member), 13
zfp_stream.minbits (C member), 13
zfp_stream.minexp (C member), 14
zfp_stream_align (C function), 23
zfp_stream_bit_stream (C function), 20
zfp_stream_close (C function), 20
zfp_stream_compressed_size (C function), 20
zfp_stream_flush (C function), 23
zfp_stream_maximum_size (C function), 20
zfp_stream_mode (C function), 20
zfp_stream_open (C function), 20
zfp_stream_params (C function), 20
zfp_stream_rewind (C function), 20
zfp_stream_set_accuracy (C function), 21
zfp_stream_set_bit_stream (C function), 20
zfp_stream_set_mode (C function), 21
zfp_stream_set_params (C function), 21
zfp_stream_set_precision (C function), 21
zfp_stream_set_rate (C function), 20
zfp_type (C type), 19
zfp_type_size (C function), 20
ZFP_UINT64 (C macro), 8
ZFP_UINT64_SUFFIX (C macro), 8
ZFP_VERSION (C macro), 17
ZFP_VERSION_MAJOR (C macro), 17
ZFP_VERSION_MINOR (C macro), 17
ZFP_VERSION_PATCH (C macro), 17
ZFP_VERSION_STRING (C macro), 17
zfp_version_string (C variable), 19
ZFP_WITH_ALIGNED_ALLOC (C macro), 8
ZFP_WITH_CACHE_FAST_HASH (C macro), 9
ZFP_WITH_CACHE_PROFILE (C macro), 9
ZFP_WITH_CACHE_TWOWAY (C macro), 8
zfp_write_header (C function), 22