zfp Documentation

Release 1.0.1

Peter Lindstrom
Danielle Asher

Dec 15, 2023

CONTENTS

CHAPTER

ONE

INTRODUCTION

zfp is an open-source library for representing multidimensional numerical arrays in compressed form to reduce storage and bandwidth requirements. zfp consists of four main components:

- An **efficient number format** for representing small, fixed-size *blocks* of real values. The zfp format usually provides higher accuracy per bit stored than conventional number formats like IEEE 754 floating point.
- A set of *classes* that implement storage and manipulation of a **multidimensional array data type**. zfp arrays support high-speed read and write random access to individual array elements and are a drop-in replacement for `std::vector` and native C/C++ arrays. zfp arrays provide accessors like *proxy pointers*, *iterators*, and *views*. zfp arrays allow specifying an exact memory footprint or an error tolerance.
- A *C library* for **streaming compression** of partial or whole arrays of integers or floating-point numbers, e.g., for applications that read and write large data sets to and from disk. This library supports fast, parallel (de)compression via OpenMP and CUDA.
- A **command-line executable** for *compressing binary files* of integer or floating-point arrays, e.g., as a substitute for general-purpose compressors like `gzip`.

As a compressor, zfp 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. Fully *lossless compression*, where values are represented exactly, is also supported.

zfp is primarily written in C and C++ but also includes *Python* and *Fortran* bindings. zfp is being developed at Lawrence Livermore National Laboratory and is supported by the U.S. Department of Energy's Exascale Computing Project. zfp is a 2023 R&D 100 Award Winner.

1.1 Availability

zfp is freely available as open source on GitHub and is distributed under the terms of a permissive three-clause *BSD license*. zfp may be *installed* using CMake or GNU Make. Installation from source code is recommended for users who wish to configure the internals of zfp and select which components (e.g., programming models, language bindings) to install.

zfp is also available through several package managers, including Conda (both C/C++ and Python packages are available), PIP, Spack, and MacPorts. Linux packages are available for several distributions and may be installed, for example, using `apt` and `yum`.

1.2 Application Support

zfp has been incorporated into several independently developed applications, plugins, and formats, such as

- Compressed file I/O in ADIOS.
- Compression codec in the BLOSC meta compressor.
- H5Z-ZFP plugin for HDF5®. zfp is also one of the select compressors shipped with HDF5 binaries.
- Compression functions for Intel® Integrated Performance Primitives.
- Compressed MPI messages in MVAPICH2-GDR.
- Compressed file I/O in OpenInventor™.
- Compression codec underlying the OpenZGY format.
- Compressed file I/O in TTK.
- Third-party module in VTK.
- Compression worklet in VTK-m.
- Compression codec in Zarr via numcodecs.

See this list for other software products that support zfp.

1.3 Usage

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

- Via the *C API* when doing I/O in an application or otherwise performing data (de)compression online. High-speed, parallel compression is supported via OpenMP and CUDA.
- Via zfp's 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. These classes can often substitute C/C++ arrays and STL vectors in applications with minimal code changes.
- Via the zfp *command-line tool* when compressing binary files offline.
- Via *third-party* I/O libraries or tools that support zfp.

1.4 Technology

zfp compresses d-dimensional (1D, 2D, 3D, and 4D) arrays of integer or floating-point values by partitioning the array into cubical blocks of 4^d values, i.e., 4, 16, 64, or 256 values for 1D, 2D, 3D, and 4D arrays, respectively. Each such block is independently compressed to a fixed- or variable-length bit string, and these bit strings may be concatenated into a single stream of bits.

zfp usually truncates each per-block 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. The trailing bits can usually be discarded (zeroed) with limited impact on accuracy.

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 generally uses lossy but optionally error-bounded compression. Bit-for-bit lossless compression is also possible through one of zfp's *compression modes*.

zfp works best for 2D-4D arrays that exhibit spatial correlation, such as continuous fields from physics simulations, images, regularly sampled terrain surfaces, etc. Although zfp also provides support for 1D arrays, e.g., for audio signals or even unstructured floating-point streams, the compression scheme has not been well optimized for this use case, and compression ratio and quality may not be competitive with floating-point compressors designed specifically for 1D streams.

In all use 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.

1.5 Resources

zfp is 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 has evolved since the original publication; the algorithm implemented in the current version is described in:

> James Diffenderfer, Alyson Fox, Jeffrey Hittinger, Geoffrey Sanders, Peter Lindstrom
> "Error Analysis of ZFP Compression for Floating-Point Data"
> SIAM Journal on Scientific Computing
> 41(3):A1867-A1898, 2019
> doi:10.1137/18M1168832

For more information on zfp, please see the zfp website. For bug reports, please consult the GitHub issue tracker. For questions, comments, and requests, please contact us.

CHAPTER TWO

LICENSE

2.1 Notice

CHAPTER

THREE

INSTALLATION

zfp consists of four distinct parts: a compression library written in C, a set of C++ header files that implement compressed arrays and corresponding C wrappers, optional Python and Fortran bindings, and a set of C and C++ examples and utilities. 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`.

zfp is preferably built using CMake, although the core library can also be built using GNU make on Linux, macOS, and MinGW.

zfp conforms to various language standards, including C89, C99, C++98, C++11, and C++14.

Note: zfp requires compiler support for 64-bit integers.

3.1 CMake Builds

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

```
cd zfp-1.0.1
mkdir build
cd build
cmake ..
make
```

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

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

By default, CMake builds will attempt to locate and use OpenMP. To disable OpenMP, type:

```
cmake -DZFP_WITH_OPENMP=OFF ..
```

To build zfp using Visual Studio on Windows, start a DOS shell and type:

```
cd zfp-1.0.1
mkdir build
cd build
cmake ..
cmake --build . --config Release
```

This builds zfp in release mode. Replace 'Release' with 'Debug' to build zfp in debug mode. See the instructions for Linux on how to change the cmake line to also build the example programs.

3.2 GNU Builds

To build zfp using gcc without OpenMP, type:

```
cd zfp-1.0.1
gmake
```

This builds `libzfp` as a static library as well as the zfp command-line utility. To enable OpenMP parallel compression, type:

```
gmake ZFP_WITH_OPENMP=1
```

Note: GNU builds expose only limited functionality of zfp. For instance, CUDA and Python support are not included. For full functionality, build zfp using CMake.

3.3 Testing

To test that zfp is working properly, type:

```
ctest
```

or using GNU make:

```
gmake test
```

If the GNU build 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, which will be indicated by checksum errors. 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.

3.4 Build Targets

To specify which components to build, set the macros below to `ON` (CMake) or `1` (GNU make), e.g.,

```
cmake -DBUILD_UTILITIES=OFF -DBUILD_EXAMPLES=ON ..
```

or using GNU make

```
gmake BUILD_UTILITIES=0 BUILD_EXAMPLES=1
```

Regardless of the settings below, `libzfp` will always be built.

`BUILD_ALL`

Build all subdirectories; enable all options (except *`BUILD_SHARED_LIBS`*). Default: off.

BUILD_CFP

Build `libcfp` for C bindings to the compressed-array classes. Default: off.

BUILD_ZFPY

Build zfPy for Python bindings to the C API.

CMake will attempt to automatically detect the Python installation to use. If CMake finds multiple Python installations, it will use the newest one. To specify a specific Python installation to use, set *PYTHON_LIBRARY* and *PYTHON_INCLUDE_DIR* on the cmake line:

```
cmake -DBUILD_ZFPY=ON -DPYTHON_LIBRARY=/path/to/lib/libpython2.7.so -DPYTHON_
↪INCLUDE_DIR=/path/to/include/python2.7 ..
```

CMake default: off. GNU make default: off and ignored.

BUILD_ZFORP

Build `libzFORp` for Fortran bindings to the C API. Requires Fortran standard 2018 or later. GNU make users may specify the Fortran compiler to use via

```
gmake BUILD_ZFORP=1 FC=/path/to/fortran-compiler
```

Default: off.

BUILD_UTILITIES

Build **zfp** command-line utility for compressing binary files. Default: on.

BUILD_EXAMPLES

Build code examples. Default: off.

BUILD_TESTING

Build **testzfp** tests. Default: on.

BUILD_TESTING_FULL

Build all unit tests. Default: off.

BUILD_SHARED_LIBS

Build shared objects (`.so`, `.dylib`, or `.dll` files). CMake default: on. GNU make default: off.

Note: On macOS, add `OS=mac` when building shared libraries with GNU make.

3.5 Configuration

The behavior of zfp can be configured at compile time via a set of macros in the same manner that *build targets* are specified, e.g.,

```
cmake -DZFP_WITH_OPENMP=OFF ..
```

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_OPENMP

CMake and GNU make macro for enabling or disabling OpenMP support. CMake builds will by default enable OpenMP when available. Set this macro to 0 or OFF to disable OpenMP support. For GNU builds, OpenMP is disabled by default. Set this macro to 1 or ON to enable OpenMP support. See also OMPFLAGS in `Config` in case the compiler does not recognize `-fopenmp`. For example, Apple clang requires `OMPFLAGS=-Xclang -fopenmp`, `LDFLAGS=-lomp`, and an installation of `libomp`. CMake default: on. GNU make default: off.

ZFP_WITH_CUDA

CMake macro for enabling or disabling CUDA support for GPU compression and decompression. When enabled, CUDA and a compatible host compiler must be installed. For a full list of compatible compilers, please consult the NVIDIA documentation. If a CUDA installation is in the user's path, it will be automatically found by CMake. Alternatively, the CUDA binary directory can be specified using the `CUDA_BIN_DIR` environment variable. CMake default: off. GNU make default: off and ignored.

ZFP_ROUNDING_MODE

Experimental feature. By default, zfp coefficients are truncated, not rounded, which can result in biased errors (see FAQ *#30*). To counter this, two rounding modes are available: `ZFP_ROUND_FIRST` (round during compression; analogous to mid-tread quantization) and `ZFP_ROUND_LAST` (round during decompression; analogous to mid-riser quantization). With `ZFP_ROUND_LAST`, the values returned on decompression are slightly modified (and usually closer to the original values) without impacting the compressed data itself. This rounding mode works with all *compression modes*. With `ZFP_ROUND_FIRST`, the values are modified before compression, thus impacting the compressed stream. This rounding mode tends to be more effective at reducing bias, but is invoked only with *fixed-precision* and *fixed-accuracy* compression modes. Both of these rounding modes break the regression tests since they alter the compressed or decompressed representation, but they may be used with libraries built with the default rounding mode, `ZFP_ROUND_NEVER`, and versions of zfp that do not support a rounding mode with no adverse effects. Note: *ZFP_ROUNDING_MODE* is currently supported only by the `serial` and `omp` *execution policies*. Default: `ZFP_ROUND_NEVER`.

ZFP_WITH_TIGHT_ERROR

Experimental feature. When enabled, this feature takes advantage of the error reduction associated with proper rounding; see *ZFP_ROUNDING_MODE*. The reduced error due to rounding allows the tolerance in *fixed-accuracy mode* to be satisfied using fewer bits of compressed data. As a result, when enabled, the observed maximum absolute error is closer to the tolerance and the compression ratio is increased. This feature requires the rounding mode to be `ZFP_ROUND_FIRST` or `ZFP_ROUND_LAST` and is supported only by the `serial` and `omp` *execution policies*. Default: undefined/off.

ZFP_WITH_DAZ

When enabled, blocks consisting solely of subnormal floating-point numbers (tiny numbers close to zero) are treated as blocks of all zeros (DAZ = denormals-are-zero). The main purpose of this option is to avoid the potential for floating-point overflow in the zfp implementation that may occur in step 2 of the *lossy compression algorithm* when converting to zfp's block-floating-point representation (see Issue #119). Such overflow tends to be benign but loses all precision and usually results in "random" subnormals upon decompression. When enabled, compressed streams may differ slightly but are decompressed correctly by libraries built without this option. This option may break some regression tests. Note: *ZFP_WITH_DAZ* is currently ignored by all *execution policies* other than `serial` and `omp`. Default: undefined/off.

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.

CFP_NAMESPACE

Macro for renaming the outermost cfp namespace, e.g., to avoid name clashes. Default: `cfp`.

PYTHON_LIBRARY

Path to the Python library, e.g., `/usr/lib/libpython2.7.so`. CMake default: undefined/off. GNU make default: off and ignored.

PYTHON_INCLUDE_DIR

Path to the Python include directory, e.g., `/usr/include/python2.7`. CMake default: undefined/off. GNU make default: off and ignored.

3.6 Dependencies

The core zfp library and compressed arrays require only a C89 and C++98 compiler. The optional components have additional dependencies, as outlined in the sections below.

3.6.1 CMake

CMake builds require version 3.9 or later. CMake is available here.

3.6.2 OpenMP

OpenMP support requires OpenMP 2.0 or later.

3.6.3 CUDA

CUDA support requires CUDA 7.0 or later, CMake, and a compatible host compiler (see *ZFP_WITH_CUDA*).

3.6.4 C/C++

The zfp C library and cfp C wrappers around the compressed-array classes conform to the C90 standard (ISO/IEC 9899:1990). The C++ classes conform to the C++98 standard (ISO/IEC 14882:1998).

3.6.5 Python

The optional Python bindings require CMake and the following minimum versions:

- Python: Python 2.7 & Python 3.5
- Cython: 0.22
- NumPy: 1.8.0

The necessary dependencies can be installed using `pip` and the zfp `requirements.txt`:

```
pip install -r $ZFP_ROOT/python/requirements.txt
```

3.6.6 Fortran

The optional Fortran bindings require a Fortran 2018 compiler.

CHAPTER

FOUR

ALGORITHM

zfp uses two different algorithms to support *lossy* and *lossless* compression. These algorithms are described in detail below.

4.1 Lossy Compression

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 lossy 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 single dedicated sign bit, these integers are subsequently treated as unsigned. Unlike sign-magnitude representations, the leftmost one-bit in negabinary simultaneously encodes the sign and approximate magnitude of a number. Moreover, unlike

two's complement, numbers small in magnitude have many leading zeros in negabinary regardless of sign, which facilitates encoding.

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 is the smallest number such that the $4^d - n$ highest bits in all previous bit planes are all zero. Initially, $n = 0$. 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.

   As an example, x = 000001001101000 with m = 10 is encoded as **0**100**1**1**1**10**1**1000**1**, where the bits in boldface indicate "group tests" that determine if the remainder of x (to the left) contains any one-bits. Again, this variable-length code is generated and parsed from right to left.

8. The embedded coder emits one bit at a time, with each successive bit potentially improving the accuracy of the approximation. The early bits are most important and have the greatest impact on accuracy, 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 block of values. 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*.

4.2 Lossless Compression

The reversible (lossless) compression algorithm shares most steps with the lossy algorithm. The main differences are steps 2, 3, and 8, which are the only sources of error. Since step 2 may introduce loss in the conversion to zfp's block-floating-point representation, the reversible algorithm adds a test to see if this conversion is lossless. It does so by converting the values back to the source format and testing the result for bitwise equality with the uncompressed data. If this test passes, then a modified decorrelating transform is performed in step 3 that uses reversible integer subtraction operations only. Finally, step 8 is modified so that no one-bits are truncated in the variable-length bit stream. However, all least significant bit planes with all-zero bits are truncated, and the number of encoded bit planes is recorded in step 7. As with lossy compression, a floating-point block consisting of all ("positive") zeros is represented as a single bit, making it possible to efficiently encode sparse data.

If the block-floating-point transform is not lossless, then the reversible compression algorithm falls back on a simpler scheme that reinterprets floating-point values as integers via *type punning*. This lossless conversion from floating-point to integer data replaces step 2, and the algorithm proceeds from there with the modified step 3. Moreover, this conversion ensures that special values like infinities, NaNs, and negative zero are preserved.

The lossless algorithm handles integer data also, for which step 2 is omitted.

CHAPTER

FIVE

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 five different compression modes that are mutually exclusive: *expert*, *fixed-rate*, *fixed-precision*, *fixed-accuracy*, and *reversible* 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. One can verify the active compression mode on a `zfp_stream` through `zfp_stream_compression_mode()`. The members that govern the compression parameters are described below.

5.1 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 in the *compression algorithm*. 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 equals one bit, 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 (applies to floating-point data only; this parameter is ignored for integer data). 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).

Warning: For floating-point data, the *zfp_stream.maxbits* parameter must be large enough to allow the common block exponent and any control bits to be encoded. This implies $maxbits \geq 9$ for single-precision data and $maxbits \geq 12$ for double-precision data. Choosing a smaller value is of no use as it would prevent any fraction (value) bits from being encoded, resulting in an all-zero decompressed block. More importantly, such a constraint will not be respected by zfp for performance reasons, which if not accounted for could potentially lead to buffer overruns.

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.

5.2 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.

Note: Use fixed-rate mode only if you have to bound the compressed size or need read and write random access to blocks.

5.3 Fixed-Precision Mode

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.

5.4 Fixed-Accuracy Mode

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 and 4D 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 (see *Reversible Mode* for guaranteed 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.

Note: Fixed-accuracy mode is available for floating-point (not integer) data only.

5.5 Reversible Mode

As of zfp 0.5.5, reversible (lossless) compression is supported. As with the other compression modes, each block is compressed and decompressed independently, but reversible mode uses a different compression algorithm that ensures a bit-for-bit identical reconstruction of integer and floating-point data. For IEEE-754 floating-point data, reversible mode preserves special values such as subnormals, infinities, NaNs, and positive and negative zero.

The expert mode parameters corresponding to reversible mode are:

```
minbits = ZFP_MIN_BITS
maxbits = ZFP_MAX_BITS
maxprec = ZFP_MAX_PREC
minexp < ZFP_MIN_EXP
```

Reversible mode is enabled via `zfp_stream_set_reversible()` and through the *-R* command-line option in the *zfp executable*. It is supported by both the low- and high-level interfaces and by the serial and OpenMP execution policies, but it is not yet implemented in CUDA.

CHAPTER

SIX

PARALLEL EXECUTION

As of zfp 0.5.3, parallel compression (but not decompression) is supported on multicore processors via OpenMP threads. zfp 0.5.4 adds CUDA support for fixed-rate compression and decompression on the GPU.

Since zfp partitions arrays into small independent blocks, a large amount of data parallelism is inherent in the compression scheme that can be exploited. In principle, concurrency is limited only by the number of blocks that make up an array, though in practice each thread is responsible for compressing a *chunk* of several contiguous blocks.

Note: zfp parallel compression is confined to shared memory on a single compute node or GPU. No effort is made to coordinate compression across distributed memory on networked compute nodes, although zfp's fine-grained partitioning of arrays should facilitate distributed parallel compression.

This section describes the zfp parallel compression algorithm and explains how to configure `libzfp` and enable parallel compression at run time via its *high-level C API*.

Note: Parallel compression is not supported via the *low-level API*, which ignores all execution policy settings and always executes in serial.

6.1 Execution Policies

zfp supports multiple *execution policies*, which dictate how (e.g., sequentially, in parallel) and where (e.g., on the CPU or GPU) arrays are compressed. Currently three execution policies are available: `serial`, `omp`, and `cuda`. The default mode is `serial`, which ensures sequential compression on a single thread. The `omp` and `cuda` execution policies allow for data-parallel compression on multiple threads.

The execution policy is set by `zfp_stream_set_execution()` and pertains to a particular `zfp_stream`. Hence, each stream (and array) may use a policy suitable for that stream. For instance, very small arrays are likely best compressed in serial, while parallel compression is best reserved for very large arrays that can take the most advantage of concurrent execution.

As outlined in FAQ *#23*, the final compressed stream is independent of execution policy.

6.2 Execution Parameters

Each execution policy allows tailoring the execution via its associated *execution parameters*. Examples include number of threads, chunk size, scheduling, etc. The `serial` and `cuda` policies have no parameters. The subsections below discuss the `omp` parameters.

Whenever the execution policy is changed via *`zfp_stream_set_execution()`*, its parameters (if any) are initialized to their defaults, overwriting any prior setting.

6.2.1 OpenMP Thread Count

By default, the number of threads to use is given by the current setting of the OpenMP internal control variable *nthreads-var*. Unless the calling thread has explicitly requested a thread count via the OpenMP API, this control variable usually defaults to the number of threads supported by the hardware (e.g., the number of available cores).

To set the number of requested threads to be used by zfp, which may differ from the thread count of encapsulating or surrounding OpenMP parallel regions, call *`zfp_stream_set_omp_threads()`*.

The user is advised to call the zfp API functions to modify OpenMP behavior rather than make direct OpenMP calls. For instance, use *`zfp_stream_set_omp_threads()`* rather than `omp_set_num_threads()`. To indicate that the current OpenMP settings should be used, for instance as determined by the global OpenMP environment variable `OMP_NUM_THREADS`, pass a thread count of zero (the default setting) to *`zfp_stream_set_omp_threads()`*.

Note that zfp does not modify *nthreads-var* or other control variables but uses a `num_threads` clause on the OpenMP `#pragma` line. Hence, no OpenMP state is changed and any subsequent OpenMP code is not impacted by zfp's parallel compression.

6.2.2 OpenMP Chunk Size

The d-dimensional array is partitioned into *chunks*, with each chunk representing a contiguous sequence of *blocks* of 4^d array elements each. Chunks represent the unit of parallel work assigned to a thread. By default, the array is partitioned so that each thread processes one chunk. However, the user may override this behavior by setting the chunk size (in number of zfp blocks) via *`zfp_stream_set_omp_chunk_size()`*. See FAQ *#25* for a discussion of chunk sizes and parallel performance.

6.2.3 OpenMP Scheduling

zfp does not specify how to schedule chunk processing. The schedule used is given by the OpenMP *def-sched-var* internal control variable. If load balance is poor, it may be improved by using smaller chunks, which may or may not impact performance depending on the OpenMP schedule in use. Future versions of zfp may allow specifying how threads are mapped to chunks, whether to use static or dynamic scheduling, etc.

6.3 Fixed- vs. Variable-Rate Compression

Following partitioning into chunks, zfp assigns each chunk to a thread. If there are more chunks than threads supported, chunks are processed in unspecified order.

In *variable-rate mode*, there is no way to predict the exact number of bits that each chunk compresses to. Therefore, zfp allocates a temporary memory buffer for each chunk. Once all chunks have been compressed, they are concatenated into a single bit stream in serial, after which the temporary buffers are deallocated.

In *fixed-rate mode*, the final location of each chunk's bit stream is known ahead of time, and zfp may not have to allocate temporary buffers. However, if the chunks are not aligned on *word boundaries*, then race conditions may occur. In other words, for chunk size C, rate R, and word size W, the rate and chunk size must be such that $C \times 4^{\mathrm{d}} \times R$ is a multiple of W to avoid temporary buffers. Since W is a small power of two no larger than 64, this is usually an easy requirement to satisfy.

When chunks are whole multiples of the word size, no temporary buffers are allocated and the threads write compressed data directly to the target buffer. The CUDA implementation uses atomics to avoid race conditions, and therefore does not need temporary buffers, regardless of chunk alignment.

6.4 Using OpenMP

In order to use OpenMP compression, zfp must be built with OpenMP support. If built with CMake, OpenMP support is automatically enabled when available. To manually disable OpenMP support, see the *ZFP_WITH_OPENMP* macro.

To avoid compilation errors on systems with spotty OpenMP support (e.g., macOS), OpenMP is by default disabled in GNU builds. To enable OpenMP, see *GNU Builds* and the *ZFP_WITH_OPENMP* macro.

6.5 Using CUDA

CUDA support is by default disabled. Enabling it requires an installation of CUDA and a compatible host compiler. Furthermore, the *ZFP_WITH_CUDA* macro must be set and zfp must be built with CMake. See *ZFP_WITH_CUDA* for further details.

6.5.1 Device Memory Management

The CUDA version of zfp supports both host and device memory. If device memory is allocated for fields or compressed streams, this is automatically detected and handled in a consistent manner. For example, with compression, if host memory pointers are provided for both the field and compressed stream, then device memory will transparently be allocated and the uncompressed data will be copied to the GPU. Once compression completes, the compressed stream is copied back to the host and device memory is deallocated. If both pointers are device pointers, then no copies are made. Additionally, any combination of mixing host and device pointers is supported.

6.5.2 CUDA Limitations

The CUDA implementation has a number of limitations:

- Only the *fixed-rate mode* mode is supported. Other modes will be supported in a future release.
- 4D arrays are not supported.
- *Headers* are not supported. Any header already present in the stream will be silently overwritten on compression.
- zfp must be built with a `ZFP_BIT_STREAM_WORD_SIZE` of 64 bits.
- Although *strides* are supported, fields must be contiguous when stored in host memory, i.e., with no unused memory addresses between the minimum and maximum address spanned by the field (see `zfp_field_is_contiguous()`). This requirement avoids having to copy and allocate more temporary memory than needed to hold the array if it were not strided. Note that the strides can still be arbitrary as long as they serve only to permute the array elements. Moreover, this restriction applies only to the CUDA execution policy and the case where the uncompressed field resides on the host.

We expect to address these limitations over time.

6.6 Setting the Execution Policy

Enabling parallel compression at run time is often as simple as calling `zfp_stream_set_execution()`

```
if (zfp_stream_set_execution(stream, zfp_exec_omp)) {
  // use OpenMP parallel compression
  ...
  zfpsize = zfp_compress(stream, field);
}
```

before calling `zfp_compress()`. Replacing `zfp_exec_omp` with `zfp_exec_cuda` enables CUDA execution. If OpenMP or CUDA is disabled or not supported, then the return value of functions setting these execution policies and parameters will indicate failure. Execution parameters are optional and may be set using the functions discussed above.

The source code for the **zfp** command-line tool includes further examples on how to set the execution policy. To use parallel compression and decompression in this tool, see the `-x` command-line option.

Note: As of zfp 0.5.4, the execution policy refers to both compression and decompression. The OpenMP implementation does not yet support decompression, and hence `zfp_decompress()` will fail if the execution policy is not reset to `zfp_exec_serial` before calling the decompressor. Similarly, the CUDA implementation supports only fixed-rate mode and will fail if other compression modes are specified.

The following table summarizes which execution policies are supported with which *compression modes*:

(de)compression mode		serial	OpenMP	CUDA
compression	fixed rate	✓	✓	✓
	fixed precision	✓	✓	
	fixed accuracy	✓	✓	
	reversible	✓	✓	
decompression	fixed rate	✓		✓
	fixed precision	✓		
	fixed accuracy	✓		
	reversible	✓		

`zfp_compress()` and `zfp_decompress()` both return zero if the current execution policy is not supported for the requested compression mode.

6.7 Parallel Compression

Once the execution policy and parameters have been selected, compression is executed by calling `zfp_compress()` from a single thread. This function in turn inspects the execution policy given by the `zfp_stream` argument and dispatches the appropriate function for executing compression.

6.8 Parallel Decompression

Parallel decompression is in principle possible using the same strategy as used for compression. However, in zfp's *variable-rate modes*, the compressed blocks do not occupy fixed storage, and therefore the decompressor needs to be instructed where each compressed block resides in the bit stream to enable parallel decompression. Because the zfp bit stream does not currently store such information, variable-rate parallel decompression is not yet supported, though plans are to make such functionality available in the near future.

The CUDA implementation supports fixed-rate decompression. OpenMP fixed-rate decompression has been implemented and will be released in the near future.

Future versions of zfp will allow efficient encoding of block sizes and/or offsets to allow each thread to quickly locate the blocks it is responsible for decompressing, which will allow for variable-rate compression and decompression. Such capabilities are already present in the implementation of the zfp *read-only arrays*.

CHAPTER

SEVEN

HIGH-LEVEL C API

The `libzfp` C API provides functionality for sequentially compressing and decompressing whole integer and floating-point arrays or single blocks. It is broken down into a *high-level API* and a *low-level API*. The high-level API handles compression of entire arrays and supports a variety of back-ends (e.g., serial, OpenMP). The low-level API exists for processing individual, possibly partial blocks as well as reduced-precision integer data less than 32 bits wide. Both C APIs are declared in `zfp.h`.

The following sections are available:

7.1 Macros

`ZFP_VERSION_MAJOR`

`ZFP_VERSION_MINOR`

`ZFP_VERSION_PATCH`

`ZFP_VERSION_TWEAK`

Macros identifying the zfp library version (*major.minor.patch.tweak*). *`ZFP_VERSION_TWEAK`* is new as of zfp 1.0.0 and is used to mark intermediate develop versions (unofficial releases).

ZFP_VERSION_DEVELOP

Macro signifying that the current version is an intermediate version that differs from the last official release. This macro is undefined for official releases; when defined, its value equals 1. Note that this macro may be defined even if the four *version identifiers* have not changed. Available as of zfp 1.0.0.

ZFP_VERSION

A single integer constructed from the four *version identifiers*. This integer can be generated by *ZFP_MAKE_VERSION* or *ZFP_MAKE_FULLVERSION*. Its value equals the global constant *zfp_library_version*.

Note: Although *ZFP_VERSION* increases monotonically with release date and with the four *version identifiers* it depends on, the mapping to *ZFP_VERSION* changed with the introduction of *ZFP_VERSION_TWEAK* in zfp 1.0.0.

Going forward, we recommend using *ZFP_MAKE_VERSION* or *ZFP_MAKE_FULLVERSION* in conditional code that depends on *ZFP_VERSION*, e.g., `#if ZFP_VERSION >= ZFP_MAKE_VERSION(1, 0, 0)`. Note that such constructions should not be used with older versions of zfp, e.g., `if (zfp_library_version == ZFP_MAKE_VERSION(0, 5, 5))` will not give the expected result with binary versions of `libzfp` before version 1.0.0.

ZFP_VERSION_STRING

ZFP_VERSION_STRING is a string literal composed of the four *version identifiers*. It is a component of *zfp_version_string*.

ZFP_MAKE_VERSION(major, minor, patch)

ZFP_MAKE_VERSION_STRING(major, minor, patch)

Utility macros for constructing *ZFP_VERSION* and *ZFP_VERSION_STRING*, respectively. Available as of zfp 1.0.0, these macros may be used by applications to test for a certain zfp version number, e.g., `#if ZFP_VERSION >= ZFP_MAKE_VERSION(1, 0, 0)`.

ZFP_MAKE_FULLVERSION(major, minor, patch, tweak)

ZFP_MAKE_FULLVERSION_STRING(major, minor, patch, tweak)

Utility macros for constructing *ZFP_VERSION* and *ZFP_VERSION_STRING*, respectively. Includes tweak version used by intermediate develop versions. Available as of zfp 1.0.0, these macros may be used by applications to test for a certain zfp version number, e.g., `#if ZFP_VERSION >= ZFP_MAKE_FULLVERSION(1, 0, 0, 2)`.

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 4D blocks of doubles, is given by *ZFP_MAX_BITS*. See also *zfp_stream*.

ZFP_META_NULL

Null representation of the 52-bit encoding of field metadata. This value is returned by *zfp_field_metadata()* when the field metadata cannot be encoded in 64 bits, such as when the array dimensions are too large (see *Limitations*). In addition to signaling error, this value is guaranteed not to represent valid metadata.

The ZFP_HEADER bit mask specifies which portions of a header to output (if any). The constants below 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. See *zfp_read_header()* and *zfp_write_header()* for how to read and write header information.

ZFP_HEADER_MAGIC

Magic constant that identifies the data as a zfp stream compressed using a particular CODEC version.

ZFP_HEADER_META

Array size and scalar type information stored in the *zfp_field* struct.

ZFP_HEADER_MODE

Compression mode and parameters stored in the *zfp_stream* struct.

ZFP_HEADER_FULL

Full header information (bitwise OR of all ZFP_HEADER constants).

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()*).

The ZFP_DATA bit mask specifies which portions of array data structures to compute total storage size for. These constants should be bitwise ORed together. Use *ZFP_DATA_ALL* to count all storage used.

ZFP_DATA_UNUSED

Allocated but unused data.

ZFP_DATA_PADDING

Padding for alignment purposes.

ZFP_DATA_META

Class members and other fixed-size storage.

ZFP_DATA_MISC

Miscellaneous uncategorized storage.

ZFP_DATA_PAYLOAD

Compressed data encoding array elements.

ZFP_DATA_INDEX

Block *index* information.

ZFP_DATA_CACHE

Uncompressed *cached* data.

ZFP_DATA_HEADER

Header information.

ZFP_DATA_ALL

All storage (bitwise OR of all ZFP_DATA constants).

ZFP_ROUND_FIRST

ZFP_ROUND_NEVER

ZFP_ROUND_LAST

Available rounding modes for *ZFP_ROUNDING_MODE*, which specifies at build time how zfp performs rounding in lossy compression mode.

7.2 Types

type **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_execution exec; // execution policy and parameters
} zfp_stream;
```

---

type **zfp_execution**

The *zfp_stream* also stores information about how to execute compression, e.g., sequentially or in parallel. The execution is determined by the policy and any policy-specific parameters such as number of threads.

```
typedef struct {
  zfp_exec_policy policy; // execution policy (serial, omp, cuda, ...)
  void* params;           // execution parameters
} zfp_execution;
```

Warning: As of zfp 1.0.0 *zfp_execution* replaces the former zfp_exec_params with a void* to the associated zfp_exec_params type (e.g., *zfp_exec_params_omp*) to limit ABI-breaking changes due to future extensions to zfp execution policies.

type **zfp_exec_policy**

Currently three execution policies are available: serial, OpenMP parallel, and CUDA parallel.

```
typedef enum {
  zfp_exec_serial = 0, // serial execution (default)
  zfp_exec_omp    = 1, // OpenMP multi-threaded execution
  zfp_exec_cuda   = 2  // CUDA parallel execution
} zfp_exec_policy;
```

type **zfp_exec_params_omp**

Execution parameters for OpenMP parallel compression. These are initialized to default values. When nonzero, they indicate the number of threads to request for parallel compression and the number of consecutive blocks to assign to each thread.

```
typedef struct {
  uint threads;    // number of requested threads
  uint chunk_size; // number of blocks per chunk
} zfp_exec_params_omp;
```

type **zfp_mode**

Enumerates the compression modes.

```
typedef enum {
  zfp_mode_null            = 0, // an invalid configuration of the 4 params
  zfp_mode_expert          = 1, // expert mode (4 params set manually)
  zfp_mode_fixed_rate      = 2, // fixed rate mode
  zfp_mode_fixed_precision = 3, // fixed precision mode
  zfp_mode_fixed_accuracy  = 4, // fixed accuracy mode
  zfp_mode_reversible      = 5  // reversible (lossless) mode
} zfp_mode;
```

type **zfp_config**

Encapsulates compression mode and parameters (if any).

```
typedef struct {
  zfp_mode mode;      // compression mode */
  union {
    double rate;      // compressed bits/value (negative for word alignment)
    uint precision;   // uncompressed bits/value
    double tolerance; // absolute error tolerance
    struct {
      uint minbits;   // min number of compressed bits/block
      uint maxbits;   // max number of compressed bits/block
      uint maxprec;   // max number of uncompressed bits/value
      int minexp;     // min floating point bit plane number to store
    } expert;         // expert mode arguments
  } arg;              // arguments corresponding to compression mode
} zfp_config;
```

type **zfp_type**

Enumerates the scalar types supported by the compressor and describes 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;
```

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)
  size_t nx, ny, nz, nw;      // sizes (zero for unused dimensions)
  ptrdiff_t sx, sy, sz, sw; // strides (zero for contiguous array a[nw][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][n4];
```

would be described by a *zfp_field* with members

```
type = zfp_type_double;
nx = n4; ny = n3; nz = n2; nw = n1;
sx = 1; sy = n4; sz = n3 * n4; sw = n2 * n3 * n4;
data = &array[0][0][0][0];
```

The strides, when nonzero, specify how the array is laid out in memory. Strides can be used in case multiple fields are stored interleaved via "array of struct" (AoS) rather than "struct of array" (SoA) storage, or if the dimensions should be transposed during (de)compression. Strides may even be negative, allowing one or more dimensions to be traversed in reverse order. Given 4D array indices (*x*, *y*, *z*, *w*), the corresponding array element is stored at

```
data[x * sx + y * sy + z * sz + w * sw]
```

where `data` is a pointer to the first array element.

Warning: The *zfp_field* struct was modified in zfp 1.0.0 to use `size_t` and `ptrdiff_t` for array dimensions and strides, respectively, to support 64-bit addressing of very large arrays (previously, `uint` and `int` were used). This ABI incompatible change may require rebuilding applications that use zfp and may in some cases also require code changes to handle pointers to `size_t` instead of pointers to `uint` (see *zfp_field_size()*, for instance).

Warning: It is paramount that the field dimensions, *nx*, *ny*, *nz*, and *nw*, and strides, *sx*, *sy*, *sz*, and *sw*, be correctly mapped to how the uncompressed array is laid out in memory. Although compression will still succeed if array dimensions are accidentally transposed, compression ratio and/or accuracy may suffer greatly. Since the leftmost index, *x*, is assumed to vary fastest, zfp can be thought of as assuming Fortran ordering. For C ordered arrays, the user should transpose the dimensions or specify strides to properly describe the memory layout. See *this FAQ* for further details.

type **zfp_bool**

zfp_bool is new as of zfp 1.0.0. Although merely an alias for `int`, this type serves to document that a return value or function parameter should be treated as Boolean. Two enumerated constants are available:

```
enum {
  zfp_false = 0,
  zfp_true = !zfp_false
};
```

The reason why *zfp_bool* is not an enumerated type itself is that in C++ this would require an explicit cast between the `bool` type resulting from logical expressions, e.g., `zfp_bool done = static_cast<zfp_bool>(queue.empty() && work == 0)`. Such casts from `bool` to a non-enumerated `int` are not necessary.

The zfp 1.0.0 API has changed to use *zfp_bool* in place of `int` where appropriate; this change should not affect existing code.

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 link to `libzfp` when built as a static library.

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.

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

Rewind bit stream to beginning for compression or decompression.

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

Return bit stream associated with compressed stream (see `zfp_stream_set_bit_stream()`).

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

Associate bit stream with compressed stream.

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. The buffer may then be resized (using `realloc()`) after the actual number of bytes is known, as returned by `zfp_compress()`.

7.4.2 Compression Parameters

zfp_mode **`zfp_stream_compression_mode`**(const *zfp_stream* *stream)

Return compression mode associated with compression parameters. Return `zfp_mode_null` when compression parameters are invalid.

---

void **`zfp_stream_set_reversible`**(*zfp_stream* *stream)

Enable *reversible* (lossless) compression.

---

double **`zfp_stream_rate`**(const *zfp_stream* *stream, uint dims)

Return rate in compressed bits per value if *stream* is in *fixed-rate mode* (see `zfp_stream_set_rate()`), else zero. *dims* is the dimensionality of the compressed data.

---

double **`zfp_stream_set_rate`**(*zfp_stream* *stream, double rate, *zfp_type* type, uint dims, *zfp_bool* align)

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 Boolean *align* should be `zfp_true` if *word alignment* is needed, e.g., to support random access writes of blocks for zfp's *compressed arrays*. Such alignment may further constrain the rate. The closest supported rate is returned, which may differ from the requested rate.

---

uint **`zfp_stream_precision`**(const *zfp_stream* *stream)

Return precision in uncompressed bits per value if *stream* is in *fixed-precision mode* (see `zfp_stream_set_precision()`), else zero.

---

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_accuracy`**(const *zfp_stream* *stream)

Return accuracy as an absolute error tolerance if *stream* is in *fixed-accuracy mode* (see `zfp_stream_set_accuracy()`), else zero.

---

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.

---

uint64 **zfp_stream_mode**(const *zfp_stream* *stream)

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*.

---

zfp_mode **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. Return the mode associated with the newly-set compression parameters. If the decoded compression parameters are invalid, they are not set and the function returns `zfp_mode_null`.

---

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.

---

zfp_bool **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 `zfp_true` upon success.

7.4.3 Execution Policy

zfp_exec_policy **zfp_stream_execution**(const *zfp_stream* *stream)

Return current *execution policy*.

---

uint **zfp_stream_omp_threads**(const *zfp_stream* *stream)

Return number of OpenMP threads to request for compression. See *zfp_stream_set_omp_threads()*.

---

uint **zfp_stream_omp_chunk_size**(const *zfp_stream* *stream)

Return number of blocks to compress together per OpenMP thread. See *zfp_stream_set_omp_chunk_size()*.

---

zfp_bool **zfp_stream_set_execution**(*zfp_stream* *stream, *zfp_exec_policy* policy)

Set *execution policy*. If different from the previous policy, initialize the execution parameters to their default values. `zfp_true` is returned if the execution policy is supported.

---

zfp_bool **zfp_stream_set_omp_threads**(*zfp_stream* *stream, uint threads)

Set the number of OpenMP threads to use during compression. If *threads* is zero, then the number of threads is given by the value of the OpenMP *nthreads-var* internal control variable when *zfp_compress()* is called (usually the maximum number available). This function also sets the execution policy to OpenMP. Upon success, `zfp_true` is returned.

---

zfp_bool **zfp_stream_set_omp_chunk_size**(*zfp_stream* *stream, uint chunk_size)

Set the number of consecutive blocks to compress together per OpenMP thread. If zero, use one chunk per thread. This function also sets the execution policy to OpenMP. Upon success, `zfp_true` is returned.

7.4.4 Compression Configuration

These functions encode a desired compression mode and associated parameters (if any) in a single struct, e.g., for configuring zfp's *read-only array classes*.

zfp_config **`zfp_config_none`**()

Unspecified configuration.

---

zfp_config **`zfp_config_rate`**(double rate, *zfp_bool* align)

Fixed-rate mode using *rate* compressed bits per value. When *align* is true, word alignment is enforced to further constrain the rate (see *`zfp_stream_set_rate()`*).

---

zfp_config **`zfp_config_precision`**(uint precision)

Fixed-precision mode using *precision* uncompressed bits per value (see also *`zfp_stream_set_precision()`*).

---

zfp_config **`zfp_config_accuracy`**(double tolerance)

Fixed-accuracy mode with absolute error no larger than *tolerance* (see also *`zfp_stream_set_accuracy()`*).

---

zfp_config **`zfp_config_reversible`**()

Reversible (lossless) mode (see also *`zfp_stream_set_reversible()`*).

---

zfp_config **`zfp_config_expert`**(uint minbits, uint maxbits, uint maxprec, int minexp)

Expert mode with given parameters (see also *`zfp_stream_set_params()`*).

7.4.5 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, size_t 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, size_t nx, size_t 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, size_t nx, size_t ny, size_t 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.

---

zfp_field ***zfp_field_4d***(void *pointer, *zfp_type* type, size_t nx, size_t ny, size_t nz, size_t nw)

Allocate and return a field struct that describes an existing 4D array, `a[nw][nz][ny][nx]`, of $nx \times ny \times nz \times nw$ 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 field with index $x = y = z = w = 0$.

---

void ***zfp_field_begin**(const *zfp_field* *field)

Return pointer to the lowest memory address occupied by the field. Equals `zfp_field_pointer()` if all strides are positive. Available since zfp 1.0.0.

---

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, 3, or 4).

---

size_t **zfp_field_size**(const *zfp_field* *field, size_t *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.

---

size_t **zfp_field_size_bytes**(const *zfp_field* *field)

Return number of bytes spanned by the field payload data. This includes gaps in memory in case the field layout, as given by the strides, is not contiguous (see `zfp_field_is_contiguous()`). Available since zfp 1.0.0.

---

size_t **zfp_field_blocks**(const *zfp_field* *field)

Return total number of d-dimensional blocks (whether partial or whole) spanning the array. Each whole block consists of 4^d scalars. Available since zfp 1.0.0.

---

zfp_bool **zfp_field_stride**(const *zfp_field* *field, ptrdiff_t *stride)

Return `zfp_false` if the array is stored contiguously as `a[nx]`, `a[ny][nx]`, `a[nz][ny][nx]`, or `a[nw][nz][ny][nx]` depending on dimensionality. Return `zfp_true` if the array is strided and laid out differently in memory. 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 `zfp_field` for more information on strides. Return false if the array is stored contiguously (the default) as `a[nx]`, `a[ny][nx]`, `a[nz][ny][nx]`, or `a[nw][nz][ny][nx]` depending on dimensionality. Return true if nonzero strides have been specified.

---

zfp_bool **zfp_field_is_contiguous**(const *zfp_field* *field)

Return true if the field occupies a contiguous portion of memory. Note that the field layout may be contiguous even if a raster order traversal does not visit memory in a monotonically increasing or decreasing order, e.g., if the layout is simply a permutation of the default layout. Available since zfp 1.0.0.

---

uint64 **zfp_field_metadata**(const *zfp_field* *field)

Return 52-bit compact encoding of the scalar type and array dimensions. This function returns *`ZFP_META_NULL`* on failure, e.g., if the array dimensions are *too large* to be encoded in 52 bits.

---

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, size_t nx)

Specify dimensions of 1D array `a[nx]`.

---

void **zfp_field_set_size_2d**(*zfp_field* *field, size_t nx, size_t ny)

Specify dimensions of 2D array `a[ny][nx]`.

---

void **zfp_field_set_size_3d**(*zfp_field* *field, size_t nx, size_t ny, size_t nz)

Specify dimensions of 3D array `a[nz][ny][nx]`.

---

void **zfp_field_set_size_4d**(*zfp_field* *field, size_t nx, size_t ny, size_t nz, size_t nw)

Specify dimensions of 4D array `a[nw][nz][ny][nx]`.

---

void **zfp_field_set_stride_1d**(*zfp_field* *field, ptrdiff_t sx)

Specify stride for 1D array: `sx = &a[1] - &a[0]`.

---

void **zfp_field_set_stride_2d**(*zfp_field* *field, ptrdiff_t sx, ptrdiff_t 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, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t 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]`.

---

void **zfp_field_set_stride_4d**(*zfp_field* *field, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

Specify strides for 4D array: `sx = &a[0][0][0][1] - &a[0][0][0][0]; sy = &a[0][0][1][0] - &a[0][0][0][0]; sz = &a[0][1][0][0] - &a[0][0][0][0]; sw = &a[1][0][0][0] - &a[0][0][0][0]`.

---

zfp_bool **zfp_field_set_metadata**(*zfp_field* *field, uint64 meta)

Specify array scalar type and dimensions from compact 52-bit representation. Return `zfp_true` upon success. See *`zfp_field_metadata()`* for how to encode *meta*.

7.4.6 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*. Then flush the stream to emit any buffered bits and align the stream on a word boundary. The resulting byte offset within the bit stream is returned, which equals the total number of bytes of compressed storage if the stream was rewound before the *`zfp_compress()`* call. Zero is returned if compression failed.

---

size_t **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. Upon success, the nonzero return value is the same as would be returned by a corresponding *`zfp_compress()`* call, i.e., the current byte offset or the number of compressed bytes consumed. Zero is returned if decompression failed.

---

size_t **zfp_write_header**(*zfp_stream* *stream, const *zfp_field* *field, uint mask)

Write an optional variable-length header to the stream that encodes compression parameters, array metadata, etc. The header information written is determined by the bit *mask* (see `macros`). Unlike in *`zfp_compress()`*, no word alignment is enforced. See the *limitations* section for limits on the maximum array size supported by the header. 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 *stream* and *field* data structures are populated with the information stored in the header, as specified by the bit *mask* (see `macros`). The caller must ensure that *mask* agrees between header read and write calls. The return value is the number of bits read, or zero upon failure.

CHAPTER

EIGHT

LOW-LEVEL C API

The `libzfp` 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 and can be gathered from or scattered to a larger array by setting appropriate strides. As of zfp 1.0.0, templated C++ wrappers are also available to simplify calling the low-level API from C++. The C API is declared in `zfp.h`; the C++ wrappers are found in `zfp.hpp`.

Note: Because the unit of parallel work in zfp is a *block*, and because the low-level API operates on individual blocks, this API supports only the the serial *execution policy*. Any other execution policy set in `zfp_stream` is silently ignored. For parallel execution, see the *high-level API*.

The following topics are available:

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

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

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

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

size_t **zfp_encode_block_double_1**(*zfp_stream* *stream, const double *block)

Encode 1D contiguous block of 4 values.

---

size_t **zfp_encode_block_strided_int32_1**(*zfp_stream* *stream, const int32 *p, ptrdiff_t sx)

size_t **zfp_encode_block_strided_int64_1**(*zfp_stream* *stream, const int64 *p, ptrdiff_t sx)

size_t **zfp_encode_block_strided_float_1**(*zfp_stream* *stream, const float *p, ptrdiff_t sx)

size_t **zfp_encode_block_strided_double_1**(*zfp_stream* *stream, const double *p, ptrdiff_t sx)

Encode 1D complete block from strided array with stride *sx*.

---

size_t **zfp_encode_partial_block_strided_int32_1**(*zfp_stream* *stream, const int32 *p, size_t nx, ptrdiff_t sx)

size_t **zfp_encode_partial_block_strided_int64_1**(*zfp_stream* *stream, const int64 *p, size_t nx, ptrdiff_t sx)

size_t **zfp_encode_partial_block_strided_float_1**(*zfp_stream* *stream, const float *p, size_t nx, ptrdiff_t sx)

size_t **zfp_encode_partial_block_strided_double_1**(*zfp_stream* *stream, const double *p, size_t nx, ptrdiff_t sx)

Encode 1D partial block of size *nx* from strided array with stride *sx*.

8.2.2 2D Data

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

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

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

size_t **zfp_encode_block_double_2**(*zfp_stream* *stream, const double *block)

Encode 2D contiguous block of 4×4 values.

---

size_t **zfp_encode_block_strided_int32_2**(*zfp_stream* *stream, const int32 *p, ptrdiff_t sx, ptrdiff_t sy)

size_t **zfp_encode_block_strided_int64_2**(*zfp_stream* *stream, const int64 *p, ptrdiff_t sx, ptrdiff_t sy)

size_t **zfp_encode_block_strided_float_2**(*zfp_stream* *stream, const float *p, ptrdiff_t sx, ptrdiff_t sy)

size_t **zfp_encode_block_strided_double_2**(*zfp_stream* *stream, const double *p, ptrdiff_t sx, ptrdiff_t sy)

Encode 2D complete block from strided array with strides *sx* and *sy*.

---

size_t **zfp_encode_partial_block_strided_int32_2**(*zfp_stream* *stream, const int32 *p, size_t nx, size_t ny, ptrdiff_t sx, ptrdiff_t sy)

size_t **zfp_encode_partial_block_strided_int64_2**(*zfp_stream* *stream, const int64 *p, size_t nx, size_t ny, ptrdiff_t sx, ptrdiff_t sy)

size_t **zfp_encode_partial_block_strided_float_2**(*zfp_stream* *stream, const float *p, size_t nx, size_t ny, ptrdiff_t sx, ptrdiff_t sy)

size_t **zfp_encode_partial_block_strided_double_2**(*zfp_stream* *stream, const double *p, size_t nx, size_t ny, ptrdiff_t sx, ptrdiff_t sy)

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

8.2.3 3D Data

size_t **zfp_encode_block_int32_3**(*zfp_stream* *stream, const int32 *block)

size_t **zfp_encode_block_int64_3**(*zfp_stream* *stream, const int64 *block)

size_t **zfp_encode_block_float_3**(*zfp_stream* *stream, const float *block)

size_t **zfp_encode_block_double_3**(*zfp_stream* *stream, const double *block)

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

---

size_t **zfp_encode_block_strided_int32_3**(*zfp_stream* *stream, const int32 *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

size_t **zfp_encode_block_strided_int64_3**(*zfp_stream* *stream, const int64 *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz) size_t **zfp_encode_block_strided_float_3**(*zfp_stream* *stream, const float *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

size_t **zfp_encode_block_strided_double_3**(*zfp_stream* *stream, const double *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

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

---

size_t **zfp_encode_partial_block_strided_int32_3**(*zfp_stream* *stream, const int32 *p, size_t nx, size_t ny, size_t nz, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

size_t **zfp_encode_partial_block_strided_int64_3**(*zfp_stream* *stream, const int64 *p, size_t nx, size_t ny, size_t nz, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

size_t **zfp_encode_partial_block_strided_float_3**(*zfp_stream* *stream, const float *p, size_t nx, size_t ny, size_t nz, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

size_t **zfp_encode_partial_block_strided_double_3**(*zfp_stream* *stream, const double *p, size_t nx, size_t ny, size_t nz, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

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

8.2.4 4D Data

size_t **zfp_encode_block_int32_4**(*zfp_stream* *stream, const int32 *block)

size_t **zfp_encode_block_int64_4**(*zfp_stream* *stream, const int64 *block)

size_t **zfp_encode_block_float_4**(*zfp_stream* *stream, const float *block)

size_t **zfp_encode_block_double_4**(*zfp_stream* *stream, const double *block)

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

---

size_t **zfp_encode_block_strided_int32_4**(*zfp_stream* *stream, const int32 *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

size_t **zfp_encode_block_strided_int64_4**(*zfp_stream* *stream, const int64 *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

size_t **zfp_encode_block_strided_float_4**(*zfp_stream* *stream, const float *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

size_t **zfp_encode_block_strided_double_4**(*zfp_stream* *stream, const double *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

Encode 4D complete block from strided array with strides *sx*, *sy*, *sz*, and *sw*.

---

size_t **zfp_encode_partial_block_strided_int32_4**(*zfp_stream* *stream, const int32 *p, size_t nx, size_t ny, size_t nz, size_t nw, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw) size_t **zfp_encode_partial_block_strided_int64_4**(*zfp_stream* *stream, const int64 *p, size_t nx, size_t ny, size_t nz, size_t nw, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

size_t **zfp_encode_partial_block_strided_float_4**(*zfp_stream* *stream, const float *p, size_t nx, size_t ny, size_t nz, size_t nw, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

size_t **zfp_encode_partial_block_strided_double_4**(*zfp_stream* *stream, const double *p, size_t nx, size_t ny, size_t nz, size_t nw, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

Encode 4D partial block of size $nx \times ny \times nz \times nw$ from strided array with strides *sx*, *sy*, *sz*, and *sw*.

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

size_t **zfp_decode_block_int32_1**(*zfp_stream* *stream, int32 *block)

size_t **zfp_decode_block_int64_1**(*zfp_stream* *stream, int64 *block)

size_t **zfp_decode_block_float_1**(*zfp_stream* *stream, float *block)

size_t **zfp_decode_block_double_1**(*zfp_stream* *stream, double *block)

Decode 1D contiguous block of 4 values.

---

size_t **zfp_decode_block_strided_int32_1**(*zfp_stream* *stream, int32 *p, ptrdiff_t sx)

size_t **zfp_decode_block_strided_int64_1**(*zfp_stream* *stream, int64 *p, ptrdiff_t sx)

size_t **zfp_decode_block_strided_float_1**(*zfp_stream* *stream, float *p, ptrdiff_t sx)

size_t **zfp_decode_block_strided_double_1**(*zfp_stream* *stream, double *p, ptrdiff_t sx)

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

---

size_t **zfp_decode_partial_block_strided_int32_1**(*zfp_stream* *stream, int32 *p, size_t nx, ptrdiff_t sx)

size_t **zfp_decode_partial_block_strided_int64_1**(*zfp_stream* *stream, int64 *p, size_t nx, ptrdiff_t sx)

size_t **zfp_decode_partial_block_strided_float_1**(*zfp_stream* *stream, float *p, size_t nx, ptrdiff_t sx)

size_t **zfp_decode_partial_block_strided_double_1**(*zfp_stream* *stream, double *p, size_t nx, ptrdiff_t sx)

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

8.3.2 2D Data

size_t **zfp_decode_block_int32_2**(*zfp_stream* *stream, int32 *block)

size_t **zfp_decode_block_int64_2**(*zfp_stream* *stream, int64 *block)

size_t **zfp_decode_block_float_2**(*zfp_stream* *stream, float *block)

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

Decode 2D contiguous block of 4×4 values.

---

size_t **zfp_decode_block_strided_int32_2**(*zfp_stream* *stream, int32 *p, ptrdiff_t sx, ptrdiff_t sy)

size_t **zfp_decode_block_strided_int64_2**(*zfp_stream* *stream, int64 *p, ptrdiff_t sx, ptrdiff_t sy)

size_t **zfp_decode_block_strided_float_2**(*zfp_stream* *stream, float *p, ptrdiff_t sx, ptrdiff_t sy)

size_t **zfp_decode_block_strided_double_2**(*zfp_stream* *stream, double *p, ptrdiff_t sx, ptrdiff_t sy)

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

---

size_t **zfp_decode_partial_block_strided_int32_2**(*zfp_stream* *stream, int32 *p, size_t nx, size_t ny, ptrdiff_t sx, ptrdiff_t sy)

size_t **zfp_decode_partial_block_strided_int64_2**(*zfp_stream* *stream, int64 *p, size_t nx, size_t ny, ptrdiff_t sx, ptrdiff_t sy)

size_t **zfp_decode_partial_block_strided_float_2**(*zfp_stream* *stream, float *p, size_t nx, size_t ny, ptrdiff_t sx, ptrdiff_t sy)

size_t **zfp_decode_partial_block_strided_double_2**(*zfp_stream* *stream, double *p, size_t nx, size_t ny, ptrdiff_t sx, ptrdiff_t sy)

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

8.3.3 3D Data

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

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

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

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

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

---

size_t **zfp_decode_block_strided_int32_3**(*zfp_stream* *stream, int32 *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

size_t **zfp_decode_block_strided_int64_3**(*zfp_stream* *stream, int64 *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

size_t **zfp_decode_block_strided_float_3**(*zfp_stream* *stream, float *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz) size_t **zfp_decode_block_strided_double_3**(*zfp_stream* *stream, double *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

Decode 3D complete block to strided array with strides *sx*, *sy*, and *sz*.

---

size_t **zfp_decode_partial_block_strided_int32_3**(*zfp_stream* *stream, int32 *p, size_t nx, size_t ny, size_t nz, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

size_t **zfp_decode_partial_block_strided_int64_3**(*zfp_stream* *stream, int64 *p, size_t nx, size_t ny, size_t nz, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

size_t **zfp_decode_partial_block_strided_float_3**(*zfp_stream* *stream, float *p, size_t nx, size_t ny, size_t nz, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

size_t **zfp_decode_partial_block_strided_double_3**(*zfp_stream* *stream, double *p, size_t nx, size_t ny, size_t nz, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

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

8.3.4 4D Data

size_t **zfp_decode_block_int32_4**(*zfp_stream* *stream, int32 *block)

size_t **zfp_decode_block_int64_4**(*zfp_stream* *stream, int64 *block)

size_t **zfp_decode_block_float_4**(*zfp_stream* *stream, float *block)

size_t **zfp_decode_block_double_4**(*zfp_stream* *stream, double *block)

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

---

size_t **zfp_decode_block_strided_int32_4**(*zfp_stream* *stream, int32 *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

size_t **zfp_decode_block_strided_int64_4**(*zfp_stream* *stream, int64 *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

size_t **zfp_decode_block_strided_float_4**(*zfp_stream* *stream, float *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

size_t **zfp_decode_block_strided_double_4**(*zfp_stream* *stream, double *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

Decode 4D complete block to strided array with strides *sx*, *sy*, *sz*, and *sw*.

---

size_t **zfp_decode_partial_block_strided_int32_4**(*zfp_stream* *stream, int32 *p, size_t nx, size_t ny, size_t nz, size_t nw, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

size_t **zfp_decode_partial_block_strided_int64_4**(*zfp_stream* *stream, int64 *p, size_t nx, size_t ny, size_t nz, size_t nw, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw) size_t **`zfp_decode_partial_block_strided_float_4`**(*zfp_stream* *stream, float *p, size_t nx, size_t ny, size_t nz, size_t nw, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

size_t **`zfp_decode_partial_block_strided_double_4`**(*zfp_stream* *stream, double *p, size_t nx, size_t ny, size_t nz, size_t nw, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

Decode 4D partial block of size $nx \times ny \times nz \times nw$ to strided array with strides *sx*, *sy*, *sz*, and *sw*.

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. Use *dims* = 0 to promote a single value.

---

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. Use *dims* = 0 to demote a single value.

8.5 C++ Wrappers

To facilitate calling the low-level API from C++, a number of wrappers are available (as of zfp 1.0.0) that are templated on scalar type and dimensionality. Each function of the form `zfp_function_type_dims`, where *type* denotes scalar type and *dims* denotes dimensionality, has a corresponding C++ wrapper `zfp::function<type, dims>`. For example, the C function *`zfp_encode_block_float_2()`* has a C++ wrapper *`zfp::encode_block<float, 2>()`*. Often *dims* can be inferred from the parameters of overloaded functions, in which case it is omitted as template parameter. The C++ wrappers are defined in `zfp.hpp`.

8.5.1 Encoder

template<typename **Scalar**, uint **dims**>
size_t **encode_block**(zfp_stream *stream, const *Scalar* *block)

Encode contiguous block of dimensionality *dims*.

---

template<typename **Scalar**>
size_t **encode_block_strided**(zfp_stream *stream, const *Scalar* *p, ptrdiff_t sx)

template<typename **Scalar**>
size_t **encode_block_strided**(zfp_stream *stream, const *Scalar* *p, ptrdiff_t sx, ptrdiff_t sy)

template<typename **Scalar**>
size_t **encode_block_strided**(zfp_stream *stream, const *Scalar* *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

template<typename **Scalar**>
size_t **encode_block_strided**(zfp_stream *stream, const *Scalar* *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

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

---

template<typename **Scalar**>
size_t **encode_partial_block_strided**(zfp_stream *stream, const *Scalar* *p, size_t nx, ptrdiff_t sx)

template<typename **Scalar**>
size_t **encode_partial_block_strided**(zfp_stream *stream, const *Scalar* *p, size_t nx, size_t ny, ptrdiff_t sx, ptrdiff_t sy)

template<typename **Scalar**>
size_t **encode_partial_block_strided**(zfp_stream *stream, const *Scalar* *p, size_t nx, size_t ny, size_t nz, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

template<typename **Scalar**>
size_t **encode_partial_block_strided**(zfp_stream *stream, const *Scalar* *p, size_t nx, size_t ny, size_t nz, size_t nw, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

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

8.5.2 Decoder

template<typename **Scalar**, uint **dims**>
size_t **decode_block**(zfp_stream *stream, *Scalar* *block)

Decode contiguous block of dimensionality *dims*.

---

template<typename **Scalar**>
size_t **decode_block_strided**(zfp_stream *stream, *Scalar* *p, ptrdiff_t sx)

template<typename **Scalar**>
size_t **decode_block_strided**(zfp_stream *stream, *Scalar* *p, ptrdiff_t sx, ptrdiff_t sy)

template<typename **Scalar**>

size_t **decode_block_strided**(zfp_stream *stream, *Scalar* *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

template<typename **Scalar**>
size_t **decode_block_strided**(zfp_stream *stream, *Scalar* *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

Decode complete block to strided array with strides *sx*, *sy*, *sz*, and *sw*.

---

template<typename **Scalar**>
size_t **decode_partial_block_strided**(zfp_stream *stream, *Scalar* *p, size_t nx, ptrdiff_t sx)

template<typename **Scalar**>
size_t **decode_partial_block_strided**(zfp_stream *stream, *Scalar* *p, size_t nx, size_t ny, ptrdiff_t sx, ptrdiff_t sy)

template<typename **Scalar**>
size_t **decode_partial_block_strided**(zfp_stream *stream, *Scalar* *p, size_t nx, size_t ny, size_t nz, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz)

template<typename **Scalar**>
size_t **decode_partial_block_strided**(zfp_stream *stream, *Scalar* *p, size_t nx, size_t ny, size_t nz, size_t nw, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw)

Decode partial block of size $nx \times ny \times nz \times nw$ to strided array with strides *sx*, *sy*, *sz*, and *sw*.

CHAPTER

NINE

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 and 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

type **bitstream_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*).

Note: To avoid potential name clashes, this type was renamed in zfp 1.0.0 from the shorter and more ambiguous type name `word`.

type **bitstream_offset**

Type holding the offset, measured in number of bits, into the bit stream where the next bit will be read or written. This type allows referencing bits in streams at least 2^{64} bits long. Note that it is possible that `sizeof(bitstream_offset) > sizeof(size_t)` since a stream may be as long as *sizeof(size_t) * CHAR_BIT* bits.

type **bitstream_size**

Alias for *bitstream_offset* that signifies the bit length of a stream or substream rather than an offset into it.

type **bitstream_count**

Type sufficient to count the number of bits read or written in functions like *stream_read_bits()* and *stream_write_bits()*. `sizeof(bitstream_count) <= sizeof(bitstream_size)`.

type **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 {
  bitstream_count bits;  // number of buffered bits (0 <= bits < word size)
  bitstream_word buffer; // incoming/outgoing bits (buffer < 2^bits)
  bitstream_word* ptr;   // pointer to next word to be read/written
  bitstream_word* begin; // beginning of stream
  bitstream_word* end;   // end of stream (not enforced)
  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` and `stream_alignment()`.

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*.

---

bitstream ***stream_clone**(const *bitstream* *stream)

Create a copy of *stream* that points to the same memory buffer.

---

bitstream_count **stream_alignment**()

Word size in bits. This is a functional form of the constant `stream_word_bits` and returns the same value. Available since zfp 1.0.0.

---

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* specified in `stream_open()`.

---

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

Read a single bit from *stream*.

---

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

Write single *bit* to *stream*. *bit* must be one of 0 or 1. The value of *bit* is returned.

---

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

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

---

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

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

---

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

Return bit offset to next bit to be read.

---

bitstream_offset **`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, *bitstream_offset* offset)

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

---

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

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

---

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

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

---

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

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

---

bitstream_count **`stream_align`**(*bitstream* *stream)

Align stream on next word boundary by skipping bits, i.e., without reading them. No skipping is done if the stream is already word aligned. Return the number of skipped bits, if any.

---

bitstream_count **`stream_flush`**(*bitstream* *stream)

Write out any remaining buffered bits. When one or more bits are buffered, append zero-bits to the stream to align it on a word boundary. Return the number of bits of padding, if any.

---

void **stream_copy**(*bitstream* *dst, *bitstream* *src, *bitstream_size* n)

Copy *n* bits from *src* to *dst*, advancing both bit streams.

---

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, *block*, in number of words and spacing, *delta*, in number of blocks for *strided access*. Return nonzero upon success. Requires `BIT_STREAM_STRIDED`.

CHAPTER

TEN

PYTHON BINDINGS

zfp 0.5.5 adds zfPy: Python bindings that allow compressing and decompressing NumPy integer and floating-point arrays. The zfPy implementation is based on Cython and requires both NumPy and Cython to be installed. Currently, zfPy supports only serial execution.

The zfPy API is limited to two functions, for compression and decompression, which are described below.

10.1 Compression

`zfpy.compress_numpy`(*arr*, *tolerance=-1*, *rate=-1*, *precision=-1*, *write_header=True*)

Compress NumPy array, *arr*, and return a compressed byte stream. The non-expert *compression mode* is selected by setting one of *tolerance*, *rate*, or *precision*. If none of these arguments is specified, then *reversible mode* is used. By default, a header that encodes array shape and scalar type as well as compression parameters is prepended, which can be omitted by setting *write_header* to *False*. If this function fails for any reason, an exception is thrown.

zfPy compression currently requires a NumPy array (ndarray) populated with the data to be compressed. The array metadata (i.e., shape, strides, and scalar type) are used to automatically populate the *`zfp_field`* structure passed to *`zfp_compress()`*. By default, all that is required to be passed to the compression function is the NumPy array; this will result in a stream that includes a header and is losslessly compressed using the *reversible mode*. For example:

```python
import zfpy
import numpy as np

my_array = np.arange(1, 20)
compressed_data = zfpy.compress_numpy(my_array)
decompressed_array = zfpy.decompress_numpy(compressed_data)

# confirm lossless compression/decompression
np.testing.assert_array_equal(my_array, decompressed_array)
```

Using the fixed-accuracy, fixed-rate, or fixed-precision modes simply requires setting one of the *tolerance*, *rate*, or *precision* arguments, respectively. For example:

```python
compressed_data = zfpy.compress_numpy(my_array, tolerance=1e-3)
decompressed_array = zfpy.decompress_numpy(compressed_data)

# Note the change from "equal" to "allclose" due to the lossy compression
np.testing.assert_allclose(my_array, decompressed_array, atol=1e-3)
```

Since NumPy arrays are C-ordered by default (i.e., the rightmost index varies fastest) and `zfp_compress()` assumes Fortran ordering (i.e., the leftmost index varies fastest), `compress_numpy()` automatically reverses the order of dimensions and strides in order to improve the expected memory access pattern during compression. The `decompress_numpy()` function also reverses the order of dimensions and strides, and therefore decompression will restore the shape of the original array. Note, however, that the zfp stream does not encode the memory layout of the original NumPy array, and therefore layout information like strides, contiguity, and C vs. Fortran order may not be preserved. Nevertheless, zfPy correctly compresses NumPy arrays with any memory layout, including Fortran ordering and non-contiguous storage.

Byte streams produced by `compress_numpy()` can be decompressed by the *zfp command-line tool*. In general, they cannot be *deserialized* as compressed arrays, however.

Note: `decompress_numpy()` requires a header to decompress properly, so do not set *write_header* = *False* during compression if you intend to decompress the stream with zfPy.

10.2 Decompression

zfpy.**decompress_numpy**(*compressed_data*)

Decompress a byte stream, *compressed_data*, produced by `compress_numpy()` (with header enabled) and return the decompressed NumPy array. This function throws on exception upon error.

`decompress_numpy()` consumes a compressed stream that includes a header and produces a NumPy array with metadata populated based on the contents of the header. Stride information is not stored in the zfp header, so `decompress_numpy()` assumes that the array was compressed with the first (leftmost) dimension varying fastest (typically referred to as Fortran-ordering). The returned NumPy array is in C-ordering (the default for NumPy arrays), so the shape of the returned array is reversed from the shape information stored in the embedded header. For example, if the header declares the array to be of shape (nx, ny, nz) = (2, 4, 8), then the returned NumPy array will have a shape of (8, 4, 2). Since the `compress_numpy()` function also reverses the order of dimensions, arrays both compressed and decompressed with zfPy will have compatible shape.

Note: Decompressing a stream without a header requires using the internal `_decompress()` Python function (or the *C API*).

zfpy.**_decompress**(*compressed_data*, *ztype*, *shape*, *out=None*, *tolerance=-1*, *rate=-1*, *precision=-1*)

Decompress a headerless compressed stream (if a header is present in the stream, it will be incorrectly interpreted as compressed data). *ztype* specifies the array scalar type while *shape* specifies the array dimensions; both must be known by the caller. The compression mode is selected by specifying one (or none) of *tolerance*, *rate*, and *precision*, as in `compress_numpy()`, and also must be known by the caller. If *out* = *None*, a new NumPy array is allocated. Otherwise, *out* specifies the NumPy array or memory buffer to decompress into. Regardless, the decompressed NumPy array is returned unless an error occurs, in which case an exception is thrown.

In `_decompress()`, *ztype* is one of the zfp supported scalar types (see `zfp_type`), which are available in zfPy as

```
type_int32 = zfp_type_int32
type_int64 = zfp_type_int64
type_float = zfp_type_float
type_double = zfp_type_double
```

These can be manually specified (e.g., `zfpy.type_int32`) or generated from a NumPy *dtype* (e.g., `zfpy.dtype_to_ztype(array.dtype)`).

If *out* is specified, the data is decompressed into the *out* buffer. *out* can be a NumPy array or a pointer to memory large enough to hold the decompressed data. Regardless of the type of *out* and whether it is provided, *_decompress()* always returns a NumPy array. If *out* is not provided, then the array is allocated for the user. If *out* is provided, then the returned NumPy array is just a pointer to or wrapper around the user-supplied *out*. If *out* is a NumPy array, then its shape and scalar type must match the required arguments *shape* and *ztype*. To avoid this constraint check, use `out = ndarray.data` rather than `out = ndarray` when calling *_decompress()*.

Warning: *_decompress()* is an “experimental” function currently used internally for testing. It does allow decompression of streams without headers, but providing too small of an output buffer or incorrectly specifying the shape or strides can result in segmentation faults. Use with care.

CHAPTER

ELEVEN

FORTRAN BINDINGS

zfp 0.5.5 adds zFORp: a Fortran API providing wrappers around the *high-level C API*. Wrappers for *compressed-array classes* will arrive in a future release. The zFORp implementation is based on the standard `iso_c_binding` module available since Fortran 2003. The use of `ptrdiff_t` in the zfp 1.0.0 C API, however, requires the corresponding `c_ptrdiff_t` available only since Fortran 2018.

Every high-level C API function can be called from a Fortran wrapper function. C structs are wrapped as Fortran derived types, each containing a single C pointer to the C struct in memory. The wrapper functions accept and return these Fortran types, so users should never need to touch the C pointers. In addition to the high-level C API, two essential functions from the *bit stream API* for opening and closing bit streams are available.

See example code `tests/fortran/testFortran.f` (on the GitHub develop branch) for how the Fortran API is used to compress and decompress data.

Note: zfp 1.0.0 simplifies the zFORp module name from `zforp_module` to `zfp`. This will likely require changing associated use statements within existing code when updating from prior versions of zFORp.

Furthermore, as outlined above, the zfp 1.0.0 API requires a Fortran 2018 compiler.

11.1 Types

type zFORp_bitstream

Type fields

- **% object** *[c_ptr]* :: A C pointer to the instance of *bitstream*

type zFORp_stream

Type fields

- **% object** *[c_ptr]* :: A C pointer to the instance of *zfp_stream*

type zFORp_field

Type fields

- **% object** *[c_ptr]* :: A C pointer to the instance of *zfp_field*

11.2 Constants

11.2.1 Enumerations

integer zFORp_type_none

integer zFORp_type_int32

integer zFORp_type_int64

integer zFORp_type_float

integer zFORp_type_double

Enums wrapping *zfp_type*

---

integer zFORp_mode_null

integer zFORp_mode_expert

integer zFORp_mode_fixed_rate

integer zFORp_mode_fixed_precision

integer zFORp_mode_fixed_accuracy

integer zFORp_mode_reversible

Enums wrapping *zfp_mode*

---

integer zFORp_exec_serial

integer zFORp_exec_omp

integer zFORp_exec_cuda

Enums wrapping *zfp_exec_policy*

11.2.2 Non-Enum Constants

integer zFORp_version_major

Wraps *ZFP_VERSION_MAJOR*

---

integer zFORp_version_minor

Wraps *ZFP_VERSION_MINOR*

---

integer zFORp_version_patch

Wraps *ZFP_VERSION_PATCH*

---

integer zFORp_version_tweak

Wraps *ZFP_VERSION_TWEAK*

integer zFORp_codec_version

Wraps *zfp_codec_version*

integer zFORp_library_version

Wraps *zfp_library_version*

character(len=36) zFORp_version_string

Wraps *zfp_version_string*

integer zFORp_min_bits

Wraps *ZFP_MIN_BITS*

integer zFORp_max_bits

Wraps *ZFP_MAX_BITS*

integer zFORp_max_prec

Wraps *ZFP_MAX_PREC*

integer zFORp_min_exp

Wraps *ZFP_MIN_EXP*

integer zFORp_header_magic

Wraps *ZFP_HEADER_MAGIC*

integer zFORp_header_meta

Wraps *ZFP_HEADER_META*

integer zFORp_header_mode

Wraps *ZFP_HEADER_MODE*

integer zFORp_header_full

Wraps *ZFP_HEADER_FULL*

`integer zFORp_meta_null`

Wraps *ZFP_META_NULL*

`integer zFORp_magic_bits`

Wraps *ZFP_MAGIC_BITS*

`integer zFORp_meta_bits`

Wraps *ZFP_META_BITS*

`integer zFORp_mode_short_bits`

Wraps *ZFP_MODE_SHORT_BITS*

`integer zFORp_mode_long_bits`

Wraps *ZFP_MODE_LONG_BITS*

`integer zFORp_header_max_bits`

Wraps *ZFP_HEADER_MAX_BITS*

`integer zFORp_mode_short_max`

Wraps *ZFP_MODE_SHORT_MAX*

11.3 Functions and Subroutines

Each of the functions included here wraps a corresponding C function. Please consult the C documentation for detailed descriptions of the functions, their parameters, and their return values.

11.3.1 Bit Stream

`function zFORp_bitstream_stream_open`(*buffer*, *bytes*)

Wrapper for *stream_open()*

Parameters

- **buffer** *[c_ptr,in]* :: Memory buffer
- **bytes** *[integer (kind=8),in]* :: Buffer size in bytes

Return

bs *[*zFORp_bitstream*]* :: Bit stream

`subroutine zFORp_bitstream_stream_close`(*bs*)

Wrapper for *stream_close()*

Parameters

bs *[*zFORp_bitstream,*inout]* :: Bit stream

11.3.2 Utility Functions

function zFORp_type_size(*scalar_type*)

Wrapper for *zfp_type_size()*

Parameters

scalar_type *[integer,in]* :: *zFORp_type* enum

Return

type_size *[integer (kind=8)]* :: Size of described *zfp_type*, in bytes, from C-language perspective

11.3.3 Compressed Stream

function zFORp_stream_open(*bs*)

Wrapper for *zfp_stream_open()*

Parameters

bs *[*zFORp_bitstream,*in]* :: Bit stream

Return

stream *[*zFORp_stream*]* :: Newly allocated compressed stream

---

subroutine zFORp_stream_close(*stream*)

Wrapper for *zfp_stream_close()*

Parameters

stream *[*zFORp_stream,*inout]* :: Compressed stream

---

function zFORp_stream_bit_stream(*stream*)

Wrapper for *zfp_stream_bit_stream()*

Parameters

stream *[*zFORp_stream,*in]* :: Compressed stream

Return

bs *[*zFORp_bitstream*]* :: Bit stream

---

function zFORp_stream_compression_mode(*stream*)

Wrapper for *zfp_stream_compression_mode()*

Parameters

stream *[*zFORp_stream,*in]* :: Compressed stream

Return

mode *[integer]* :: *zFORp_mode* enum

`function zFORp_stream_rate`(*stream*, *dims*)

Wrapper for `zfp_stream_rate()`

Parameters

- **stream** *[*zFORp_stream*,in]* :: Compressed stream
- **dims** *[integer,in]* :: Number of dimensions

Return

rate_result *[real (kind=8)]* :: Rate in compressed bits/scalar

---

`function zFORp_stream_precision`(*stream*)

Wrapper for `zfp_stream_precision()`

Parameters

stream *[*zFORp_stream*,in]* :: Compressed stream

Return

prec_result *[integer]* :: Precision in uncompressed bits/scalar

---

`function zFORp_stream_accuracy`(*stream*)

Wrapper for `zfp_stream_accuracy()`

Parameters

stream *[*zFORp_stream*,in]* :: Compressed stream

Return

tol_result *[real (kind=8)]* :: Absolute error tolerance

`function zFORp_stream_mode`(*stream*)

Wrapper for `zfp_stream_mode()`

Parameters

stream *[*zFORp_stream*,in]* :: Compressed stream

Return

mode *[integer (kind=8)]* :: 64-bit encoded mode

`subroutine zFORp_stream_params`(*stream*, *minbits*, *maxbits*, *maxprec*, *minexp*)

Wrapper for `zfp_stream_params()`

Parameters

- **stream** *[*zFORp_stream*,in]* :: Compressed stream
- **minbits** *[integer,inout]* :: Minimum number of bits per block
- **maxbits** *[integer,inout]* :: Maximum number of bits per block
- **maxprec** *[integer,inout]* :: Maximum precision
- **minexp** *[integer,inout]* :: Minimum bit plane number encoded

---

function `zFORp_stream_compressed_size`(*stream*)

Wrapper for *`zfp_stream_compressed_size()`*

Parameters
stream *[zFORp_stream,in]* :: Compressed stream

Return
compressed_size *[integer (kind=8)]* :: Compressed size in bytes

---

function `zFORp_stream_maximum_size`(*stream*, *field*)

Wrapper for *`zfp_stream_maximum_size()`*

Parameters

- **stream** *[zFORp_stream,in]* :: Compressed stream
- **field** *[zFORp_field,in]* :: Field metadata

Return
max_size *[integer (kind=8)]* :: Maximum possible compressed size in bytes

---

subroutine `zFORp_stream_rewind`(*stream*)

Wrapper for *`zfp_stream_rewind()`*

Parameters
stream *[zFORp_stream,in]* :: Compressed stream

---

subroutine `zFORp_stream_set_bit_stream`(*stream*, *bs*)

Wrapper for *`zfp_stream_set_bit_stream()`*

Parameters

- **stream** *[zFORp_stream,in]* :: Compressed stream
- **bs** *[zFORp_bitstream,in]* :: Bit stream

11.3.4 Compression Parameters

subroutine `zFORp_stream_set_reversible`(*stream*)

Wrapper for *`zfp_stream_set_reversible()`*

Parameters
stream *[zFORp_stream,in]* :: Compressed stream

---

function `zFORp_stream_set_rate`(*stream*, *rate*, *scalar_type*, *dims*, *align*)

Wrapper for *`zfp_stream_set_rate()`*

Parameters

- **stream** *[zFORp_stream,in]* :: Compressed stream
- **rate** *[real,in]* :: Desired rate
- **scalar_type** *[integer,in]* :: *zFORp_type* enum

- **dims** *[integer,in]* :: Number of dimensions
- **align** *[integer,in]* :: Align blocks on words for write random access?

Return
rate_result *[real (kind=8)]* :: Actual set rate in bits/scalar

---

`function zFORp_stream_set_precision`(*stream*, *prec*)
Wrapper for *`zfp_stream_set_precision()`*

Parameters

- **stream** *[*zFORp_stream*,in]* :: Compressed stream
- **prec** *[integer,in]* :: Desired precision

Return
prec_result *[integer]* :: Actual set precision

`function zFORp_stream_set_accuracy`(*stream*, *tolerance*)
Wrapper for *`zfp_stream_set_accuracy()`*

Parameters

- **stream** *[*zFORp_stream*,in]* :: Compressed stream
- **tolerance** *[real (kind=8),in]* :: Desired error tolerance

Return
tol_result *[real (kind=8)]* :: Actual set tolerance

`function zFORp_stream_set_mode`(*stream*, *mode*)
Wrapper for *`zfp_stream_set_mode()`*

Parameters

- **stream** *[*zFORp_stream*,in]* :: Compressed stream
- **mode** *[integer (kind=8),in]* :: Compact encoding of compression parameters

Return
mode_result *[integer]* :: Newly set *zFORp_mode* enum

`function zFORp_stream_set_params`(*stream*, *minbits*, *maxbits*, *maxprec*, *minexp*)
Wrapper for *`zfp_stream_set_params()`*

Parameters

- **stream** *[*zFORp_stream*,in]* :: Compressed stream
- **minbits** *[integer,in]* :: Minimum number of bits per block
- **maxbits** *[integer,in]* :: Maximum number of bits per block
- **maxprec** *[integer,in]* :: Maximum precision
- **minexp** *[integer,in]* :: Minimum bit plane number encoded

Return
is_success *[integer]* :: Indicate whether parameters were successfully set (1) or not (0)

11.3.5 Execution Policy

function zFORp_stream_execution(*stream*)

Wrapper for *zfp_stream_execution()*

Parameters

stream *[*zFORp_stream,*in]* :: Compressed stream

Return

execution_policy *[integer]* :: *zFORp_exec* enum indicating active execution policy

---

function zFORp_stream_omp_threads(*stream*)

Wrapper for *zfp_stream_omp_threads()*

Parameters

stream *[*zFORp_stream,*in]* :: Compressed stream

Return

thread_count *[integer]* :: Number of OpenMP threads to use upon execution

---

function zFORp_stream_omp_chunk_size(*stream*)

Wrapper for *zfp_stream_omp_chunk_size()*

Parameters

stream *[*zFORp_stream,*in]* :: Compressed stream

Return

chunk_size_blocks *[integer (kind=8)]* :: Specified chunk size, in blocks

---

function zFORp_stream_set_execution(*stream*, *execution_policy*)

Wrapper for *zfp_stream_set_execution()*

Parameters

- **stream** *[*zFORp_stream,*in]* :: Compressed stream
- **execution_policy** *[integer,in]* :: *zFORp_exec* enum indicating desired execution policy

Return

is_success *[integer]* :: Indicate whether execution policy was successfully set (1) or not (0)

---

function zFORp_stream_set_omp_threads(*stream*, *thread_count*)

Wrapper for *zfp_stream_set_omp_threads()*

Parameters

- **stream** *[*zFORp_stream,*in]* :: Compressed stream
- **thread_count** *[integer,in]* :: Desired number of OpenMP threads

Return

is_success *[integer]* :: Indicate whether number of threads was successfully set (1) or not (0)

---

function zFORp_stream_set_omp_chunk_size(*stream*, *chunk_size*)

Wrapper for *zfp_stream_set_omp_chunk_size()*

Parameters

- **stream** *[zFORp_stream,in]* :: Compressed stream
- **chunk_size** *[integer,in]* :: Desired chunk size, in blocks

Return

is_success *[integer]* :: Indicate whether chunk size was successfully set (1) or not (0)

11.3.6 Array Metadata

function zFORp_field_alloc()

Wrapper for *zfp_field_alloc()*

Return

field *[zFORp_field]* :: Newly allocated field

---

function zFORp_field_1d(*uncompressed_ptr*, *scalar_type*, *nx*)

Wrapper for *zfp_field_1d()*

Parameters

- **uncompressed_ptr** *[c_ptr,in]* :: Pointer to uncompressed data
- **scalar_type** *[integer,in]* :: *zFORp_type* enum describing uncompressed scalar type
- **nx** *[integer,in]* :: Number of array elements

Return

field *[zFORp_field]* :: Newly allocated field

---

function zFORp_field_2d(*uncompressed_ptr*, *scalar_type*, *nx*, *ny*)

Wrapper for *zfp_field_2d()*

Parameters

- **uncompressed_ptr** *[c_ptr,in]* :: Pointer to uncompressed data
- **scalar_type** *[integer,in]* :: *zFORp_type* enum describing uncompressed scalar type
- **nx** *[integer,in]* :: Number of array elements in x dimension
- **ny** *[integer,in]* :: Number of array elements in y dimension

Return

field *[zFORp_field]* :: Newly allocated field

---

function zFORp_field_3d(*uncompressed_ptr*, *scalar_type*, *nx*, *ny*, *nz*)

Wrapper for *zfp_field_3d()*

Parameters

- **uncompressed_ptr** *[c_ptr,in]* :: Pointer to uncompressed data

- **scalar_type** *[integer,in]* :: *zFORp_type* enum describing uncompressed scalar type
- **nx** *[integer,in]* :: Number of array elements in *x* dimension
- **ny** *[integer,in]* :: Number of array elements in *y* dimension
- **nz** *[integer,in]* :: Number of array elements in *z* dimension

Return
field *[*zFORp_field*]* :: Newly allocated field

`function zFORp_field_4d`(*uncompressed_ptr*, *scalar_type*, *nx*, *ny*, *nz*, *nw*)

Wrapper for `zfp_field_4d()`

Parameters

- **uncompressed_ptr** *[c_ptr,in]* :: Pointer to uncompressed data
- **scalar_type** *[integer,in]* :: *zFORp_type* enum describing uncompressed scalar type
- **nx** *[integer,in]* :: Number of array elements in *x* dimension
- **ny** *[integer,in]* :: Number of array elements in *y* dimension
- **nz** *[integer,in]* :: Number of array elements in *z* dimension
- **nw** *[integer,in]* :: Number of array elements in *w* dimension

Return
field *[*zFORp_field*]* :: Newly allocated field

`subroutine zFORp_field_free`(*field*)

Wrapper for `zfp_field_free()`

Parameters
field *[*zFORp_field*,inout]* :: Field metadata

`function zFORp_field_pointer`(*field*)

Wrapper for `zfp_field_pointer()`

Parameters
field *[*zFORp_field*,in]* :: Field metadata

Return
arr_ptr *[c_ptr]* :: Pointer to raw (uncompressed/decompressed) array

---

`function zFORp_field_begin`(*field*)

Wrapper for `zfp_field_begin()`

Parameters
field *[*zFORp_field*,in]* :: Field metadata

Return
begin_ptr *[c_ptr]* :: Pointer to lowest memory address spanned by field

---

function zFORp_field_type(*field*)

Wrapper for *zfp_field_type()*

Parameters
field *[zFORp_field,in]* :: Field metadata

Return
scalar_type *[integer]* :: *zFORp_type* enum describing uncompressed scalar type

---

function zFORp_field_precision(*field*)

Wrapper for *zfp_field_precision()*

Parameters
field *[zFORp_field,in]* :: Field metadata

Return
prec *[integer]* :: Scalar type precision in number of bits

---

function zFORp_field_dimensionality(*field*)

Wrapper for *zfp_field_dimensionality()*

Parameters
field *[zFORp_field,in]* :: Field metadata

Return
dims *[integer]* :: Dimensionality of array

---

function zFORp_field_size(*field*, *size_arr*)

Wrapper for *zfp_field_size()*

Parameters

- **field** *[zFORp_field,in]* :: Field metadata
- **size_arr** *[integer,dimension(4),target,inout]* :: Integer array to write field dimensions into

Return
total_size *[integer (kind=8)]* :: Total number of array elements

---

function zFORp_field_size_bytes(*field*)

Wrapper for *zfp_field_size_bytes()*

Parameters
field *[zFORp_field,in]* :: Field metadata

Return
byte_size *[integer (kind=8)]* :: Number of bytes spanned by field data including gaps (if any)

`function zFORp_field_blocks`(*field*)

Wrapper for *zfp_field_blocks()*

Parameters

field *[*zFORp_field,*in]* :: Field metadata

Return

blocks *[integer (kind=8)]* :: Total number of blocks spanned by field

---

`function zFORp_field_stride`(*field*, *stride_arr*)

Wrapper for *zfp_field_stride()*

Parameters

- **field** *[*zFORp_field,*in]* :: Field metadata
- **stride_arr** *[integer,dimension(4),target,inout]* :: Integer array to write strides into

Return

is_strided *[integer]* :: Indicate whether field is strided (1) or not (0)

`function zFORp_field_is_contiguous`(*field*)

Wrapper for *zfp_field_is_contiguous()*

Parameters

field *[*zFORp_field,*in]* :: Field metadata

Return

is_contiguous *[integer]* :: Indicate whether field is contiguous (1) or not (0)

`function zFORp_field_metadata`(*field*)

Wrapper for *zfp_field_metadata()*

Parameters

field *[*zFORp_field,*in]* :: Field metadata

Return

encoded_metadata *[integer (kind=8)]* :: Compact encoding of metadata

---

`subroutine zFORp_field_set_pointer`(*field*, *arr_ptr*)

Wrapper for *zfp_field_set_pointer()*

Parameters

- **field** *[*zFORp_field,*in]* :: Field metadata
- **arr_ptr** *[c_ptr,in]* :: Pointer to beginning of uncompressed array

---

function zFORp_field_set_type(*field*, *scalar_type*)

Wrapper for `zfp_field_set_type()`

Parameters

- **field** *[zFORp_field,in]* :: Field metadata
- **scalar_type** *[integer]* :: *zFORp_type* enum indicating desired scalar type

Return

type_result *[integer]* :: *zFORp_type* enum indicating actual scalar type

---

subroutine zFORp_field_set_size_1d(*field*, *nx*)

Wrapper for `zfp_field_set_size_1d()`

Parameters

- **field** *[zFORp_field,in]* :: Field metadata
- **nx** *[integer,in]* :: Number of array elements

---

subroutine zFORp_field_set_size_2d(*field*, *nx*, *ny*)

Wrapper for `zfp_field_set_size_2d()`

Parameters

- **field** *[zFORp_field,in]* :: Field metadata
- **nx** *[integer,in]* :: Number of array elements in x dimension
- **ny** *[integer,in]* :: Number of array elements in y dimension

---

subroutine zFORp_field_set_size_3d(*field*, *nx*, *ny*, *nz*)

Wrapper for `zfp_field_set_size_3d()`

Parameters

- **field** *[zFORp_field,in]* :: Field metadata
- **nx** *[integer,in]* :: Number of array elements in x dimension
- **ny** *[integer,in]* :: Number of array elements in y dimension
- **nz** *[integer,in]* :: Number of array elements in z dimension

---

subroutine zFORp_field_set_size_4d(*field*, *nx*, *ny*, *nz*, *nw*)

Wrapper for `zfp_field_set_size_4d()`

Parameters

- **field** *[zFORp_field,in]* :: Field metadata
- **nx** *[integer,in]* :: Number of array elements in x dimension
- **ny** *[integer,in]* :: Number of array elements in y dimension
- **nz** *[integer,in]* :: Number of array elements in z dimension

- **nw** *[integer,in]* :: Number of array elements in *w* dimension

---

subroutine zFORp_field_set_stride_1d(*field*, *sx*)

Wrapper for `zfp_field_set_stride_1d()`

Parameters

- **field** *[*zFORp_field*,in]* :: Field metadata
- **sx** *[integer,in]* :: Stride in number of scalars

---

subroutine zFORp_field_set_stride_2d(*field*, *sx*, *sy*)

Wrapper for `zfp_field_set_stride_2d()`

Parameters

- **field** *[*zFORp_field*,in]* :: Field metadata
- **sx** *[integer,in]* :: Stride in *x* dimension
- **sy** *[integer,in]* :: Stride in *y* dimension

---

subroutine zFORp_field_set_stride_3d(*field*, *sx*, *sy*, *sz*)

Wrapper for `zfp_field_set_stride_3d()`

Parameters

- **field** *[*zFORp_field*,in]* :: Field metadata
- **sx** *[integer,in]* :: Stride in *x* dimension
- **sy** *[integer,in]* :: Stride in *y* dimension
- **sz** *[integer,in]* :: Stride in *z* dimension

---

subroutine zFORp_field_set_stride_4d(*field*, *sx*, *sy*, *sz*, *sw*)

Wrapper for `zfp_field_set_stride_4d()`

Parameters

- **field** *[*zFORp_field*,in]* :: Field metadata
- **sx** *[integer,in]* :: Stride in *x* dimension
- **sy** *[integer,in]* :: Stride in *y* dimension
- **sz** *[integer,in]* :: Stride in *z* dimension
- **sw** *[integer,in]* :: Stride in *w* dimension

---

function `zFORp_field_set_metadata`(*field*, *encoded_metadata*)

Wrapper for `zfp_field_set_metadata()`

Parameters

- **field** *[zFORp_field,in]* :: Field metadata
- **encoded_metadata** *[integer (kind=8),in]* :: Compact encoding of metadata

Return

is_success *[integer]* :: Indicate whether metadata was successfully set (1) or not (0)

11.3.7 Compression and Decompression

function `zFORp_compress`(*stream*, *field*)

Wrapper for `zfp_compress()`

Parameters

- **stream** *[zFORp_stream,in]* :: Compressed stream
- **field** *[zFORp_field,in]* :: Field metadata

Return

bitstream_offset_bytes *[integer (kind=8)]* :: Bit stream offset after compression, in bytes, or zero on failure

---

function `zFORp_decompress`(*stream*, *field*)

Wrapper for `zfp_decompress()`

Parameters

- **stream** *[zFORp_stream,in]* :: Compressed stream
- **field** *[zFORp_field,in]* :: Field metadata

Return

bitstream_offset_bytes *[integer (kind=8)]* :: Bit stream offset after decompression, in bytes, or zero on failure

---

function `zFORp_write_header`(*stream*, *field*, *mask*)

Wrapper for `zfp_write_header()`

Parameters

- **stream** *[zFORp_stream,in]* :: Compressed stream
- **field** *[zFORp_field,in]* :: Field metadata
- **mask** *[integer,in]* :: *Bit mask* indicating which parts of header to write

Return

num_bits_written *[integer (kind=8)]* :: Number of header bits written or zero on failure

---

function zFORp_read_header(*stream*, *field*, *mask*)

Wrapper for *zfp_read_header()*

Parameters

- **stream** *[zFORp_stream,in]* :: Compressed stream
- **field** *[zFORp_field,in]* :: Field metadata
- **mask** *[integer,in]* :: *Bit mask* indicating which parts of header to read

Return

num_bits_read *[integer (kind=8)]* :: Number of header bits read or zero on failure

CHAPTER

TWELVE

COMPRESSED-ARRAY C++ CLASSES

zfp's compressed arrays are C++ classes, plus *C wrappers* around these classes, that implement random-accessible single- and multi-dimensional floating-point arrays. Since its first release, zfp provides *fixed-rate* arrays, `zfp::array`, that support both read and write access to individual array elements. As of 1.0.0, zfp also supports read-only arrays, `zfp::const_array`, for data that is static or is updated only infrequently. The read-only arrays support all of zfp's *compression modes* including variable-rate and lossless compression.

For fixed-rate arrays, the 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). zfp supports 1D, 2D, 3D, and (as of version 1.0.0) 4D arrays. For higher-dimensional arrays, consider using an array of zfp arrays. Note that array dimensions need not be multiples of four; zfp transparently handles partial blocks on array boundaries.

Read-only arrays allow setting compression mode and parameters on construction, and can optionally be initialized with uncompressed data. These arrays do not allow updating individual array elements, though the contents of the whole array may be updated by re-compressing and overwriting the array. This may be useful in applications that decompress the whole array, perform a computation that updates its contents (e.g., a stencil operation that advances the solution of a PDE), and then compress to memory the updated array.

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, iterators, and views.

The following sections are available:

- *Read-Write Fixed-Rate Arrays*
- *Read-Only Variable-Rate Arrays*
- *Caching*
- *Serialization*
- *References*
- *Pointers*

12.1 Read-Write Fixed-Rate Arrays

There are eight array classes for 1D, 2D, 3D, and 4D read-write 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.

Note: Much of the compressed-array API was modified in zfp 1.0.0 to support 64-bit indexing of very large arrays. In particular, array dimensions and indices now use the `size_t` type instead of `uint` and strides use the `ptrdiff_t` type instead of `int`.

12.1.1 Base Class

class **`array`**

Virtual base class for common array functionality.

---

zfp_type *`array`*`::`**`scalar_type`**`()` const

Return the underlying scalar type (*`zfp_type`*) of the array.

---

uint *`array`*`::`**`dimensionality`**`()` const

Return the dimensionality (aka. rank) of the array: 1, 2, 3, or 4.

---

array::*header* *`array`*`::`**`get_header`**`()` const

Deprecated function as of zfp 1.0.0. See the *Header* section on how to construct a header.

---

static *array* **`array`*`::`**`construct`**(const *header* &h, const void *buffer = 0, size_t buffer_size_bytes = 0)

Construct a compressed-array object whose scalar type, dimensions, and rate are given by the *header* *h*. Return a base class pointer upon success. The optional *buffer* points to compressed data that, when passed, is copied into the array. If *buffer* is absent, the array is default initialized with all zeroes. The optional *buffer_size_bytes* parameter specifies the buffer length in bytes. When passed, a comparison is made to ensure that the buffer size is at least as large as the size implied by the header. If this function fails for any reason, an *`exception`* is thrown.

12.1.2 Common Methods

The following methods are common to 1D, 2D, 3D, and 4D 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.

---

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.

---

size_t *array*::**size_bytes**(uint mask = ZFP_DATA_ALL) const

Return storage size of components of array data structure indicated by *mask*. The mask is constructed via bitwise OR of *predefined constants*. Available as of zfp 1.0.0.

---

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()`.

---

void **array*::**compressed_data**() const

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

Note: As of zfp 1.0.0, the return value is `void*` rather than `uchar*` to simplify pointer conversion and to dispel any misconception that the compressed data needs only `uchar` alignment. Compressed streams are always word aligned (see `stream_word_bits` and `BIT_STREAM_WORD_TYPE`).

---

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

Return the cache size in number of bytes.

---

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

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

---

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.

---

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*, *y* varying faster than *z*, etc.

---

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. If *p* = 0, then the array is zero-initialized.

---

const_reference *array*::**operator[]**(size_t index) const

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

Note: As of zfp 1.0.0, the return value is no longer `Scalar` but is a *const reference* to the corresponding array element (conceptually equivalent to `const Scalar&`). This API change was necessary to allow obtaining a const pointer to the element when the array itself is const qualified, e.g., `const_pointer p = &a[index];`.

---

reference *array*::**operator[]**(size_t 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 random-access mutable iterator to beginning of array.

---

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

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

---

const_iterator *array*::**begin**() const

const_iterator *array*::**cbegin**() const

Return random-access const iterator to beginning of array.

const_iterator *array*::**end**() const

const_iterator *array*::**cend**() const

Return random-access const iterator to end of array.

Note: Const *references*, *pointers*, and *iterators* are available as of zfp 1.0.0.

12.1.3 1D, 2D, 3D, and 4D Arrays

Below are classes and methods specific to each array dimensionality and template scalar type (`float` or `double`). Since the classes and methods share obvious similarities regardless of dimensionality, only one generic description for all dimensionalities is provided.

Note: In the class declarations below, the class template for the scalar type is omitted for readability, e.g., `class array1` is used as shorthand for `template <typename Scalar> class array1`. Wherever the type `Scalar` appears, it refers to this template argument.

class **array1** : public *array*

class **array2** : public *array*

class **array3** : public *array*

class **array4** : public *array*

This is a 1D, 2D, 3D, or 4D 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>*.

---

class **arrayANY** : public *array*

Fictitious class used to refer to any one of *array1*, *array2*, *array3*, and *array4*. This class is not part of the zfp API.

---

array1::**array1**()

array2::**array2**()

array3::**array3**()

array4::**array4**()

Default constructor. Creates an empty array whose size and rate are both zero.

Note: The default constructor is useful when the array size or rate is not known at time of construction. Before the array can become usable, however, it must be *resized* and its rate must be set via *array::set_rate()*. These two tasks can be performed in either order. Furthermore, the desired cache size should be set using *array::set_cache_size()*, as the default constructor creates a cache that holds only one zfp block, i.e., the minimum possible.

---

array1::**array1**(size_t n, double rate, const Scalar *p = 0, size_t cache_size = 0) *array2*::**array2**(size_t nx, size_t ny, double rate, const Scalar *p = 0, size_t cache_size = 0)

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

array4::**array4**(size_t nx, size_t ny, size_t nz, size_t nw, double rate, const Scalar *p = 0, size_t cache_size = 0)

Constructor of array with dimensions n (1D), $nx \times ny$ (2D), $nx \times ny \times nz$ (3D), or $nx \times ny \times nz \times nw$ (4D) using *rate* bits per value, at least *cache_size* bytes of cache, and optionally initialized from flat, uncompressed array *p*. If *cache_size* is zero, a default cache size suitable for the array dimensions is chosen.

---

array1::**array1**(const *array*::*header* &h, const void *buffer = 0, size_t buffer_size_bytes = 0)

array2::**array2**(const *array*::*header* &h, const void *buffer = 0, size_t buffer_size_bytes = 0)

array3::**array3**(const *array*::*header* &h, const void *buffer = 0, size_t buffer_size_bytes = 0)

array4::**array4**(const *array*::*header* &h, const void *buffer = 0, size_t buffer_size_bytes = 0)

Constructor from previously *serialized* compressed array. The *header*, *h*, contains array metadata, while the optional *buffer* points to the compressed data that is to be copied to the array. The optional *buffer_size_bytes* parameter specifies the *buffer* length. If the constructor fails, an *exception* is thrown. See `array::construct()` for further details on the *buffer* and *buffer_size_bytes* parameters.

---

array1::**array1**(const *array1* &a)

array2::**array2**(const *array2* &a)

array3::**array3**(const *array3* &a)

array4::**array4**(const *array4* &a)

Copy constructor. Performs a deep copy.

---

virtual *array1*::**~array1**()

virtual *array2*::**~array2**()

virtual *array3*::**~array3**()

virtual *array4*::**~array4**()

Virtual destructor (allows for inheriting from zfp arrays).

---

array1 &*array1*::**operator=**(const *array1* &a)

array2 &*array2*::**operator=**(const *array2* &a)

array3 &*array3*::**operator=**(const *array3* &a)

array4 &*array4*::**operator=**(const *array4* &a)

Assignment operator. Performs a deep copy.

---

size_t *array2*::**size_x**() const

size_t *array2*::**size_y**() const

size_t *array3*::**size_x**() const

size_t *array3*::**size_y**() const

size_t *array3*::**size_z**() const

size_t *array4*::**size_x**() const

size_t *array4*::**size_y**() const

size_t *array4*::**size_z**() const

size_t *array4*::**size_w**() const

Return array dimensions.

---

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

void *array2*::**resize**(size_t nx, size_t ny, bool clear = true)

void *array3*::**resize**(size_t nx, size_t ny, size_t nz, bool clear = true)

void *array4*::**resize**(size_t nx, size_t ny, size_t nz, size_t nw, 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.

Note: It is often desirable (though not a requirement) to also set the cache size when resizing an array, e.g., in proportion to the array size; see `array::set_cache_size()`. This is particularly important when the array is default constructed, which initializes the cache size to the minimum possible of only one zfp block.

---

const_reference array1::**operator()**(size_t i) const

const_reference array2::**operator()**(size_t i, size_t j) const

const_reference array3::**operator()**(size_t i, size_t j, size_t k) const

const_reference array4::**operator()**(size_t i, size_t j, size_t k, size_t l) const

Return const reference to element stored at multi-dimensional index given by *i*, *j*, *k*, and *l* (inspector).

Note: As of zfp 1.0.0, the return value is no longer `Scalar` but is a *const reference* to the corresponding array element (essentially equivalent to `const Scalar&`). This API change was necessary to allow obtaining a const pointer to the element when the array itself is const qualified, e.g., `const_pointer p = &a(i, j, k);`.

---

reference array1::**operator()**(size_t i)

reference array2::**operator()**(size_t i, size_t j)

reference array3::**operator()**(size_t i, size_t j, size_t k)

reference array4::**operator()**(size_t i, size_t j, size_t k, size_t l)

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

12.2 Read-Only Variable-Rate Arrays

Read-only arrays are preferable in applications that store static data, e.g., constant tables or simulation output, or data that is updated only periodically as a whole, such as when advancing the solution of a partial differential equation. Because such updates have to be applied to the whole array, one may choose to tile large arrays into smaller zfp arrays to support finer granularity updates. Read-only arrays have the benefit of supporting all of zfp's *compression modes*, most of which provide higher accuracy per bit stored than fixed-rate mode.

The read-only arrays share an API with the read-write fixed-rate arrays, with only a few differences:

- All methods other than those that specify array-wide settings, such as compression mode and parameters, array dimensions, and array contents, are `const` qualified. There are, thus, no methods for obtaining a writeable reference, pointer, or iterator. Consequently, one may not initialize such arrays one element at a time. Rather, the user initializes the whole array by passing a pointer to uncompressed data.
- Whereas the constructors for fixed-rate arrays accept a *rate* parameter, the read-only arrays allow specifying any compression mode and corresponding parameters (if any) via a `zfp_config` object.
- Additional methods are available for setting and querying compression mode and parameters after construction.
- Read-only arrays are templated on a block index class that encodes the bit offset to each block of data. Multiple index classes are available that trade compactness and speed of access. The default `hybrid4` index represents 64-bit offsets using only 24 bits of amortized storage per block. An "implicit" index is available for fixed-rate read-only arrays, which computes rather than stores offsets to equal-sized blocks.

Note: Whereas variable-rate compression almost always improves accuracy per bit of compressed data over fixed rate, one should also weigh the storage and compute overhead associated with the block index needed for variable-rate storage. The actual storage overhead can be determined by passing `ZFP_DATA_INDEX` to `const_array::size_bytes()`. This overhead tends to be small for 3D and 4D arrays.

Array initialization may be done at construction time, by passing a pointer to uncompressed data, or via the method `const_array::set()`, which overwrites the contents of the whole array. This method may be called more than once to update (i.e., re-initialize) the array.

Read-only arrays support a subset of references, pointers, iterators, and views; in particular those with a `const_` prefix.

Currently, not all capabilities of read-write arrays are available for read-only arrays. For example, (de)serialization and construction from a view have not yet been implemented, and there are no C bindings.

Read-only arrays derive from the *array base class*. Additional methods are documented below.

class **`const_array1`** : public *array*

class **`const_array2`** : public *array*

class **`const_array3`** : public *array*

class **`const_array4`** : public *array*

1D, 2D, 3D, or 4D read-only 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., `const_array1f` is a synonym for `const_array1<float>`.

---

class **`const_array`** : public *array*

Fictitious class used to denote one of the 1D, 2D, 3D, and 4D read-only array classes. This pseudo base class serves only to document the API shared among the four arrays.

---

const_array1::**const_array1**()

const_array2::**const_array2**()

const_array3::**const_array3**()

const_array4::**const_array4**()

Default constructor. Creates an empty array whose size is zero and whose compression mode is unspecified. The array's cache size is initialized to the minimum possible, which can have performance implications; see *this note*.

---

const_array1::**const_array1**(size_t n, const zfp_config &config, const Scalar *p = 0, size_t cache_size = 0)

const_array2::**const_array2**(size_t nx, size_t ny, const zfp_config &config, const Scalar *p = 0, size_t cache_size = 0)

const_array3::**const_array3**(size_t nx, size_t ny, size_t nz, const zfp_config &config, const Scalar *p = 0, size_t cache_size = 0)

const_array4::**const_array4**(size_t nx, size_t ny, size_t nz, size_t nw, const zfp_config &config, const Scalar *p = 0, size_t cache_size = 0)

Constructor of array with dimensions n (1D), $nx \times ny$ (2D), $nx \times ny \times nz$ (3D), or $nx \times ny \times nz \times nw$ (4D). The compression mode and parameters are given by *config* (see *configuration*). The array uses at least *cache_size* bytes of cache, and is optionally initialized from flat, uncompressed array *p*. If *cache_size* is zero, a default cache size suitable for the array dimensions is chosen.

---

const_array1::**const_array1**(const *const_array1* &a)

const_array2::**const_array2**(const *const_array2* &a)

const_array3::**const_array3**(const *const_array3* &a)

const_array4::**const_array4**(const *const_array4* &a)

Copy constructor. Performs a deep copy.

---

virtual *const_array1*::**~const_array1**()

virtual *const_array2*::**~const_array2**()

virtual *const_array3*::**~const_array3**()

virtual *const_array4*::**~const_array4**()

Virtual destructor (allows for inheritance).

---

const_array1 &*const_array1*::**operator=**(const *const_array1* &a)

const_array2 &*const_array2*::**operator=**(const *const_array2* &a)

const_array3 &`const_array3::`**`operator=`**(const *const_array3* &a)

const_array4 &`const_array4::`**`operator=`**(const *const_array4* &a)

Assignment operator. Performs a deep copy.

---

size_t `const_array::`**`size`**() const

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

---

size_t `const_array2::`**`size_x`**() const

size_t `const_array2::`**`size_y`**() const

size_t `const_array3::`**`size_x`**() const

size_t `const_array3::`**`size_y`**() const

size_t `const_array3::`**`size_z`**() const

size_t `const_array4::`**`size_x`**() const

size_t `const_array4::`**`size_y`**() const

size_t `const_array4::`**`size_z`**() const

size_t `const_array4::`**`size_w`**() const

Return array dimensions.

---

void `const_array1::`**`resize`**(size_t n, bool clear = true)

void `const_array2::`**`resize`**(size_t nx, size_t ny, bool clear = true)

void `const_array3::`**`resize`**(size_t nx, size_t ny, size_t nz, bool clear = true)

void `const_array4::`**`resize`**(size_t nx, size_t ny, size_t nz, size_t nw, 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. See also *this note*.

---

zfp_mode `const_array::`**`mode`**() const

Currently selected *compression mode*. If not yet specified, `zfp_mode_null` is returned.

---

double `const_array::`**`rate`**() const

Return rate in compressed bits per value when *fixed-rate mode* is enabled, else zero.

---

uint `const_array::`**`precision`**() const

Return precision in uncompressed bits per value when *fixed-precision mode* is enabled, else zero.

---

double *const_array*::**accuracy**() const

Return accuracy as absolute error tolerance when *fixed-accuracy mode* is enabled, else zero.

---

void *const_array*::**params**(uint *minbits, uint *maxbits, uint *maxprec, int *minexp) const

Expert mode compression parameters (available for all compression modes). Pointers may be `null` if the corresponding parameter is not requested.

---

double *const_array*::**set_reversible**()

Enable *reversible mode*. This method destroys the previous contents of the array.

---

double *const_array*::**set_rate**(double rate)

Set desired rate in compressed bits per value (enables *fixed-rate mode*). This method destroys the previous contents of the array. See also `array::set_rate()`.

Note: Whereas the *read-write fixed-rate arrays* (`zfp::array`) require that block storage is word aligned, the read-only arrays (`zfp::const_array`) are not subject to such restrictions and therefore support finer rate granularity. For a d-dimensional `const_array`, the rate granularity is 4^{-d} bits/value, e.g., a quarter bit/value for 1D arrays.

---

uint *const_array*::**set_precision**(uint precision)

Set desired precision in uncompressed bits per value (enables *fixed-precision mode*). This method destroys the previous contents of the array.

---

double *const_array*::**set_accuracy**(double tolerance)

Set desired accuracy as absolute error tolerance (enables *fixed-accuracy mode*). This method destroys the previous contents of the array.

---

bool *const_array*::**set_params**(uint minbits, uint maxbits, uint maxprec, int minexp)

Set *expert mode* parameters. This method destroys the previous contents of the array. Return whether the codec supports the combination of parameters.

---

void *const_array*::**set_config**(const zfp_config &config)

Set compression mode and parameters given by *config* (see *configuration*). This is a more general method for setting compression parameters such as rate, precision, accuracy, and *expert mode* parameters.

---

size_t *const_array*::**size_bytes**(uint mask = ZFP_DATA_ALL) const

Return storage size of components of array data structure indicated by *mask*. The mask is constructed via bitwise OR of *predefined constants*.

---

size_t *const_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()*.

---

void **const_array*::**compressed_data**() const

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

---

size_t *const_array*::**cache_size**() const

Return the cache size in number of bytes.

---

void *const_array*::**set_cache_size**(size_t bytes)

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

---

void *const_array*::**clear_cache**() const

Empty cache.

---

void *const_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, y varying faster than z, etc.

---

void *const_array*::**set**(const Scalar *p, bool compact = true)

Initialize array by copying and compressing floating-point data stored at *p*. If $p = 0$, then the array is zero-initialized. The uncompressed data is assumed to be stored as in the *get()* method. Since the size of compressed data may not be known a priori, this method conservatively allocates enough space to hold it. If *compact* is true, any unused storage for compressed data is freed after initialization.

---

const_reference *const_array1*::**operator()**(size_t i) const

const_reference *const_array2*::**operator()**(size_t i, size_t j) const

const_reference *const_array3*::**operator()**(size_t i, size_t j, size_t k) const

const_reference *const_array4*::**operator()**(size_t i, size_t j, size_t k, size_t l) const

Return const reference to element stored at multi-dimensional index given by i, j, k, and l (inspector).

---

const_reference *const_array*::**operator[]**(size_t index) const

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

---

const_iterator *const_array*::**begin**() const

const_iterator *const_array*::**cbegin**() const

Return random-access const iterator to beginning of array.

---

const_iterator **end**() const

const_iterator **cend**() const

Return random-access const iterator to end of array.

12.3 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., $2 \times (nx / 4) \times (ny / 4)$ 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 *cache_size* parameter provided to the constructor is set to zero bytes, then a default cache size of at least $\sqrt{n}$ blocks is used, where n is the total number of blocks contained in the array.

The cache size can be set during construction, or can be set at a later time via *array::set_cache_size()*. Note that if *cache_size* = 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 *array::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*.

12.4 Serialization

zfp's read-write compressed arrays can be serialized to sequential, contiguous storage and later recovered back into an object, e.g., to support I/O of compressed-array objects. Two pieces of information are needed to describe a zfp array: the raw compressed data, obtained via *array::compressed_data()* and *array::compressed_size()*, and a *header* that describes the array scalar type, dimensions, and rate. The user may concatenate the header and compressed data to form a fixed-rate byte stream that can be read by the zfp *command-line tool*. When serializing the array, the user should first call *array::flush_cache()* before accessing the raw compressed data.

There are two primary ways to construct a compressed-array object from compressed data: via array-specific *constructors* and via a generic *factory function*:

- When the array scalar type (i.e., float or double) and dimensionality (i.e., 1D, 2D, 3D, or 4D) are already known, the corresponding array *constructor* may be used. If the scalar type and dimensionality stored in the header do not match the array class, then an *exception* is thrown.
- zfp provides a *factory function* that can be used when the serialized array type is unknown but described in the header. This function returns a pointer to the abstract base class, *array*, which the caller should dynamically cast to the corresponding derived array, e.g., by examining *array::scalar_type()* and *array::dimensionality()*.

  The (static) factory function is made available by including zfp/factory.hpp. This header must be included *after* first including the header files associated with the compressed arrays, i.e., zfp/array1.hpp, zfp/array2.hpp, zfp/array3.hpp, and zfp/array4.hpp. Only those arrays whose header files are included can be constructed by the factory function. This design decouples the array classes so that they may be included independently, for example, to reduce compilation time.

Both types of deserialization functions accept an *array::header*, an optional buffer holding compressed data, and an optional buffer size. If this buffer is provided, then a separate copy of the compressed data it holds is made, which is used to initialize the array. If the optional buffer size is also provided, then these functions throw an *exception* if the size is not at least as large as is expected from the metadata stored in the header. This safeguard is implemented to avoid accessing memory beyond the end of the buffer. If no buffer is provided, then all array elements are default initialized to zero. The array may later be initialized by directly reading/copying data into the space pointed to by *array::compressed_data()* and calling *array::clear_cache()* (in either order).

Below is a simple example of serialization of a 3D compressed array of doubles (error checking has been omitted for clarity):

```
zfp::array3d a(nx, ny, nz, rate);
...
a.flush_cache();
zfp::array::header h(a);
fwrite(h.data(), h.size_bytes(), 1, file);
fwrite(a.compressed_data(), a.compressed_size(), 1, file);
```

We may then deserialize this array using the factory function. The following example reads the compressed data directly into the array without making a copy:

```
zfp::array::header h;
fread(h.data(), h.size_bytes(), 1, file);
zfp::array* p = zfp::array::construct(h);
fread(p->compressed_data(), p->compressed_size(), 1, file);
assert(p->dimensionality() == 3 && p->scalar_type() == zfp_type_double);
zfp::array3d& a = *dynamic_cast<zfp::array3d*>(p);
```

When the array is no longer in use, call delete p; to deallocate it.

Note: The array serialization API changed significantly in zfp 1.0.0. The *array::get_header()* function is now deprecated and has been replaced with a *header constructor* that takes an array as parameter. Exceptions are now part of the main zfp namespace rather than nested within the array header. The header is no longer a simple POD data structure but should be queried for its data pointer and size.

12.4.1 Header

Short 12-byte headers are used to describe array metadata and compression parameters when serializing a compressed array. This header is the same as supported by the *zfp_read_header()* and *zfp_write_header()* functions, using *ZFP_HEADER_FULL* to indicate that complete metadata is to be stored in the header. The header is also compatible with the zfp *command-line tool*. Processing of the header may result in an *exception* being thrown.

Note: Compressed-array headers use zfp's most concise representation of only 96 bits. Such short headers support compressed blocks up to 2048 bits long. This implies that the highest rate for 3D arrays is $2048/4^3$ = 32 bits/value; the highest rate for 4D arrays is only $2048/4^4$ = 8 bits/value. 3D and 4D arrays whose rate exceeds these limits cannot be serialized and result in an exception being thrown. 1D and 2D arrays support rates up to 512 and 128 bits/value, respectively, which both are large enough to represent all usable rates.

---

class *array*::**header**

The header stores information such as scalar type, array dimensions, and compression parameters such as rate. Compressed-array headers are always 96 bits long.

---

header::**header**()

Default constructor for header.

---

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

Construct header for compressed-array *a*. Throws an *exception* upon failure.

---

header::**header**(const void *buffer, size_t bytes = 0)

Deserialize header from memory buffer given by *buffer* of optional size *bytes*. This memory buffer is obtained from an existing header during serialization via `header::data()` and `header::size_bytes()`. The constructor throws an *exception* upon failure.

---

zfp_type *header*::**scalar_type**() const

Scalar type associated with array (see *array::scalar_type()*).

---

uint *header*::**dimensionality**() const

Dimensionality associated with array (see *array::dimensionality()*).

---

size_t *header*::**size_x**() const

size_t *header*::**size_y**() const

size_t *header*::**size_z**() const

size_t *header*::**size_w**() const

Array dimensions. Unused dimensions have a size of zero.

---

double *header*::**rate**() const

Rate in bits per value (see *array::rate()*);

---

virtual const void **header*::**data**() const = 0

Return pointer to header data.

---

virtual size_t *header*::**size_bytes**(uint mask = ZFP_DATA_HEADER) const = 0

When *mask* = *ZFP_DATA_HEADER*, return header payload size in bytes pointed to by *header::data()*. Only those bytes are needed to (de)serialize a header. The header object stores additional (redundant) metadata whose size can be queried via *ZFP_DATA_META*.

12.4.2 Exceptions

class **exception** : public std::runtime_error

Compressed arrays may throw this exception upon serialization, when constructing a header via its *constructor*, or deserialization, when constructing a compressed array via its *constructor* or *factory function*. The `exception::what()` method returns a `std::string` error message that indicates the cause of the exception. Most error messages changed in zfp 1.0.0.

12.5 References

class *array1*::**const_reference**

class *array2*::**const_reference**

class *array3*::**const_reference**

class *array4*::**const_reference**

class *array1*::**reference** : public *array1*::*const_reference*

class *array2*::**reference** : public *array2*::*const_reference*

class *array3*::**reference** : public *array3*::*const_reference*

class *array4*::**reference** : public *array4*::*const_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.

zfp 1.0.0 adds `const` qualified versions of references, pointers, and iterators to support const correctness and potential performance improvements when only read access is needed. As with STL containers, the corresponding types are prefixed by `const_`, e.g., `const_reference`. The mutable versions of these classes inherit the read-only API from the corresponding const versions.

Only references into *read-write arrays* are discussed here; the *read-only arrays* support the same `const_reference` API.

Note: Do not confuse `const_reference` and `const reference`. The former is a reference to an immutable array element, while the latter means that the proxy reference object itself is immutable.

References define a single type:

type `reference::`**`value_type`**

type `const_reference::`**`value_type`**

Scalar type associated with referenced array elements.

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. References are obtained via *array inspectors* and *mutators*.

---

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

value_type `const_reference::`**`operator value_type`**`()` const

Conversion operator for dereferencing the reference. Return the value of the referenced array element.

---

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

const_pointer `const_reference::`**`operator&`**`()` const

Return (const) pointer to the referenced array element.

---

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`.

12.6 Pointers

class *array1*::**const_pointer**

class *array2*::**const_pointer**

class *array3*::**const_pointer**

class *array4*::**const_pointer**

class *array1*::**pointer** : public *array1*::*const_pointer*

class *array2*::**pointer** : public *array2*::*const_pointer*

class *array3*::**pointer** : public *array3*::*const_pointer*

class *array4*::**pointer** : public *array4*::*const_pointer*

Similar to *references*, zfp supports proxy pointers (also known as fancy 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 to one-dimensional arrays. 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 array into which it points. Like conventional pointers, proxy pointers can be passed to other functions and manipulated there, for instance, by passing the pointer by reference via `pointer&`.

As of zfp 1.0.0, const qualified pointers `const_pointer` are available, and conceptually are equivalent to `const Scalar*`. Pointers are available for *read-only arrays* also.

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

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

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

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

reference `pointer::operator*() const`

const_reference `const_pointer::operator*() const`

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

---

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

const_reference `const_pointer::operator[](ptrdiff_t d) const`

Offset dereference operator. Return proxy (const) reference to the value stored at `p[d]`.

---

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

const_pointer `const_pointer::operator+(ptrdiff_t d) const`

Return a copy of the pointer incremented by *d*.

---

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

const_pointer `const_pointer::operator-(ptrdiff_t d) const`

Return a copy of the pointer decremented by *d*.

---

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

ptrdiff_t `const_pointer::operator-(const const_pointer &q) const`

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

---

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

bool `const_pointer::operator==(const const_pointer &q) const`

Return true if *p* and *q* point to the same array element.

---

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

bool `const_pointer::operator!=(const const_pointer &q) const`

Return true if *p* and *q* do not point to the same array element. This operator returns false if *p* and *q* do not point into the same array.

---

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

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

bool `pointer::operator<(const pointer &q) const`

bool `pointer::operator>(const pointer &q) const`

bool `const_pointer::operator<=(const const_pointer &q) const` bool `const_pointer::`**`operator>=`**`(const const_pointer &q) const`

bool `const_pointer::`**`operator<`**`(const const_pointer &q) const`

bool `const_pointer::`**`operator>`**`(const const_pointer &q) const`

Return true if the two pointers satisfy the given relationship. These operators return false if *p* and *q* do not point into the same array.

---

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

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

Prefix increment pointer, i.e., `++p`. Return reference to the incremented pointer.

---

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

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

Prefix decrement pointer, i.e., `--p`. Return reference to the decremented pointer.

---

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

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

Postfix increment pointer, i.e., `p++`. Return a copy of the pointer before it was incremented.

---

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

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

Postfix decrement pointer, i.e., `p--`. Return a copy of the pointer before it was decremented.

---

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

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

Increment pointer by *d*. Return a copy of the incremented pointer.

---

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

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

Decrement pointer by *d*. Return a copy of the decremented pointer.

12.7 Iterators

class *array1*::**const_iterator**

class *array2*::**const_iterator**

class *array3*::**const_iterator**

class *array4*::**const_iterator**

class *array1*::**iterator** : public *array1*::*const_iterator*

class *array2*::**iterator** : public *array2*::*const_iterator*

class *array3*::**iterator** : public *array3*::*const_iterator*

class *array4*::**iterator** : public *array4*::*const_iterator*

Iterators provide a mechanism for traversing a possibly multi-dimensional array—or a *view* of a subset of an 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 sequentially 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 compression 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*.

Blocks are visited in raster order similarly to how individual array elements are indexed, that is, first by x, then by y, then by z, etc. Within each block, elements are visited in the same raster order. All 4^d values in a block are visited before moving on to the next block (see Fig. 12.1).

As of zfp 1.0.0, all iterators provided by zfp are random access iterators (previously, multi-dimensional array iterators were only forward iterators). zfp iterators are STL compliant and can be used in STL algorithms that support random access iterators.

zfp 1.0.0 adds `const` qualified versions of iterators, given by the `const_iterator` class. Such iterators are available also for *read-only arrays*.

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`**

The *reference* type associated with the iterator's parent array class.

type `iterator::`**`pointer`**

The *pointer* type associated with the iterator's parent array class.

type `iterator::`**`iterator_category`**

Type of iterator: `std::random_access_iterator_tag`.

For const iterators, the following additional types are defined:

type `const_iterator::`**`const_reference`**

The immutable reference type associated with the iterator's container class.

type `const_iterator::`**`const_pointer`**

The immutable pointer type associated with the iterator's container class.

The following operations are defined on iterators:

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

const_iterator `const_iterator::`**`operator=`**`(`const const_iterator &it`)`

Assignment (copy) operator. Make the iterator point to the same element as *it*.

---

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

const_reference `const_iterator::`**`operator*`**`()` const

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

---

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

const_reference `const_iterator::`**`operator[]`**`(`difference_type d`)` const

Offset dereference operator. Return (const) reference to the value *d* elements relative to the current element in the iteration sequence (*d* may be negative). This operator executes in constant time regardless of array dimensionality but is more costly than sequential iteration via *`iterator::operator++()`*.

---

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

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

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

---

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

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

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

---

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

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

Return difference between this iterator and *it* in number of elements. The difference $p - q$ between two iterators, p and q, is negative if $p < q$. The iterators must refer to elements in the same array.

---

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

Return true if the two iterators point to the same element.

---

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

bool `const_iterator::operator!=`(const const_iterator &it) const

Return true if the two iterators do not point to the same element.

---

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

bool `const_iterator::operator<=`(const const_iterator &it) const

bool `const_iterator::operator>=`(const const_iterator &it) const

bool `const_iterator::operator<`(const const_iterator &it) const

bool `const_iterator::operator>`(const 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.

---

iterator &`iterator::operator++`()

const_iterator &`const_iterator::operator++`()

Prefix increment (`++it`). Return a reference to the incremented iterator.

---

iterator `iterator::operator++`(int)

const_iterator `const_iterator::operator++`(int)

Postfix increment (`it++`). Return the value of the iterator before being incremented.

---

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

const_iterator &`const_iterator::operator--`()

Prefix decrement (`--it`). Return a reference to the decremented iterator.

---

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

const_iterator `const_iterator::operator--`(int)

Postfix decrement (`it--`). Return the value of the iterator before being decremented.

---

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

const_iterator `const_iterator::operator+=`(difference_type d)

Increment iterator *d* times. Return value of incremented iterator. Although `++it` and `it += 1` are semantically equivalent, the former is more efficient for multidimensional arrays.

---

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

const_iterator `const_iterator::operator-=`(difference_type d)

Decrement iterator *d* times. Return value of decremented iterator. Although `--it` and `it -= 1` are semantically equivalent, the former is more efficient for multidimensional arrays.

---

size_t `iterator::i`() const

size_t `iterator::j`() const

size_t `iterator::k`() const

size_t `iterator::l`() const

size_t `const_iterator::i`() const

size_t `const_iterator::j`() const

size_t `const_iterator::k`() const

size_t `const_iterator::l`() const

Return array index or local view index of element pointed to by the iterator. `iterator::i()` is defined for all arrays. `iterator::j()` is defined only for 2D, 3D, and 4D arrays. `iterator::k()` is defined only for 3D and 4D arrays. `iterator::l()` is defined only for 4D arrays.

12.8 Views

zfp 0.5.4 adds array views. Much like how *references* allow indirect access to single array elements, *views* provide indirect access to whole arrays, or more generally to rectangular subsets of arrays. A view of an array does not allocate any storage for the array elements. Rather, the view accesses shared storage managed by the underlying array. This allows for multiple entries into an array without the need for expensive deep copies. In a sense, views can be thought of as *shallow copies* of arrays.

When a view exposes a whole array `array<type>`, it provides similar functionality to a C++ reference `array<type>&` or pointer `array<type>*` to the array. However, views are more general in that they also allow restricting access to a user-specified subset of the array, and unlike pointers also provide for the same syntax when accessing the array, e.g., `array_view(i, j)` instead of `(*array_ptr)(i, j)`.

zfp's *nested views* further provide for multidimensional array access analogous to the C/C++ nested array syntax `array[i][j]`. Finally, zfp's *private views* can be used to ensure thread-safe access to its compressed arrays.

Access to array elements through a view is via inspectors and mutators that return a `const_reference` or `reference`, respectively (see *References*). As of zfp 1.0.0, it is also possible to obtain pointers to array elements through views and to iterate over them. View pointers and iterators allow referencing only the elements visible through the view, e.g., a rectangular subset of an array (see Fig. 12.1). Those elements are indexed as if the view were a contiguous array, and pointer arithmetic assumes that the possibly smaller view and not the underlying array is flattened. *Private views* maintain their own cache and therefore implement their own proxy references, pointers, and iterators.

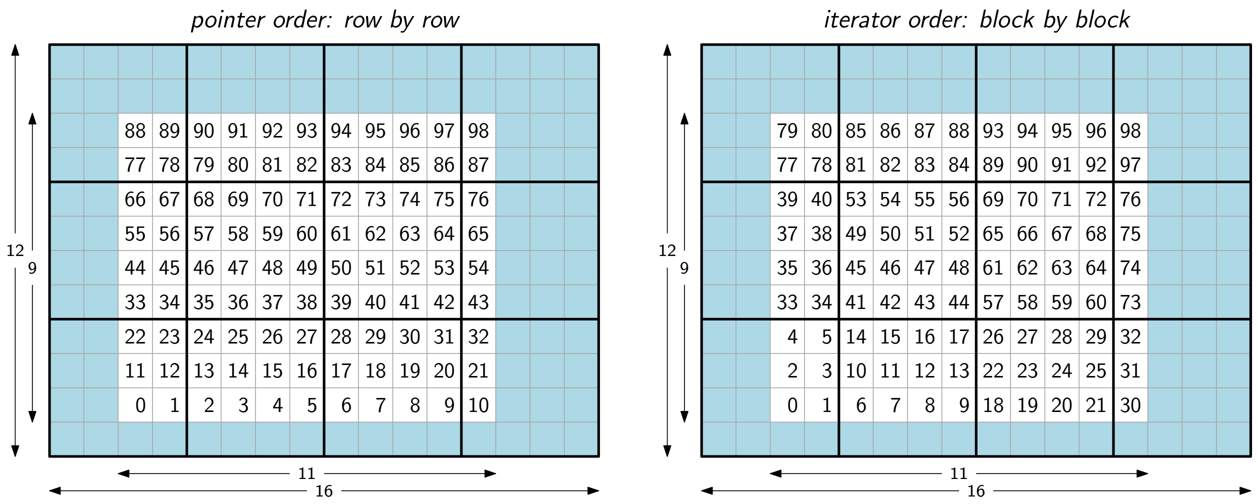


Fig. 12.1: An 11 × 9 element view of a 2D array of dimensions 16 × 12. The numbered elements indicate the order in which the view is traversed using pointers and iterators. We have `view(10, 7) == (&view(0, 0))[87] == view.begin()[97] == view.end()[-2]`.

With the zfp 1.0.0 release of *read-only arrays*, such arrays also support the two kinds of immutable views (`const_view` and `private_const_view`). The documentation below applies to views into read-only arrays as well.

Note: Like iterators and proxy references and pointers, a view is valid only during the lifetime of the array that it references. **No reference counting** is done to keep the array alive. It is up to the user to ensure that the referenced array object is valid when accessed through a view.

There are several types of views distinguished by these attributes:

- Read-only vs. read-write access.
- Shared vs. private access.
- Flat vs. nested indexing.

Each of these attributes is discussed in detail below in these sections:

- *Immutable view*
- *Mutable view*
- *Flat view*
- *Nested view*
- *Slicing*
- *Private immutable view*
- *Private mutable view*

12.8.1 Immutable view

The most basic view is the immutable `const_view`, which supports read-only access to the array elements it references. This view serves primarily as a base class for more specialized views. Its constructors allow establishing access to a whole array or to a rectangular subset of an array. Note that like references, pointers, and iterators, views are types nested within the arrays that they reference.

class *array1*::**const_view**

class *array2*::**const_view**

class *array3*::**const_view**

class *array4*::**const_view**

Immutable view into 1D, 2D, 3D, and 4D array.

---

array1::*const_view*::**const_view**(*array1* *array)

array2::*const_view*::**const_view**(*array2* *array)

array3::*const_view*::**const_view**(*array3* *array)

array4::*const_view*::**const_view**(*array4* *array)

Constructor for read-only access to a whole array. As already mentioned, these views are valid only during the lifetime of the underlying array object.

---

array1::*const_view*::**const_view**(*array1* *array, size_t x, size_t nx)

array2::*const_view*::**const_view**(*array2* *array, size_t x, size_t y, size_t nx, size_t ny)

array3::*const_view*::**const_view**(*array3* *array, size_t x, size_t y, size_t z, size_t nx, size_t ny, size_t nz)

array4::*const_view*::**const_view**(*array4* *array, size_t x, size_t y, size_t z, size_t w, size_t nx, size_t ny, size_t nz, size_t nw)

Constructors for read-only access to a rectangular subset of an array. The subset is specified by an offset, e.g., (*x*, *y*, *z*) for a 3D array, and dimensions, e.g., (*nx*, *ny*, *nz*) for a 3D array. The rectangle must fit within the surrounding array.

---

size_t *array1*::*const_view*::**global_x**(size_t i) const

size_t *array2*::*const_view*::**global_x**(size_t i) const

size_t *array2*::*const_view*::**global_y**(size_t j) const

size_t *array3*::*const_view*::**global_x**(size_t i) const

size_t *array3*::*const_view*::**global_y**(size_t j) const

size_t *array3*::*const_view*::**global_z**(size_t k) const

size_t *array4*::*const_view*::**global_x**(size_t i) const

size_t *array4*::*const_view*::**global_y**(size_t j) const size_t *array4*::*const_view*::**global_z**(size_t k) const

size_t *array4*::*const_view*::**global_w**(size_t l) const

Return global array index associated with local view index. For instance, if a 1D view has been constructed with offset x, then `global_x(i)` returns $x + i$.

---

size_t *array1*::*const_view*::**size_x**() const

size_t *array2*::*const_view*::**size_x**() const

size_t *array2*::*const_view*::**size_y**() const

size_t *array3*::*const_view*::**size_x**() const

size_t *array3*::*const_view*::**size_y**() const

size_t *array3*::*const_view*::**size_z**() const

size_t *array4*::*const_view*::**size_x**() const

size_t *array4*::*const_view*::**size_y**() const

size_t *array4*::*const_view*::**size_z**() const

size_t *array4*::*const_view*::**size_w**() const

Return dimensions of view.

---

const_reference array1::*const_view*::**operator()**(size_t i) const

const_reference array2::*const_view*::**operator()**(size_t i, size_t j) const

const_reference array3::*const_view*::**operator()**(size_t i, size_t j, size_t k) const

const_reference array4::*const_view*::**operator()**(size_t i, size_t j, size_t k, size_t l) const

Return reference to scalar stored at multi-dimensional index given by $x + i$, $y + j$, $z + k$, and $w + l$, where x, y, z, and w specify the offset into the array.

---

const_reference array1::*const_view*::**operator[]**(size_t index) const

Alternative inspector for 1D arrays identical to *array1::const_view::operator()()*.

---

array1::*const_view*::const_iterator array1::*const_view*::**begin**() const

array2::*const_view*::const_iterator array2::*const_view*::**begin**() const

array3::*const_view*::const_iterator array3::*const_view*::**begin**() const

array4::*const_view*::const_iterator array4::*const_view*::**begin**() const

array1::*const_view*::const_iterator array1::*const_view*::**cbegin**() const

array2::*const_view*::const_iterator array2::*const_view*::**cbegin**() const *array3*::*const_view*::const_iterator `array3::const_view::`**`cbegin`**() const

array4::*const_view*::const_iterator `array4::const_view::`**`cbegin`**() const

Random-access const iterator to first element of view.

---

array1::*const_view*::const_iterator `array1::const_view::`**`end`**() const

array2::*const_view*::const_iterator `array2::const_view::`**`end`**() const

array3::*const_view*::const_iterator `array3::const_view::`**`end`**() const

array4::*const_view*::const_iterator `array4::const_view::`**`end`**() const

array1::*const_view*::const_iterator `array1::const_view::`**`cend`**() const

array2::*const_view*::const_iterator `array2::const_view::`**`cend`**() const

array3::*const_view*::const_iterator `array3::const_view::`**`cend`**() const

array4::*const_view*::const_iterator `array4::const_view::`**`cend`**() const

Random-access const iterator to end of view.

There are a number of common methods inherited from a base class, `preview`, further up the class hierarchy.

double `arrayANY::const_view::`**`rate`**() const

Return rate in bits per value. Same as `array::rate()`.

---

size_t `arrayANY::const_view::`**`size`**() const

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

With the above definitions, the following example shows how a 2D view is constructed and accessed:

```
zfp::array2d a(200, 100, rate); // define 200x100 array of doubles
zfp::array2d::const_view v(&a, 10, 5, 20, 20); // v is a 20x20 view into array a
assert(v(2, 1) == a(12, 6)); // v(2, 1) == a(10 + 2, 5 + 1) == a(12, 6)
assert(v.size() == 400); // 20x20 == 400
```

12.8.2 Mutable view

The basic mutable `view` derives from the `const_view` but adds operators for write-access. Its constructors are similar to those for the `const_view`.

class `array1::`**`view`** : public *array1*::*const_view*

class `array2::`**`view`** : public *array2*::*const_view*

class `array3::`**`view`** : public *array3*::*const_view*

class `array4::`**`view`** : public *array4*::*const_view*

Mutable view into 1D, 2D, 3D, and 4D array.

`array1::view::`**`view`**(*array1* *array)

`array2::view::view`(*array2* *array)

`array3::view::view`(*array3* *array)

`array4::view::view`(*array4* *array)

`array1::view::view`(*array1* *array, size_t x, size_t nx)

`array2::view::view`(*array2* *array, size_t x, size_t y, size_t nx, size_t ny)

`array3::view::view`(*array3* *array, size_t x, size_t y, size_t z, size_t nx, size_t ny, size_t nz)

`array4::view::view`(*array4* *array, size_t x, size_t y, size_t z, size_t w, size_t nx, size_t ny, size_t nz, size_t nw)

Whole-array and sub-array mutable view constructors. See *const_view constructors* for details.

---

reference `array1::view::operator()`(size_t i)

reference `array2::view::operator()`(size_t i, size_t j)

reference `array3::view::operator()`(size_t i, size_t j, size_t k)

reference `array4::view::operator()`(size_t i, size_t j, size_t k, size_t l)

These operators, whose arguments have the same meaning as in the *array accessors*, return *proxy references* to individual array elements for write access.

12.8.3 Flat view

The views discussed so far require multidimensional indexing, e.g., (i, j, k) for 3D views. Some applications prefer one-dimensional linear indexing, which is provided by the specialized flat view. For example, in a 3D view with dimensions (nx, ny, nz), a multidimensional index (i, j, k) corresponds to the flat view index

```
index = i + nx * (j + ny * k)
```

This is true regardless of the view offset (x, y, z).

The flat view derives from the mutable view and adds `operator[]` for flat indexing. This operator is essentially equivalent to `array::operator[]()` defined for 2D, 3D, and 4D arrays. Flat views also provide functions for converting between multidimensional and flat indices.

Flat views are available only for 2D, 3D, and 4D arrays. The basic mutable view, `array1::view`, for 1D arrays can be thought of as either a flat or a nested view.

class `array2::flat_view` : public *array2*::*view*

class `array3::flat_view` : public *array3*::*view*

class `array4::flat_view` : public *array4*::*view*

Flat, mutable views for 2D, 3D, and 4D arrays.

`array2::flat_view::flat_view`(*array2* *array)

`array3::flat_view::flat_view`(*array3* *array)

`array4::flat_view::flat_view`(*array4* *array)

array2::*flat_view*::**flat_view**(*array2* *array, size_t x, size_t y, size_t nx, size_t ny)

array3::*flat_view*::**flat_view**(*array3* *array, size_t x, size_t y, size_t z, size_t nx, size_t ny, size_t nz)

array4::*flat_view*::**flat_view**(*array4* *array, size_t x, size_t y, size_t z, size_t w, size_t nx, size_t ny, size_t nz, size_t nw)

Whole-array and sub-array flat view constructors. See *const_view constructors* for details.

---

size_t *array2*::*flat_view*::**index**(size_t i, size_t j) const

size_t *array3*::*flat_view*::**index**(size_t i, size_t j, size_t k) const

size_t *array4*::*flat_view*::**index**(size_t i, size_t j, size_t k, size_t l) const

Return flat index associated with multidimensional index.

---

void *array2*::*flat_view*::**ij**(size_t &i, size_t &j, size_t index) const

void *array3*::*flat_view*::**ijk**(size_t &i, size_t &j, size_t &k, size_t index) const

void *array4*::*flat_view*::**ijkl**(size_t &i, size_t &j, size_t &k, size_t &l, size_t index) const

Convert flat index to multidimensional index.

---

const_reference array2::*flat_view*::**operator[]**(size_t index) const

const_reference array3::*flat_view*::**operator[]**(size_t index) const

const_reference array4::*flat_view*::**operator[]**(size_t index) const

Return array element associated with given flat index.

---

reference array2::*flat_view*::**operator[]**(size_t index)

reference array3::*flat_view*::**operator[]**(size_t index)

reference array4::*flat_view*::**operator[]**(size_t index)

Return reference to array element associated with given flat index.

12.8.4 Nested view

C and C++ support nested arrays (arrays of arrays), e.g., `double a[10][20][30]`, which are usually accessed via nested indexing `a[i][j][k]`. Here `a` is a 3D array, `a[i]` is a 2D array, and `a[i][j]` is a 1D array. This 3D array can also be accessed via flat indexing, e.g.,

```
a[i][j][k] == (&a[0][0][0])[600 * i + 30 * j + k]
```

Nested views provide a mechanism to access array elements through nested indexing and to extract lower-dimensional "slices" of multidimensional arrays. Nested views are mutable.

Nested views are associated with a dimensionality. For instance, if `v` is a 3D nested view of a 3D array, then `v[i]` is a 2D nested view (of a 3D array), `v[i][j]` is a 1D nested view (of a 3D array), and `v[i][j][k]` is a (reference to a) scalar array element. Note that the order of indices is reversed when using nested indexing compared to multidimensional indexing, e.g., `v(i, j, k) == v[k][j][i]`.

Whereas `operator[]` on an array object accesses an element through flat indexing, the same array can be accessed through a nested view to in effect provide nested array indexing:

```
zfp::array3d a(30, 20, 10, rate); // define 30x20x10 3D array
assert(a[32] == a(2, 1, 0)); // OK: flat and multidimensional indexing
assert(a[32] == a[0][1][2]); // ERROR: a does not support nested indexing
zfp::array3d::nested_view v(&a); // define a nested view of a
assert(a[32] == v[0][1][2]); // OK: v supports nested indexing
zfp::array2d b(v[5]); // define and deep copy 30x20 2D slice of a
assert(a(2, 1, 5) == b(2, 1)); // OK: multidimensional indexing
```

class *array2*::**nested_view1**

View of a 1D slice of a 2D array.

class *array2*::**nested_view2**

2D view of a 2D (sub)array.

class *array3*::**nested_view1**

View of a 1D slice of a 3D array.

class *array3*::**nested_view2**

View of a 2D slice of a 3D array.

class *array3*::**nested_view3**

3D view of a 3D (sub)array.

class *array4*::**nested_view1**

View of a 1D slice of a 4D array.

class *array4*::**nested_view2**

View of a 2D slice of a 4D array.

class *array4*::**nested_view3**

View of a 3D slice of a 4D array.

class *array4*::**nested_view4**

4D view of a 4D (sub)array.

array2::*nested_view2*::**nested_view2**(*array2* *array)

array3::*nested_view3*::**nested_view3**(*array3* *array)

array4::*nested_view4*::**nested_view4**(*array4* *array)

array2::*nested_view2*::**nested_view2**(*array2* *array, size_t x, size_t y, size_t nx, size_t ny)

array3::*nested_view3*::**nested_view3**(*array3* *array, size_t x, size_t y, size_t z, size_t nx, size_t ny, size_t nz)

array4::*nested_view4*::**nested_view4**(*array4* *array, size_t x, size_t y, size_t z, size_t w, size_t nx, size_t ny, size_t nz, size_t nw)

Whole-array and sub-array nested view constructors. See *const_view* constructors for details. Lower-dimensional view constructors are not accessible to the user but are invoked when accessing views via nested indexing.

---

size_t *array2*::*nested_view1*::**size_x**() const

size_t *array2*::*nested_view2*::**size_x**() const

size_t *array2*::*nested_view2*::**size_y**() const

size_t *array3*::*nested_view1*::**size_x**() const

size_t *array3*::*nested_view2*::**size_x**() const

size_t *array3*::*nested_view2*::**size_y**() const

size_t *array3*::*nested_view3*::**size_x**() const

size_t *array3*::*nested_view3*::**size_y**() const

size_t *array3*::*nested_view3*::**size_z**() const

size_t *array4*::*nested_view1*::**size_x**() const

size_t *array4*::*nested_view2*::**size_x**() const

size_t *array4*::*nested_view2*::**size_y**() const

size_t *array4*::*nested_view3*::**size_x**() const

size_t *array4*::*nested_view3*::**size_y**() const

size_t *array4*::*nested_view3*::**size_z**() const

size_t *array4*::*nested_view4*::**size_x**() const

size_t *array4*::*nested_view4*::**size_y**() const

size_t *array4*::*nested_view4*::**size_z**() const

size_t *array4*::*nested_view4*::**size_w**() const

View dimensions.

---

array4::*nested_view3* `array4::nested_view4::`**`operator[]`**(size_t index) const

Return view to a 3D slice of 4D array.

---

array3::*nested_view2* `array3::nested_view3::`**`operator[]`**(size_t index) const

array4::*nested_view2* `array4::nested_view3::`**`operator[]`**(size_t index) const

Return view to a 2D slice of a 3D or 4D array.

---

array2::*nested_view1* `array2::nested_view2::`**`operator[]`**(size_t index) const

array3::*nested_view1* `array3::nested_view2::`**`operator[]`**(size_t index) const

array4::*nested_view1* `array4::nested_view2::`**`operator[]`**(size_t index) const

Return view to a 1D slice of a 2D, 3D, or 4D array.

---

const_reference `array2::nested_view1::`**`operator[]`**(size_t index) const

const_reference `array3::nested_view1::`**`operator[]`**(size_t index) const

const_reference `array4::nested_view1::`**`operator[]`**(size_t index) const

Return scalar element of a 2D, 3D, or 4D array.

---

reference `array2::nested_view1::`**`operator[]`**(size_t index)

reference `array3::nested_view1::`**`operator[]`**(size_t index)

reference `array4::nested_view1::`**`operator[]`**(size_t index)

Return reference to a scalar element of a 2D, 3D, or 4D array.

---

const_reference `array2::nested_view1::`**`operator()`**(size_t i) const

const_reference `array2::nested_view2::`**`operator()`**(size_t i, size_t j) const

const_reference `array3::nested_view1::`**`operator()`**(size_t i) const

const_reference `array3::nested_view2::`**`operator()`**(size_t i, size_t j) const

const_reference `array3::nested_view3::`**`operator()`**(size_t i, size_t j, size_t k) const

const_reference `array4::nested_view1::`**`operator()`**(size_t i) const

const_reference `array4::nested_view2::`**`operator()`**(size_t i, size_t j) const

const_reference `array4::nested_view3::`**`operator()`**(size_t i, size_t j, size_t k) const

const_reference `array4::nested_view4::`**`operator()`**(size_t i, size_t j, size_t k, size_t l) const

Return const reference to a scalar element of a 2D, 3D, or 4D array.

---

reference array2::nested_view1::**operator()**(size_t i)

reference array2::nested_view2::**operator()**(size_t i, size_t j)

reference array3::nested_view1::**operator()**(size_t i)

reference array3::nested_view2::**operator()**(size_t i, size_t j)

reference array3::nested_view3::**operator()**(size_t i, size_t j, size_t k)

reference array4::nested_view1::**operator()**(size_t i)

reference array4::nested_view2::**operator()**(size_t i, size_t j)

reference array4::nested_view3::**operator()**(size_t i, size_t j, size_t k)

reference array4::nested_view4::**operator()**(size_t i, size_t j, size_t k, size_t l)

Return reference to a scalar element of a 2D, 3D, or 4D array.

12.8.5 Slicing

Arrays can be constructed as deep copies of slices of higher-dimensional arrays, as the code example above shows (i.e., `zfp::array2d b(v[5]);`). Unlike views, which have reference semantics, such array *slicing* has value semantics. In this example, 2D array *b* is initialized as a (deep) copy of a slice of 3D array *a* via nested view *v*. Subsequent modifications of *b* have no effect on *a*.

Slicing is implemented as array constructors templated on views. Upon initialization, elements are copied one at a time from the view via multidimensional indexing, e.g., `v(i, j, k)`. Note that view and array dimensionalities must match, but aside from this an array may be constructed from any view.

Slicing needs not change the dimensionality, but can be used to copy an equidimensional subset of one array to another array, as in this example:

```
zfp::array3d a(30, 20, 10, rate);
zfp::array3d::const_view v(&a, 1, 2, 3, 4, 5, 6);
zfp::array3d b(v);
assert(b(0, 0, 0) == a(1, 2, 3));
assert(b.size_x() == 4);
assert(b.size_y() == 5);
assert(b.size_z() == 6);
```

Slicing adds the following templated array constructors.

template<class **View**>
array1::**array1**(const *View* &v)

template<class **View**>
array2::**array2**(const *View* &v)

template<class **View**>
array3::**array3**(const *View* &v)

template<class **View**>
array4::**array4**(const *View* &v)

Construct array from a view via a deep copy. The view, *v*, must support *multidimensional indexing*. The rate for the constructed array is initialized to the rate of the array associated with the view. Note that the actual rate may differ if the constructed array is a lower-dimensional slice of a higher-dimensional array due to lower rate granularity (see FAQ *#12*). The cache size of the constructed array is set to the default size.

12.8.6 Private immutable view

zfp's compressed arrays are in general not thread-safe. The main reason for this is that each array maintains its own cache of uncompressed blocks. Race conditions on the cache would occur unless it were locked upon each and every array access, which would have a prohibitive performance cost.

To ensure thread-safe access, zfp provides private mutable and immutable views of arrays that maintain their own private caches. The `private_const_view` immutable view provides read-only access to the underlying array. It is similar to a *const_view* in this sense, but differs in that it maintains its own private cache rather than sharing the cache owned by the array. Multiple threads may thus access the same array in parallel through their own private views.

Note: Thread safety is ensured only for OpenMP threads, and the zfp views must be compiled by an OpenMP compliant compiler. As the zfp compressed-array class implementation is defined in headers, the application code using zfp must also be compiled with OpenMP enabled if multithreaded access to zfp arrays is desired.

Note: Private views **do not guarantee cache coherence**. If, for example, the array is modified, then already cached data in a private view is not automatically updated. It is up to the user to ensure cache coherence by flushing (compressing modified blocks) or clearing (emptying) caches when appropriate.

The cache associated with a private view can be manipulated in the same way an array's cache can. For instance, the user may set the cache size on a per-view basis.

Unlike with *private mutable views*, private immutable views may freely access any element in the array visible through the view, i.e., multiple threads may read the same array element simultaneously. For an example of how to use private views for both read and write multithreaded access, see the *diffusion* code example.

Private views support only multidimensional indexing, i.e., they are neither flat nor nested.

class *array1*::**private_const_view**

class *array2*::**private_const_view**

class *array3*::**private_const_view**

class *array4*::**private_const_view**

Immutable views of 1D, 2D, 3D, and 4D arrays with private caches.

---

array1::*private_const_view*::**private_const_view**(*array1* *array)

array2::*private_const_view*::**private_const_view**(*array2* *array)

array3::*private_const_view*::**private_const_view**(*array3* *array)

array4::*private_const_view*::**private_const_view**(*array4* *array)

array1::*private_const_view*::**private_const_view**(*array1* *array, size_t x, size_t nx)

array2::*private_const_view*::**private_const_view**(*array2* *array, size_t x, size_t y, size_t nx, size_t ny)

array3::*private_const_view*::**private_const_view**(*array3* *array, size_t x, size_t y, size_t z, size_t nx, size_t ny, size_t nz) *array4*::*private_const_view*::**private_const_view**(*array4* *array, size_t x, size_t y, size_t z, size_t w, size_t nx, size_t ny, size_t nz, size_t nw)

Whole-array and sub-array private immutable view constructors. See *const_view constructors* for details.

size_t *array1*::*private_const_view*::**size_x**() const

size_t *array2*::*private_const_view*::**size_x**() const

size_t *array2*::*private_const_view*::**size_y**() const

size_t *array3*::*private_const_view*::**size_x**() const

size_t *array3*::*private_const_view*::**size_y**() const

size_t *array3*::*private_const_view*::**size_z**() const

size_t *array4*::*private_const_view*::**size_x**() const

size_t *array4*::*private_const_view*::**size_y**() const

size_t *array4*::*private_const_view*::**size_z**() const

size_t *array4*::*private_const_view*::**size_w**() const

View dimensions.

const_reference array1::*private_const_view*::**operator()**(size_t i) const

const_reference array2::*private_const_view*::**operator()**(size_t i, size_t j) const

const_reference array3::*private_const_view*::**operator()**(size_t i, size_t j, size_t k) const

const_reference array4::*private_const_view*::**operator()**(size_t i, size_t j, size_t k, size_t l) const

Return const reference to scalar element of a 1D, 2D, 3D, or 4D array.

The following functions are common among all dimensionalities:

size_t *arrayANY*::private_const_view::**cache_size**() const

void *arrayANY*::private_const_view::**set_cache_size**(size_t csize)

void *arrayANY*::private_const_view::**clear_cache**() const

Cache manipulation. See *Caching* for details.

12.8.7 Private mutable view

The mutable `private_view` supports both read and write access and is backed by a private cache. Because block compression, as needed to support write access, is not an atomic operation, mutable views and multithreading imply potential race conditions on the compressed blocks stored by an array. Although locking the array or individual blocks upon compression would be a potential solution, this would either serialize compression, thus hurting performance, or add a possibly large memory overhead by maintaining a lock with each block.

Note: To avoid multiple threads simultaneously compressing the same block, **private mutable views of an array must reference disjoint, block-aligned subarrays** for thread-safe access. Each block of 4^d array elements must be associated with at most one private mutable view, and therefore these views must reference non-overlapping rectangular subsets that are aligned on block boundaries, except possibly for partial blocks on the array boundary. (Expert users may alternatively ensure serialization of block compression calls and cache coherence in other ways, in which case overlapping private views may be permitted.)

Aside from this requirement, the user may partition the array into disjoint views in whatever manner is suitable for the application. The `private_view` API supplies a very basic partitioner to facilitate this task, but may not result in optimal partitions or good load balance.

When multithreaded write access is desired, any direct accesses to the array itself (i.e., not through a view) could invoke compression. Even a read access may trigger compression if a modified block is evicted from the cache. Hence, such direct array accesses should be confined to serial code sections when private views are used.

As with private immutable views, **cache coherence is not enforced**. Although this is less of an issue for private mutable views due to the requirement that views may not overlap, each private mutable view overlaps an index space with the underlying array whose cache is not automatically synchronized with the view's private cache. See the *diffusion* for an example of how to enforce cache coherence with mutable and immutable private views.

The `private_view` class inherits all public functions from `private_const_view`.

class *array1*::**private_view** : public *array1*::*private_const_view*

class *array2*::**private_view** : public *array2*::*private_const_view*

class *array3*::**private_view** : public *array3*::*private_const_view*

class *array4*::**private_view** : public *array4*::*private_const_view*

Mutable views of 1D, 2D, 3D, and 4D arrays with private caches.

---

class *array1*::*private_view*::**view_reference**

class *array2*::*private_view*::**view_reference**

class *array3*::*private_view*::**view_reference**

class *array4*::*private_view*::**view_reference**

Proxy references to array elements specialized for mutable private views.

---

array1::*private_view*::**private_view**(*array1* *array)

array2::*private_view*::**private_view**(*array2* *array)

array3::*private_view*::**private_view**(*array3* *array)

array4::*private_view*::**private_view**(*array4* *array)

array1::*private_view*::**private_view**(*array1* *array, size_t x, size_t nx)

array2::*private_view*::**private_view**(*array2* *array, size_t x, size_t y, size_t nx, size_t ny)

array3::*private_view*::**private_view**(*array3* *array, size_t x, size_t y, size_t z, size_t nx, size_t ny, size_t nz)

array4::*private_view*::**private_view**(*array4* *array, size_t x, size_t y, size_t z, size_t w, size_t nx, size_t ny, size_t nz, size_t nw)

Whole-array and sub-array private mutable view constructors. See *const_view constructors* for details.

---

array1::*private_view*::*view_reference* `array1::private_view::`**`operator()`**(size_t i) const

array2::*private_view*::*view_reference* `array2::private_view::`**`operator()`**(size_t i, size_t j) const

array3::*private_view*::*view_reference* `array3::private_view::`**`operator()`**(size_t i, size_t j, size_t k) const

array4::*private_view*::*view_reference* `array4::private_view::`**`operator()`**(size_t i, size_t j, size_t k, size_t l) const

Return reference to a scalar element of a 1D, 2D, 3D, or 4D array.

The following functions are common among all dimensionalities:

void `arrayANY::private_view::`**`partition`**(size_t index, size_t count)

Partition the current view into *count* roughly equal-size pieces along the view's longest dimension and modify the view's extents to match the piece indexed by *index*, with $0 \leq index < count$. These functions may be called multiple times, e.g., to recursively partition along different dimensions. The partitioner does not generate new views; it merely modifies the current values of the view's offsets and dimensions. Note that this may result in empty views whose dimensions are zero, e.g., if there are more pieces than blocks along a dimension.

---

void `arrayANY::private_view::`**`flush_cache`**() const

Flush cache by compressing any modified blocks and emptying the cache.

12.9 Codec

zfp arrays are partitioned into independent blocks that are compressed and decompressed using a *codec* (encoder/decoder). This codec defaults to the zfp compression scheme, but can in principle be any compression scheme or number representation that represents d-dimensional blocks of 4^d values. The `zfp::array` and `zfp::const_array` classes take such a codec class as an optional template parameter.

This section documents the API that prospective codecs must support to interface with the zfp compressed-array classes. Any one codec supports a specific scalar type (e.g., `float` or `double`), denoted `Scalar` below, and data dimensionality (1D, 2D, 3D, or 4D). If the codec does not support a certain compression mode, it should throw an *exception* when the user attempts to invoke that mode. Codecs reside in the `zfp::codec` namespace, e.g., `zfp::codec::zfp3<Scalar>` is the default codec for 3D arrays.

As of zfp 1.0.0, there is in addition to the default zfp codec a "generic" codec that allows storing data in zfp arrays in "uncompressed" form using any scalar type (specified as a template parameter). This "internal" scalar type may differ from the "external" scalar type exposed to the user through the `zfp::array` API. For instance, the internal type may be `float` while the external type is `double`, which provides for 2:1 fixed-rate "compression" using IEEE 754 floating point.

class **`codec`**

Fictitious class encapsulating the codec API. This may be thought of as a base class for the classes below specialized on dimensionality.

---

class **`codec1`**

class **`codec2`**

class **`codec3`**

class **codec4**

Fictitious classes encapsulating the codec API specialized for a given data dimensionality (1D, 2D, 3D, or 4D).

---

codec &`codec`::**operator=**(const *codec* &c)

Assignment operator. Performs a deep copy. This method is invoked when performing a *deep copy* of an array.

---

size_t `codec`::**buffer_size**(const zfp_field *field) const

Maximum buffer size needed to encode the *field* of given scalar type and dimensions (see `zfp_stream_maximum_size()`). The size should be based on the current compression mode and parameters. This method is called to determine how large a buffer to allocate and pass to `codec::open()`.

---

void `codec`::**open**(void *data, size_t size)

Open codec for (de)compression to/from buffer pointed to by *data* of *size* bytes. The caller is responsible for allocating and deallocating this buffer, whose *size* is given by `codec::buffer_size()`.

---

void `codec`::**close**()

Close codec for (de)compression.

---

zfp_mode `codec`::**mode**() const

Currently selected *compression mode*. See `zfp_mode`.

---

double `codec`::**rate**() const

Rate in compressed bits/value when *fixed-rate mode* is selected. See `zfp_stream_rate()`.

---

uint `codec`::**precision**() const

Precision in uncompressed bits/value when *fixed-precision mode* is selected. See `zfp_stream_precision()`.

---

double `codec`::**accuracy**() const

Accuracy as absolute error tolerance when *fixed-accuracy mode* is selected. See `zfp_stream_accuracy()`.

---

void `codec`::**params**(uint *minbits, uint *maxbits, uint *maxprec, int *minexp) const

Compression parameters for any compression mode. These pointer parameters may be `null` if only a subset of parameters is requested. See `zfp_stream_params()`.

---

void `codec`::**set_reversible**()

Enable *reversible mode*.

---

double *codec*::**set_rate**(double rate, bool align)

Set desired *rate* in number of compressed bits/value. When *align* = true, blocks are word aligned, as needed for random access writes. Return the closest rate supported. See *zfp_stream_set_rate()*.

---

uint *codec*::**set_precision**(uint precision)

Set precision in number of uncompressed bits/value. Return the actual precision selected. See *zfp_stream_set_precision()*.

---

double *codec*::**set_accuracy**(double tolerance)

Set accuracy as absolute error tolerance. Return the closest tolerance supported. See *zfp_stream_set_accuracy()*.

---

bool *codec*::**set_params**(uint minbits, uint maxbits, uint maxprec, int minexp)

Set expert mode parameters. Return `true` on success. See *zfp_stream_set_params()*.

---

bool *codec*::**set_thread_safety**(bool safety)

Enable or disable thread safety. This function is called whenever zfp is built with OpenMP support and when the number of mutable or immutable *private views* of an array changes. When two or more private views of an array are accessed by separate threads, multiple blocks may be compressed or decompressed simultaneously. The codec then has to take care that there are no race conditions on the data structures (e.g., *bitstream*) used for (de)compression.

---

size_t *codec*::**size_bytes**(uint mask = ZFP_DATA_ALL) const

Return storage size of components of codec data structure indicated by *mask*. The mask is constructed via bitwise OR of *predefined constants*.

---

static size_t *codec*::**alignment**()

Memory alignment in number of bytes required by codec.

static const zfp_type *codec*::**type**;

Scalar type compressed by codec.

---

size_t *codec*::**encode_block**(bitstream_offset offset, const Scalar *block) const

Encode contiguous *block* of 4^d scalars and store at specified bit *offset* within compressed-data buffer. Return the number of bits of compressed storage for the block, excluding any necessary padding. This method must flush any buffered compressed data without counting any padding (e.g., for byte alignment) in the compressed size (unless the codec requires alignment of the bit offsets).

---

size_t *codec*::**decode_block**(bitstream_offset offset, Scalar *block) const

Decode contiguous *block* of 4^d scalars from specified bit *offset* within compressed-data buffer (see *codec::encode_block()*). Return number of bits of compressed data decoded, excluding any padding bits, i.e., the same value reported in encoding.

---

size_t *codec1*::**encode_block**(bitstream_offset offset, uint shape, const Scalar *block) const

size_t *codec2*::**encode_block**(bitstream_offset offset, uint shape, const Scalar *block) const

size_t *codec3*::**encode_block**(bitstream_offset offset, uint shape, const Scalar *block) const

size_t *codec4*::**encode_block**(bitstream_offset offset, uint shape, const Scalar *block) const

Encode contiguous *block* of data of given *shape* and store at specified bit *offset* within compressed-data buffer. Return the number of bits of compressed storage for the block (see also `codec::encode_block()`).

The *shape* is a $(2 \times d)$-bit encoding of the size of the d-dimensional block. For each successive pair of bits s of *shape*, the block size in the corresponding dimension is $n = 4 - s$, where $0 \le s \le 3$. Thus, *shape* = 0 implies a full block of 4^d values. The size of the fastest varying dimension is specified in the least significant bits of *shape*.

---

size_t *codec1*::**decode_block**(bitstream_offset offset, uint shape, Scalar *block) const

size_t *codec2*::**decode_block**(bitstream_offset offset, uint shape, Scalar *block) const

size_t *codec3*::**decode_block**(bitstream_offset offset, uint shape, Scalar *block) const

size_t *codec4*::**decode_block**(bitstream_offset offset, uint shape, Scalar *block) const

Decode contiguous *block* of data of given *shape* from specified bit *offset* within compressed-data buffer (see also *codec1::encode_block()*). Return number of bits of compressed data decoded, excluding any padding bits, i.e., the same value reported in encoding.

---

size_t *codec1*::**encode_block_strided**(bitstream_offset offset, uint shape, const Scalar *p, ptrdiff_t sx) const

size_t *codec2*::**encode_block_strided**(bitstream_offset offset, uint shape, const Scalar *p, ptrdiff_t sx, ptrdiff_t sy) const

size_t *codec3*::**encode_block_strided**(bitstream_offset offset, uint shape, const Scalar *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz) const

size_t *codec4*::**encode_block_strided**(bitstream_offset offset, uint shape, const Scalar *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw) const

Encode block of data stored at *p* with strides *sx*, *sy*, *sz*, and *sw*. See *zfp_field* for information on strided storage. The *shape*, *offset*, and return value are as in *codec1::encode_block()*.

---

size_t *codec1*::**decode_block_strided**(bitstream_offset offset, uint shape, Scalar *p, ptrdiff_t sx) const

size_t *codec2*::**decode_block_strided**(bitstream_offset offset, uint shape, Scalar *p, ptrdiff_t sx, ptrdiff_t sy) const

size_t *codec3*::**decode_block_strided**(bitstream_offset offset, uint shape, Scalar *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz) const size_t *codec4*::**decode_block_strided**(bitstream_offset offset, uint shape, Scalar *p, ptrdiff_t sx, ptrdiff_t sy, ptrdiff_t sz, ptrdiff_t sw) const

Decode block to strided storage pointed to by *p* with strides *sx*, *sy*, *sz*, and *sw*. See *zfp_field* for information on strided storage. The *shape*, *offset*, and return value are as in *codec1::decode_block()*.

12.10 Index

To support random access, zfp arrays must know where each block is stored in memory. For fixed-rate arrays, the number of compressed bits per block is constant, and the bit offset to each block can be quickly computed. For variable-rate arrays, the compressed block size is data dependent, and additional information must be stored to index the blocks. Toward this end, zfp arrays make use of an index class that reports the offset and size (in number of bits) of each block. The *zfp::array* and *zfp::const_array* classes take such an index class as a template parameter. This index class is new as of zfp 1.0.0, which introduced variable-rate arrays.

Because zfp is designed primarily for very large arrays, the bit offset may exceed 32 bits. A straightforward implementation stores the bit offset to each block as a 64-bit integer, with the block size given by the difference of consecutive offsets. However, this overhead of 64 bits/block may exceed the payload compressed data for low-dimensional arrays or in applications like visualization that may store less than one bit per value (amortized). It is therefore important to consider more compact representations of the block index.

zfp provides multiple index classes in the `zfp::index` namespace that balance storage size, range of representable block offsets and sizes, and speed of access:

- `implicit`: Used for fixed-rate storage where only the fixed number of bits per block is kept. This is the default index for fixed-rate arrays.
- `verbatim`: This and subsequent classes support variable-rate storage. A full 64-bit offset is stored per block.
- `hybrid4`: Four consecutive offsets are encoded together. The top 32 bits of a 44-bit base offset are stored, with the 12 least significant bits of this base set to zero. Four unsigned 16-bit deltas from the base offset complete the representation. The default for variable-rate arrays, this index offers a good tradeoff between storage, offset range, and speed.
- `hybrid8`: Eight consecutive offsets are encoded together as two 64-bit words that store the offset to the first block (the base offset) and the sizes of the first seven blocks, from which the eight offsets are derived as a prefix sum. One 64-bit word holds the 8 least significant bits of the base offset and block sizes. The other word holds another 2 (d - 1) bits for the seven block sizes plus the top 78 - 14 d bits of the base offset, where $1 \leq d \leq 4$ is the data dimensionality.

Properties of these index classes are summarized in Table 12.1.

Table 12.1: Properties of index classes. Storage is measured in amortized bits/block; offset and size denote supported ranges in number of bits.

index class	variable rate	storage	offset	size	speed
implicit		0	64	64	high
verbatim	✓	64	64	64	high
hybrid4	✓	24	44	16	medium
hybrid8	✓	16	86 - 14 d	6 + 2 d	low

This section documents the API that prospective block indices must support to interface with the zfp compressed-array classes.

class **index**

Fictitious class encapsulating the index API.

`index::index(size_t blocks)`

Construct index supporting the given number of *blocks*.

`size_t index::size_bytes(uint mask = ZFP_DATA_ALL) const`

`bitstream_size index::range() const`

Range of bit offsets spanned by index. This equals the total number of bits of compressed-array data.

`size_t index::block_size(size_t block_index) const`

Size of compressed block in number of bits.

`bitstream_offset index::block_offset(size_t block_index) const`

Bit offset to compressed block data.

`void resize(size_t blocks)`

Resize index to accommodate requested number of blocks. Any stored index data is destroyed.

`void clear()`

Clear all data stored by index.

`void flush()`

Flush any buffered index data. This method is called after all blocks have been compressed, e.g., in `array::set()`.

`void set_block_size(size_t size)`

Set a fixed compressed block size in number of bits for all blocks. This method is called when fixed-rate mode is selected.

`void set_block_size(size_t block_index, size_t size)`

Set compressed block size in number of bits for a single block. For variable-rate arrays, the zero-based *block_index* is guaranteed to increase sequentially between calls. This method throws an exception if the index cannot support the block size or offset. The user may wish to restrict the block size, e.g., by setting `maxbits` in *expert mode*, to guard against such overflow.

`static bool has_variable_rate()`

Return true if index supports variable-sized blocks.

CHAPTER

THIRTEEN

COMPRESSED-ARRAY C BINDINGS

zfp 0.5.4 adds cfp: C language bindings for compressed arrays via wrappers around the *C++ classes*. zfp 1.0.0 modifies its API (see below).

The C API has been designed to facilitate working with compressed arrays without the benefits of C++ operator overloading and self-aware objects, which greatly simplify the syntax. Whereas one possible design considered is to map each C++ method to a C function with a prefix, such as `zfp_array3d_get(a, i, j, k)` in place of `a(i, j, k)` for accessing an element of a 3D array of doubles, such code would quickly become unwieldy when part of longer expressions.

Instead, cfp uses the notion of nested C *namespaces* that are structs of function pointers, such as `cfp.array3d`. Although this may seem no more concise than a design based on prefixes, the user may alias these namespaces (somewhat similar to C++ `using namespace` declarations) using far shorter names via C macros or local variables. For instance:

```
const cfp_array3d_api _ = cfp.array3d; // _ is a namespace alias
cfp_array3d a = _.ctor(nx, ny, nz, rate, 0, 0);
double value = _.get(a, i, j, k);
_.set(a, i, j, k, value + 1);
```

which is a substitute for the C++ code

```
zfp::array3d a(nx, ny, nz, rate, 0, 0);
double value = a(i, j, k);
a(i, j, k) = value + 1;
```

Because the underlying C++ array objects have no corresponding C representation, and because C objects are not self aware (they have no implicit `this` pointer), the C interface interacts with compressed arrays through array object *pointers*, wrapped in structs, that cfp converts to pointers to the corresponding C++ objects. As a consequence, cfp compressed arrays must be allocated on the heap and must be explicitly freed via designated destructor functions to avoid memory leaks (this is not necessary for references, pointers, and iterators, which have their own C representation). The C++ constructors are mapped to C by allocating objects via C++ `new`. Moreover, the C API requires passing an array *self pointer* (wrapped within a cfp array struct) in order to manipulate the array.

As with the *C++ classes*, array elements can be accessed via multidimensional array indexing, e.g., `get(array, i, j)`, and via flat, linear indexing, e.g., `get_flat(array, i + nx * j)`.

Note: The cfp API changed in zfp 1.0.0 by wrapping array *self pointers* in structs to align the interface more closely with the C++ API and to avoid confusion when discussing arrays (now `cfp.array` rather than `cfp.array*`) and pointers to arrays (now `cfp.array*` rather than `cfp.array**`). Furthermore, zfp 1.0.0 adds support for proxy references, proxy pointers, and iterators that also wrap C++ classes. Manipulating those indirectly via pointers (like the old cfp arrays) would require additional user effort to destroy dynamically allocated lightweight objects and would also reduce code readability, e.g., `cfp_ptr1d*` (whose corresponding C++ type is `zfp::array1d::pointer*`) reads more naturally as a raw pointer to a proxy pointer than an indirectly referenced proxy pointer object that the user must remember to implicitly dereference.

The following sections are available:

- *Arrays*
- *Serialization*
- *References*
- *Pointers*
- *Iterators*

13.1 Arrays

cfp implements eight array types for 1D, 2D, 3D, and 4D arrays of floats and doubles. These array types share many functions that have the same signature. To reduce redundancy in the documentation, we define fictitious types *`cfp_arrayf`* and *`cfp_arrayd`* for N-dimensional ($1 \leq N \leq 4$) arrays of floats or doubles, *`cfp_array1`*, *`cfp_array2`*, *`cfp_array3`*, and *`cfp_array4`* for 1D, 2D, 3D, and 4D arrays of either floats or doubles, and *`cfp_array`* for arrays of any dimensionality and type. We also make use of corresponding namespaces, e.g., `cfp.array1` refers to the API common to one-dimensional arrays of floats or doubles. These types and namespaces are not actually part of the cfp API.

Note: The cfp array API makes use of `const` qualifiers for `struct` parameters (passed by value) merely to indicate when the corresponding object is not modified, e.g., `const cfp_array1f self`. This construction serves to document functions that are analogous to `const` qualified C++ member functions.

Note: Support for 4D arrays was added to cfp in version 1.0.0.

type **`cfp_array1f`**

type **`cfp_array1d`**

type **`cfp_array2f`**

type **`cfp_array2d`**

type **`cfp_array3f`**

type **`cfp_array3d`**

type **`cfp_array4f`**

type **`cfp_array4d`**

Opaque types for 1D, 2D, 3D, and 4D compressed arrays of floats and doubles.

type **`cfp_array1`**

type **`cfp_array2`**

type **`cfp_array3`**

type **cfp_array4**

Fictitious types denoting 1D, 2D, 3D, and 4D arrays of any scalar type.

---

type **cfp_arrayf**

type **cfp_arrayd**

Fictitious types denoting any-dimensional arrays of floats and doubles.

---

type **cfp_array**

Fictitious type denoting array of any dimensionality and scalar type.

---

struct **cfp**

struct **array1f**

struct **array1d**

struct **array2f**

struct **array2d**

struct **array3f**

struct **array3d**

struct **array4f**

struct **array4d**

struct **header**

Nested C "namespaces" for encapsulating the cfp API. The outer *cfp* namespace may be redefined at compile-time via the macro *CFP_NAMESPACE*, e.g., to avoid symbol clashes. The inner namespaces hold function pointers to the cfp wrappers documented below.

---

cfp_array1f cfp.array1f.**ctor**(size_t nx, double rate, const float *p, size_t cache_size)

cfp_array1d cfp.array1d.**ctor**(size_t nx, double rate, const double *p, size_t cache_size)

cfp_array2f cfp.array2f.**ctor**(size_t nx, size_t ny, double rate, const float *p, size_t cache_size)

cfp_array2d cfp.array2d.**ctor**(size_t nx, size_t ny, double rate, const double *p, size_t cache_size)

cfp_array3f cfp.array3f.**ctor**(size_t nx, size_t ny, size_t nz, double rate, const float *p, size_t cache_size)

cfp_array3d cfp.array3d.**ctor**(size_t nx, size_t ny, size_t nz, double rate, const double *p, size_t cache_size)

cfp_array4f cfp.array4f.**ctor**(size_t nx, size_t ny, size_t nz, size_t nw, double rate, const float *p, size_t cache_size) *cfp_array4d* `cfp.array4d.ctor`(size_t nx, size_t ny, size_t nz, size_t nw, double rate, const double *p, size_t cache_size)

Array constructors. If *p* is not `NULL`, then the array is initialized from uncompressed storage; otherwise the array is zero initialized. *cache_size* is the minimum size cache (in bytes) to use. If *cache_size* is zero, a default size is chosen.

---

cfp_array `cfp.array.ctor_default`()

Default constructor. Allocate an empty array that later can be *resized* and whose rate and cache size can be set by `cfp.array.set_rate()` and `cfp.array.set_cache_size()`.

---

cfp_array `cfp.array.ctor_copy`(const *cfp_array* src)

Copy constructor.

---

cfp_array `cfp.array.ctor_header`(const *cfp_header* h, const void *buffer, size_t buffer_size_bytes);

Constructor from metadata given by the *header* *h* and optionally initialized with compressed data from *buffer* of size *buffer_size_bytes*. See *corresponding C++ constructor*.

---

void `cfp.array.dtor`(*cfp_array* self)

Destructor. The destructor not only deallocates any compressed data owned by the array, but also frees memory for itself, invalidating the *self* object upon return. Note that the user must explicitly call the destructor to avoid memory leaks.

---

void `cfp.array.deep_copy`(*cfp_array* self, const *cfp_array* src)

Perform a deep copy of *src* analogous to the *C++ assignment operator*.

---

float `cfp.array1f.get`(const *cfp_array1f* self, size_t i)

float `cfp.array2f.get`(const *cfp_array2f* self, size_t i, size_t j)

float `cfp.array3f.get`(const *cfp_array3f* self, size_t i, size_t j, size_t k)

float `cfp.array4f.get`(const *cfp_array4f* self, size_t i, size_t j, size_t k, size_t l)

double `cfp.array1d.get`(const *cfp_array1d* self, size_t i)

double `cfp.array2d.get`(const *cfp_array2d* self, size_t i, size_t j)

double `cfp.array3d.get`(const *cfp_array3d* self, size_t i, size_t j, size_t k)

double `cfp.array4d.get`(const *cfp_array4d* self, size_t i, size_t j, size_t k, size_t l)

Array inspectors via multidimensional indexing.

---

void `cfp.array1f.set`(const *cfp_array1f* self, size_t i, float val) void `cfp.array2f.set`(const *cfp_array2f* self, size_t i, size_t j, float val)

void `cfp.array3f.set`(const *cfp_array3f* self, size_t i, size_t j, size_t k, float val)

void `cfp.array4f.set`(const *cfp_array4f* self, size_t i, size_t j, size_t k, size_t l, float val)

void `cfp.array1d.set`(const *cfp_array1d* self, size_t i, double val)

void `cfp.array2d.set`(const *cfp_array2d* self, size_t i, size_t j, double val)

void `cfp.array3d.set`(const *cfp_array3d* self, size_t i, size_t j, size_t k, double val)

void `cfp.array4d.set`(const *cfp_array4d* self, size_t i, size_t j, size_t k, size_t l, double val)

Array mutators for assigning values to array elements via multidimensional indexing.

---

float `cfp.arrayf.get_flat`(const *cfp_arrayf* self, size_t index)

double `cfp.arrayd.get_flat`(const *cfp_arrayd* self, size_t index)

Flat index array inspectors; see `array::operator[]()`.

---

void `cfp.arrayf.set_flat`(*cfp_arrayf* self, size_t index, float val)

void `cfp.arrayd.set_flat`(*cfp_arrayd* self, size_t index, double val)

Flat index array mutators; set array element with flat *index* to *val*.

---

void `cfp.arrayf.get_array`(const *cfp_arrayf* self, float *p)

void `cfp.arrayd.get_array`(const *cfp_arrayd* self, double *p)

Decompress entire array; see `array::get()`.

---

void `cfp.arrayf.set_array`(*cfp_arrayf* self, const float *p)

void `cfp.arrayd.set_array`(*cfp_arrayd* self, const double *p)

Initialize entire array; see `array::set()`.

---

size_t `cfp.array2.size_x`(const *cfp_array2* self)

size_t `cfp.array2.size_y`(const *cfp_array2* self)

size_t `cfp.array3.size_x`(const *cfp_array3* self)

size_t `cfp.array3.size_y`(const *cfp_array3* self)

size_t `cfp.array3.size_z`(const *cfp_array3* self)

size_t `cfp.array4.size_x`(const *cfp_array4* self)

size_t `cfp.array4.size_y`(const *cfp_array4* self)

size_t `cfp.array4.size_z`(const *cfp_array4* self)

size_t *cfp*.array4.**size_w**(const *cfp_array4* self)
  Array dimensions.

---

size_t *cfp*.array.**size**(const *cfp_array* self)
  See *array::size()*.

---

void *cfp*.array1.**resize**(*cfp_array1* self, size_t n, *zfp_bool* clear)

void *cfp*.array2.**resize**(*cfp_array2* self, size_t nx, size_t ny, *zfp_bool* clear)

void *cfp*.array3.**resize**(*cfp_array3* self, size_t nx, size_t ny, size_t nz, *zfp_bool* clear)

void *cfp*.array4.**resize**(*cfp_array4* self, size_t nx, size_t ny, size_t nz, size_t nw, *zfp_bool* clear)
  Resize array.

---

double *cfp*.array.**rate**(const *cfp_array* self)
  See *array::rate()*.

---

double *cfp*.array.**set_rate**(*cfp_array* self, double rate)
  See *array::set_rate()*.

---

size_t *cfp*.array.**cache_size**(const *cfp_array* self)
  See *array::cache_size()*.

---

void *cfp*.array.**set_cache_size**(*cfp_array* self, size_t cache_size)
  See *array::set_cache_size()*.

---

void *cfp*.array.**clear_cache**(const *cfp_array* self)
  See *array::clear_cache()*.

---

void *cfp*.array.**flush_cache**(const *cfp_array* self)
  See *array::flush_cache()*.

---

size_t *cfp*.array.**size_bytes**(const *cfp_array* self, uint mask)
  See *array::size_bytes()*.

---

size_t *cfp*.array.**compressed_size**(const *cfp_array* self)
  See *array::compressed_size()*.

---

void *cfp.array.**compressed_data**(const *cfp_array* self)

See `array::compressed_data()`.

---

cfp_ref1 cfp.array1.**ref**(*cfp_array1* self, size_t i)

cfp_ref2 cfp.array2.**ref**(*cfp_array2* self, size_t i, size_t j)

cfp_ref3 cfp.array3.**ref**(*cfp_array3* self, size_t i, size_t j, size_t k)

cfp_ref4 cfp.array4.**ref**(*cfp_array4* self, size_t i, size_t j, size_t k, size_t l)

Reference *constructor* via multidimensional indexing.

---

cfp_ref cfp.array.**ref_flat**(*cfp_array* self, size_t i)

Reference *constructor* via flat indexing.

---

cfp_ptr1 cfp.array1.**ptr**(*cfp_array1* self, size_t i)

cfp_ptr2 cfp.array2.**ptr**(*cfp_array2* self, size_t i, size_t j)

cfp_ptr3 cfp.array3.**ptr**(*cfp_array3* self, size_t i, size_t j, size_t k)

cfp_ptr4 cfp.array4.**ptr**(*cfp_array4* self, size_t i, size_t j, size_t k, size_t l)

Obtain pointer to array element via multidimensional indexing.

---

cfp_ptr cfp.array.**ptr_flat**(*cfp_array* self, size_t i)

Obtain pointer to array element via flat indexing.

---

cfp_iter cfp.array.**begin**(*cfp_array* self)

Return iterator to beginning of array; see `array::begin()`.

---

cfp_iter cfp.array.**end**(*cfp_array* self)

Return iterator to end of array; see `array::end()`.

13.2 Serialization

zfp 1.0.0 adds cfp array *serialization*. Like zfp's C++ arrays, cfp arrays can be serialized and deserialized to and from sequential storage. As with the C++ arrays, (de)serialization is done with the assistance of a header class, `cfp_header`. Currently, cfp provides no *factory function*—the caller must either know which type of array (dimensionality and scalar type) to *construct* at compile-time or obtain this information at run-time from a header *constructed* from a memory buffer.

13.2.1 Header

`cfp_header` is a wrapper around `array::header`. Although the header type is shared among all array types, the header API is accessed through the associated array type whose metadata the header describes. For example, `cfp.array3f.header.ctor(const cfp_array3f a)` constructs a header for a `cfp_array3f`. The header is dynamically allocated and must be explicitly destructed via `cfp.array.header.dtor()`.

type **cfp_header**

Wrapper around `array::header`.

---

cfp_header `cfp.array.header.`**`ctor`**(const *cfp_array* a);

Construct a header that describes the metadata of an existing array *a*.

---

cfp_header `cfp.array.header.`**`ctor_buffer`**(const void *data, size_t size)

Construct a header from header *data* buffer of given byte *size*.

---

void `cfp.array.header.`**`dtor`**(*cfp_header* self);

Destructor. Deallocates all data associated with the header. The user must call the destructor to avoid memory leaks.

---

zfp_type `cfp.array.header.`**`scalar_type`**(const *cfp_header* self);

Scalar type associated with array. See `header::scalar_type()`.

---

uint `cfp.array.header.`**`dimensionality`**(const *cfp_header* self);

Dimensionality associated with array. See `header::dimensionality()`.

---

size_t `cfp.array.header.`**`size_x`**(const *cfp_header* self);

size_t `cfp.array.header.`**`size_y`**(const *cfp_header* self);

size_t `cfp.array.header.`**`size_z`**(const *cfp_header* self);

size_t `cfp.array.header.`**`size_w`**(const *cfp_header* self);

Array dimensions. Unused dimensions have a size of zero.

---

double `cfp.array.header.`**`rate`**(const *cfp_header* self);

Rate in bits/value. See `header::rate()`.

---

const void *`cfp.array.header.`**`data`**(const *cfp_header* self);

Pointer to header data buffer needed for serializing the header. See `header::data()`.

---

size_t `cfp.array.header.`**`size_bytes`**(const *cfp_header* self, uint mask);

When *mask* = `ZFP_DATA_HEADER`, byte size of header data buffer needed for serializing the header. See `header::size_bytes()`.

13.3 Array Accessors

zfp 1.0.0 adds cfp support for proxy *references* and *pointers* to individual array elements, as well as *iterators* for traversing arrays. These are analogues to the corresponding C++ classes. As with *arrays*, fictitious types and namespaces are used to shorten the documentation.

Note: Unlike the case of arrays, for which the surrounding struct stores a pointer to the underlying array object to allow modifications of the array, the cfp proxy reference, proxy pointer, and iterator objects are all passed by value, and hence none of the functions below modify the *self* argument. To increment a pointer, for instance, one should call `p = cfp.array.pointer.inc(p)`. Note that while the references, pointers, and iterators are not themselves modified, the array elements that they reference can be modified.

13.4 References

cfp proxy references wrap the C++ *reference* classes. References are constructed via `cfp.array.ref()`, `cfp.array.pointer.ref()`, and `cfp.array.iterator.ref()` (as well as associated `ref_flat` and `ref_at` calls).

Note: cfp references exist primarily to provide parity with zfp references. As references do not exist in C, the preferred way of accessing arrays is via *proxy pointers*, *iterators*, or *index-based array accessors*.

cfp references do provide the same guarantees as C++ references, functioning as aliases to initialized members of the cfp wrapped zfp array. This is with the caveat that they are only accessed via cfp API calls (use of the = C assignment operator to shallow copy a `cfp_ref` is also allowed in this case).

---

type **`cfp_ref1f`**

type **`cfp_ref2f`**

type **`cfp_ref3f`**

type **`cfp_ref4f`**

type **`cfp_ref1d`**

type **`cfp_ref2d`**

type **`cfp_ref3d`**

type **`cfp_ref4d`**

Opaque types for proxy references to 1D, 2D, 3D, and 4D compressed float or double array elements.

---

type **`cfp_ref1`**

type **`cfp_ref2`**

type **`cfp_ref3`**

type **`cfp_ref4`**

Fictitious types denoting references into 1D, 2D, 3D, and 4D arrays of any scalar type.

---

type **`cfp_reff`**

type **`cfp_refd`**

Fictitious types denoting references into float or double arrays of any dimensionality.

---

type **`cfp_ref`**

Fictitious type denoting reference into array of any dimensionality and scalar type.

---

float *`cfp`*`.arrayf.reference.`**`get`**(const *cfp_reff* self)

double *`cfp`*`.arrayd.reference.`**`get`**(const *cfp_refd* self)

Retrieve value referenced by *self*.

---

void *`cfp`*`.arrayf.reference.`**`set`**(*cfp_reff* self, float val)

void *`cfp`*`.arrayd.reference.`**`set`**(*cfp_refd* self, double val)

Update value referenced by *self*; see *`reference::operator=()`*.

---

cfp_ptr *`cfp`*`.array.reference.`**`ptr`**(*cfp_ref* self)

Obtain proxy pointer to value referenced by *self*; see *`reference::operator&()`*.

---

void *`cfp`*`.array.reference.`**`copy`**(*cfp_ref* self, const *cfp_ref* src)

Copy value referenced by *src* to value referenced by *self*; see *`reference::operator=()`*. This performs a deep copy. This is in contrast to `self = src`, which performs only a shallow copy.

13.5 Pointers

cfp proxy pointers wrap the C++ *pointer* classes. Pointers are constructed via `cfp.array.ptr()` and *`cfp.array.reference.ptr()`* (and associated `ptr_flat` and `ptr_at` calls). All pointers are *passed by value* and are themselves not modified by these functions.

Note: As with `array::pointer`, *`cfp_ptr`* indexing is based on element-wise ordering and is unaware of zfp blocks. This may result in a suboptimal access pattern if sequentially accessing array members. To take advantage of zfp block traversal optimization, see *iterators*.

type **`cfp_ptr1f`**

type **`cfp_ptr2f`**

type **`cfp_ptr3f`**

type **`cfp_ptr4f`**

type **`cfp_ptr1d`**

type **cfp_ptr2d**

type **cfp_ptr3d**

type **cfp_ptr4d**

Opaque types for proxy pointers to 1D, 2D, 3D, and 4D compressed float or double array elements.

---

type **cfp_ptr1**

type **cfp_ptr2**

type **cfp_ptr3**

type **cfp_ptr4**

Fictitious types denoting pointers into 1D, 2D, 3D, and 4D arrays of any scalar type.

---

type **cfp_ptrf**

type **cfp_ptrd**

Fictitious types denoting pointers into float or double arrays of any dimensionality.

---

type **cfp_ptr**

Fictitious type denoting pointer into array of any dimensionality and scalar type.

---

float `cfp.arrayf.pointer.`**`get`**(const *cfp_ptrf* self)

double `cfp.arrayd.pointer.`**`get`**(const *cfp_ptrd* self)

Dereference operator; `*self`. See `pointer::operator*()`.

---

float `cfp.arrayf.pointer.`**`get_at`**(const *cfp_ptrf* self, ptrdiff_t d)

double `cfp.arrayd.pointer.`**`get_at`**(const *cfp_ptrd* self, ptrdiff_t d)

Offset dereference operator; `self[d]`. See `pointer::operator[]()`.

---

void `cfp.arrayf.pointer.`**`set`**(*cfp_ptrf* self, float val)

void `cfp.arrayd.pointer.`**`set`**(*cfp_ptrd* self, double val)

Dereference operator with assignment; `*self = val`. See `pointer::operator*()`.

---

void `cfp.arrayf.pointer.`**`set_at`**(*cfp_ptrf* self, ptrdiff_t d, float val)

void `cfp.arrayd.pointer.`**`set_at`**(*cfp_ptrd* self, ptrdiff_t d, double val)

Offset dereference operator with assignment; `self[d] = val`. See `pointer::operator[]()`.

---

cfp_ref `cfp.array.pointer.`**`ref`**`(`*cfp_ptr* self`)`

Get proxy reference to element stored at `*self`. See *`pointer::operator*()`*.

---

cfp_ref `cfp.array.pointer.`**`ref_at`**`(`*cfp_ptr* self, ptrdiff_t d`)`

Get proxy reference to element stored at `self[d]`. See *`pointer::operator[]()`*.

---

zfp_bool `cfp.array.pointer.`**`lt`**`(`const *cfp_ptr* lhs, const *cfp_ptr* rhs`)`

zfp_bool `cfp.array.pointer.`**`gt`**`(`const *cfp_ptr* lhs, const *cfp_ptr* rhs`)`

zfp_bool `cfp.array.pointer.`**`leq`**`(`const *cfp_ptr* lhs, const *cfp_ptr* rhs`)`

zfp_bool `cfp.array.pointer.`**`geq`**`(`const *cfp_ptr* lhs, const *cfp_ptr* rhs`)`

Return true if the two pointers satisfy the given *relationship*; `lhs < rhs`, `lhs > rhs`, `lhs <= rhs`, `lhs >= rhs`.

---

zfp_bool `cfp.array.pointer.`**`eq`**`(`const *cfp_ptr* lhs, const *cfp_ptr* rhs`)`

Compare two proxy pointers for equality; `lhs == rhs`. The pointers must be to elements with the same index within the same array to satisfy equality. See *`pointer::operator==()`*.

---

int `cfp.array.pointer.`**`neq`**`(`const *cfp_ptr* lhs, const *cfp_ptr* rhs`)`

Compare two proxy pointers for inequality; `lhs != rhs`. The pointers are not equal if they point to different arrays or to elements with different index within the same array. See *`pointer::operator!=()`*.

---

ptrdiff_t `cfp.array.pointer.`**`distance`**`(`const *cfp_ptr* first, const *cfp_ptr* last`)`

Return the difference between two proxy pointers in number of linear array elements; `last - first`. See *`pointer::operator-()`*.

---

cfp_ptr `cfp.array.pointer.`**`next`**`(`const *cfp_ptr* p, ptrdiff_t d`)`

Return the result of incrementing pointer by *d* elements; `p + d`. See *`pointer::operator+()`*.

---

cfp_ptr `cfp.array.pointer.`**`prev`**`(`const *cfp_ptr* p, ptrdiff_t d`)`

Return the result of decrementing pointer by *d* elements; `p - d`. See *`pointer::operator-()`*.

---

cfp_ptr `cfp.array.pointer.`**`inc`**`(`const *cfp_ptr* p`)`

Return the result of incrementing pointer by one element; `p + 1`. See *`pointer::operator++()`*.

---

cfp_ptr `cfp.array.pointer.`**`dec`**`(`const *cfp_ptr* p`)`

Return the result of decrementing pointer by one element; `p - 1`. See *`pointer::operator--()`*.

13.6 Iterators

cfp random-access iterators wrap the C++ *iterator* classes. All iterators are *passed by value* and are themselves not modified by these functions. Iterators are constructed similar to C++ iterators via `cfp.array.begin()` and `cfp.array.end()`. Iterator usage maps closely to equivalent C++ iterator syntax. For example, to set an array to all ones:

```
// _ and _iter are namespace aliases
const cfp_array3d_api _ = cfp.array3d;
const cfp_iter3d_api _iter = _.iterator;

cfp_array3d a = _.ctor(nx, ny, nz, rate, 0, 0);
cfp_iter3d it;

for (it = _.begin(a); _iter.neq(it, _.end(a)); it = _iter.inc(it))
  _iter.set(it, 1.0);
```

type **cfp_iter1f**

type **cfp_iter2f**

type **cfp_iter3f**

type **cfp_iter4f**

type **cfp_iter1d**

type **cfp_iter2d**

type **cfp_iter3d**

type **cfp_iter4d**

Opaque types for block iterators over 1D, 2D, 3D, and 4D compressed float or double array elements.

---

type **cfp_iter1**

type **cfp_iter2**

type **cfp_iter3**

type **cfp_iter4**

Fictitious types denoting iterators over 1D, 2D, 3D, and 4D arrays of any scalar type.

---

type **cfp_iterf**

type **cfp_iterd**

Fictitious types denoting iterators over float or double arrays of any dimensionality.

---

type **cfp_iter**

Fictitious type denoting iterator over array of any dimensionality and scalar type.

---

float *cfp*.arrayf.iterator.**get**(const *cfp_iterf* self)

double *cfp*.arrayd.iterator.**get**(const *cfp_iterd* self)

Return element referenced by iterator; `*self`. See *iterator::operator*()*.

---

float *cfp*.*array1f*.iterator.**get_at**(const *cfp_iter1f* self, ptrdiff_t d)

double *cfp*.*array1d*.iterator.**get_at**(const *cfp_iter1d* self, ptrdiff_t d)

Return element *d* elements (may be negative) from iterator; `self[d]`. See *iterator::operator[]()*.

---

void *cfp*.arrayf.iterator.**set**(*cfp_iterf* self, float val)

void *cfp*.arrayd.iterator.**set**(*cfp_iterd* self, double val)

Update element referenced by iterator; `*self = val`. See *iterator::operator*()*.

---

void *cfp*.*array1f*.iterator.**set_at**(*cfp_iter1* self, ptrdiff_t d, float val)

void *cfp*.*array1d*.iterator.**set_at**(*cfp_iter1* self, ptrdiff_t d, double val)

Update element *d* elements (may be negative) from iterator; `self[d] = val`. See *iterator::operator[]()*.

---

cfp_ref *cfp*.array.iterator.**ref**(*cfp_iter* self)

Return reference to element referenced by iterator; `*self`. See *iterator::operator*()*.

---

cfp_ref *cfp*.array.iterator.**ref_at**(*cfp_iter* self, ptrdiff_t d)

Return reference to an element offset *d* elements (may be negative) from iterator; `self[d]`. See *iterator::operator[]()*.

---

cfp_ptr *cfp*.array.iterator.**ptr**(*cfp_iter* self)

Return pointer to element referenced by iterator; `&*self`.

---

cfp_ptr *cfp*.array.iterator.**ptr_at**(*cfp_iter* self, ptrdiff_t d)

Return pointer to element offset *d* elements (may be negative) from iterator; `&self[d]`.

---

size_t *cfp*.array.iterator.**i**(const *cfp_iter* self)

size_t *cfp*.array.iterator.**j**(const *cfp_iter* self)

size_t *cfp*.array.iterator.**k**(const *cfp_iter* self)

size_t *cfp*.array.iterator.**l**(const *cfp_iter* self)

Return *i*, *j*, *k*, and *l* component of array element referenced by iterator; see *iterator::i()*, *iterator::j()*, *iterator::k()*, and *iterator::l()*.

---

zfp_bool `cfp.array.iterator.lt`(const *cfp_iter* lhs, const *cfp_iter* rhs)

zfp_bool `cfp.array.iterator.gt`(const *cfp_iter* lhs, const *cfp_iter* rhs)

zfp_bool `cfp.array.iterator.leq`(const *cfp_iter* lhs, const *cfp_iter* rhs)

zfp_bool `cfp.array.iterator.geq`(const *cfp_iter* lhs, const *cfp_iter* rhs)

Return true if the two iterators satisfy the given *relationship*; `lhs < rhs`, `lhs > rhs`, `lhs <= rhs`, `lhs >= rhs`.

---

zfp_bool `cfp.array.iterator.eq`(const *cfp_iter* lhs, const *cfp_iter* rhs)

Return whether two iterators are equal; `lhs == rhs`. See *`iterator::operator==()`*.

---

zfp_bool `cfp.array.iterator.neq`(const *cfp_iter* lhs, const *cfp_iter* rhs)

Return whether two iterators are not equal; `lhs != rhs`. See *`iterator::operator!=()`*.

---

ptrdiff_t `cfp.array.iterator.distance`(const *cfp_iter* first, const *cfp_iter* last)

Return the difference between two iterators; `last - first`. See *`iterator::operator-()`*.

---

cfp_iter `cfp.array.iterator.next`(const *cfp_iter* it, ptrdiff_t d)

Return the result of advancing iterator by *d* elements; `it + d`. See *`iterator::operator+()`*.

---

cfp_iter `cfp.array.iterator.prev`(const *cfp_iter* it, ptrdiff_t d)

Return the result of decrementing iterator by *d* elements; `it - d`. See *`iterator::operator-()`*.

---

cfp_iter `cfp.array.iterator.inc`(const *cfp_iter* it)

Return the result of incrementing iterator by one element; `it + 1`. See *`iterator::operator++()`*.

---

cfp_iter `cfp.array.iterator.dec`(const *cfp_iter* it)

Return the result of decrementing iterator by one element; `it - 1`. See *`iterator::operator--()`*.

CHAPTER

FOURTEEN

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, 2D, or 4D arrays.

14.1 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.

Note: In multidimensional arrays, the order in which dimensions are specified is important. In zfp, the memory layout convention is such that *x* varies faster than *y*, which varies faster than *z*, and hence *x* should map to the innermost (rightmost) array dimension in a C array and to the leftmost dimension in a Fortran array. Getting the order of dimensions right is crucial for good compression and accuracy. See the discussion of *dimensions and strides* and FAQ *#0* for further information.

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 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, 2D, and 4D 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, zfp_false);
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);
void* buffer = malloc(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);
```

Compression can be accelerated via OpenMP multithreading (since zfp 0.5.3) and CUDA (since zfp 0.5.4). To enable OpenMP parallel compression, call:

```
if (!zfp_stream_set_execution(zfp, zfp_exec_omp)) {
  // OpenMP not available; handle error
}
```

See the section *Parallel Execution* for further details on how to configure zfp and its run-time parameters for parallel compression.

Finally, the array is compressed as follows:

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

If the stream was rewound before calling *zfp_compress()*, 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 should 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);
size_t size = zfp_decompress(zfp, field);
```

The return value is zero if the decompressor failed.

14.1.1 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, size_t nx, size_t ny, size_t 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, zfp_false);   // 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
```

(continues on next page)

(continued from previous page)

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

14.2 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 previous section) 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 */ };
size_t 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 write 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 */ };
ptrdiff_t sx = &block[0][0][1] - &block[0][0][0]; // x stride =  1
ptrdiff_t sy = &block[0][1][0] - &block[0][0][0]; // y stride =  4
ptrdiff_t sz = &block[1][0][0] - &block[0][0][0]; // z stride = 16
size_t bits = zfp_encode_block_strided_double_3(zfp, &block[0][0][0], sx, sy, sz);
```

The strides are measured in number of array elements, 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. For example, 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*.

14.3 Compressed C++ Arrays

The zfp compressed-array 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]`, nested indexing `a[k][j][i]` (via *nested views*), or using multidimensional indexing via `a(i)`, `a(i, j)`, `a(i, j, k)`, or `a(i, j, k, l)`. 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's compressed arrays do not support special floating-point values like infinities and NaNs, although subnormal 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. These classes are wrapped in the `zfp` namespace.

Currently, there are eight array classes for 1D, 2D, 3D, and 4D 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] = { 0.0 };
```

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: In multidimensional arrays, the order in which dimensions are specified is important. In zfp, the memory layout convention is such that *x* varies faster than *y*, which varies faster than *z*, and hence *x* should map to the innermost (rightmost) array dimension in a C array and to the leftmost dimension in a Fortran array. Getting the order of dimensions right is crucial for good compression and accuracy. See the discussion of *dimensions and strides* and FAQ *#0* for further information.

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
4D arrays: 1/4-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
4D arrays: 1/32-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, rate, 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).

As of zfp 0.5.3, entire arrays may be copied via the copy constructor or assignment operator:

```
zfp::array3d b(a); // declare array b to be a copy of array a
zfp::array3d c; // declare empty array c
c = a; // copy a to c
```

Copies are deep and have value (not reference) semantics. In the above example, separate storage for *b* and *c* is allocated, and subsequent modifications to *b* and *c* will not modify *a*.

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
void* 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 an *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.

zfp 0.5.4 adds views to arrays, which among other things can be used to perform nested indexing:

```
zfp::array3d::nested_view v(&a);
double value = v[k][j][i];
```

A view is a shallow copy of an array or a subset of an array.

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. For example, in

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

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

When the array is const qualified, the operators `[]` and `()` are inspectors that return a proxy *const reference* that implicitly converts to a value. If used as arguments in `printf` or other functions that take a variable number of arguments, implicit conversion is not done and the reference has to be explicitly cast to value, e.g., `printf("%f", (double)a[i]);`.

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

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

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

```
void resize(size_t nx, size_t ny, size_t nz, size_t nw, 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.

As of zfp 1.0.0, all iterators for 1D-4D arrays support random access. 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++) {
  size_t i = it.i();
  size_t j = it.j();
  size_t 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, size_t count)
{
  double s = 0;
  for (size_t 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)
```

(continues on next page)

(continued from previous page)

```
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. Like 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
const double&                        zfp::array1d::const_reference
const double*                        zfp::array1d::const_pointer
std::vector<double>::const_iterator  zfp::array1d::const_iterator
```

14.3.1 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 accuracy (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 a default size of $\sqrt{n}$ blocks (rounded up to the next integer power of two) is used, where n is the total number of blocks in the array.

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

FIFTEEN

FILE COMPRESSOR

This section describes a simple, file-based zfp compression tool that is part of the zfp distribution named **zfp**. Other, third-party, file-based compression options are discussed in the *Application Support* section.

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 scalar type must be specified using *-f* (float) or *-d* (double), or using *-t* for integer-valued data. In addition, the array dimensions must be specified using *-1* (for 1D arrays), *-2* (for 2D arrays), *-3* (for 3D arrays), or *-4* (for 4D arrays). For multidimensional arrays, *x* varies faster than *y*, which in turn varies faster than *z*, and so on. That is, a 4D input file corresponding to a flattened C array `a[nw][nz][ny][nx]` is specified as `-4 nx ny nz nw`.

Note: 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.

In addition to ensuring correct dimensionality, the order of dimensions also matters. For instance, `-2 nx ny` is not equivalent to `-2 ny nx`, i.e., with the dimensions transposed.

Note: In multidimensional arrays, the order in which dimensions are specified is important. In zfp, the memory layout convention is such that x varies faster than y, which varies faster than z, and hence x should map to the innermost (rightmost) array dimension in a C array and to the leftmost dimension in a Fortran array. Getting the order of dimensions right is crucial for good compression and accuracy. See the discussion of *dimensions and strides* and FAQ *#0* for further information.

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`) mode. *Reversible mode* for lossless compression is specified using `-R`.

15.1 Usage

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

15.1.1 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.

15.1.2 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*.

15.1.3 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]`.

-4 `<nx> <ny> <nz> <nw>`

Dimensions of 4D C array `a[nw][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*, *-3*, or *-4* specifies the array dimensions. The same parameters must be given when decompressing data (without *-i*), unless a header was stored using *-h* during compression.

15.1.4 Compression parameters

One of the following *compression modes* must be selected.

-r `<rate>`

Specify fixed rate in terms of number of compressed bits per integer or 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.

-R

Reversible (lossless) mode.

-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.

15.1.5 Execution parameters

-x <policy>

Specify execution policy and parameters. The default policy is `-x serial` for sequential execution. To enable OpenMP parallel compression, use the `omp` policy. Without parameters, `-x omp` selects OpenMP with default settings, which typically implies maximum concurrency available. Use `-x omp=threads` to request a specific number of threads (see also *zfp_stream_set_omp_threads()*). A thread count of zero is ignored and results in the default number of threads. Use `-x omp=threads,chunk_size` to specify the chunk size in number of blocks (see also *zfp_stream_set_omp_chunk_size()*). A chunk size of zero is ignored and results in the default size. Use `-x cuda` to for parallel CUDA compression and decompression.

As of 0.5.4, the execution policy applies to both compression and decompression. If the execution policy is not supported for decompression, then zfp will attempt to fall back on serial decompression. This is done only when both compression and decompression are performed as part of a single execution, e.g., when specifying both *-i* and *-o*.

15.1.6 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 $100 \times 100 \times 100$ 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
- `-x omp=16,256` : parallel compression with 16 threads, 256-block chunks

CHAPTER SIXTEEN

CODE EXAMPLES

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

16.1 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*.

16.2 Compressed-Array C++ Classes

The **`array`** program shows how to declare, write to, and read from zfp's compressed-array C++ objects (in this case, 2D double-precision arrays), which is essentially as straightforward as working with STL vectors. This example initializes a 2D array with a linear ramp of $12 \times 8 = 96$ values using only four bits of storage per value, which using uncompressed storage would not be enough to distinguish more than 16 different values. For more advanced compressed-array features, see the *tutorial*.

16.3 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 parameters and cache size affects the error in the solution and solution time. The usage is:

```
diffusion [options]
  -a <tolerance> : absolute error tolerance (requires -c)
  -b <blocks> : cache size in number of 4x4 blocks
  -c : use read-only arrays (needed for -a, -p, -R)
  -d : use double-precision tiled arrays
  -f : use single-precision tiled arrays
  -h : use half-precision tiled arrays
  -i : traverse arrays using iterators instead of integer indices
  -j : use OpenMP parallel execution (requires -r)
```

(continues on next page)

(continued from previous page)

```
  -n <nx> <ny> : grid dimensions
  -p <precision> : precision in uncompressed bits/value (requires -c)
  -r <rate> : rate in compressed bits/value
  -R : reversible mode (requires -c)
  -t <nt> : number of time steps
```

Here *rate* specifies the exact number of compressed bits to store per double-precision floating-point value; *nx* and *ny* specify the grid size (default = 128 × 128); *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.

The `-j` option enables OpenMP parallel execution, which makes use of both mutable and immutable *private views* for thread-safe array access. Note that this example has not been optimized for parallel performance, but rather serves to show how to work with zfp's compressed arrays in a multithreaded setting.

This example also illustrates how *read-only arrays* (`-c`) may be used in conjunction with fixed-rate (`-r`), fixed-precision (`-p`), fixed-accuracy (`-a`), or reversible (`-R`) mode.

The output lists for each time step the current rate of the state array and in parentheses any additional storage, e.g., for the block *cache* and *index* data structures, both in bits per array element. Running diffusion with the following arguments:

```
diffusion -r 8
diffusion -r 12
diffusion -r 16
diffusion -r 24
diffusion
```

should result in this final output:

```
sum=0.995170 error=4.044954e-07
sum=0.998151 error=1.237837e-07
sum=0.998345 error=1.212734e-07
sum=0.998346 error=1.212716e-07
sum=0.998346 error=1.212716e-07
```

For speed and quality comparison, the solver solves the same problem using uncompressed double-precision row-major arrays when compression parameters are omitted. If one of `-h`, `-f`, `-d` is specified, uncompressed tiled arrays are used. These arrays are based on the zfp array classes but make use of the *generic codec*, which stores blocks as uncompressed scalars of the specified type (`half`, `float`, or `double`) while utilizing a double-precision block cache (like zfp's compressed arrays).

The **`diffusionC`** program is the same example written entirely in C using the cfp *wrappers* around the C++ compressed array classes.

16.4 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.

16.5 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.

16.6 PPM Image Compression

The **ppm** program is analogous to the **pgm** example, but has been designed for compressing color images in the ppm format. Rather than compressing RGB channels independently, ppm exploits common strategies for color image compression such as color channel decorrelation and chroma subsampling.

The usage is essentially the same as for *pgm*:

```
ppm <param> <input.ppm >output.ppm
```

where a positive param specifies the rate in bits per pixel; when negative, it specifies the precision (number of bit planes to encode) in fixed-precision mode.

16.7 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.

16.8 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.

The **iteratorC** example illustrates the equivalent cfp iterator operations. It closely follows the usage shown in the **iterator** example with some minor differences. It likewise takes no command-line options.

CHAPTER SEVENTEEN

REGRESSION TESTS

The **testzfp** program performs basic regression testing by exercising a small but important subset of `libzfp` and the compressed-array classes. It serves as a sanity check that zfp has been built properly. These tests assume the default compiler settings, i.e., with none of the settings in `Config` or `CMakeLists.txt` modified. By default, small, synthetic floating-point arrays are used in the test. To test larger arrays, use the `large` command-line option. When large arrays are used, the (de)compression throughput is also measured and reported in number of uncompressed bytes per second.

More extensive unit and functional tests are available on the zfp GitHub develop branch in the `tests` directory.

CHAPTER

EIGHTEEN

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 us.

Questions answered in this FAQ:

Q0: *Do zfp arrays use C or Fortran order?*

This is such an important question that we added it as question zero to our FAQ, but do not let this C'ism fool you.

A: zfp *compressed-array classes* and uncompressed *fields* assume that the leftmost index varies fastest, which often is referred to as Fortran order. By convention, zfp uses *x* (or *i*) to refer to the leftmost index, then *y* (or *j*), and so on.

Warning: It is critical that the order of dimensions is specified correctly to achieve good compression and accuracy. If the order of dimensions is transposed, zfp will still compress the data, but with no indication that the order was wrong. Compression ratio and/or accuracy will likely suffer significantly, however. Please see *this section* for further discussion.

In C order, the rightmost index varies fastest (e.g., *x* in `arr[z][y][x]`), meaning that if we increment the rightmost index we move to the next consecutive address in memory. If an uncompressed array, `arr`, is stored in C order, we would for compatibility with zfp let *x* be the rightmost index in `arr` but the leftmost index in the compressed zfp array, `zarr`, e.g.,:

```
const size_t nx = 5;
const size_t ny = 3;
const size_t nz = 2;
float arr[nz][ny][nx] = { ... };
zfp::array3<float> zarr(nx, ny, nz, rate, &a[0][0][0]);
```

Then `arr[z][y][x]` and `zarr(x, y, z)` refer to the same element, as do `(&arr[0][0][0])[sx * x + sy * y + sz * z]` and `zarr[sx * x + sy * y + sz * z]`, where

```
ptrdiff_t sx = &arr[0][0][1] - &arr[0][0][0]; // sx = 1
ptrdiff_t sy = &arr[0][1][0] - &arr[0][0][0]; // sy = nx = 5
ptrdiff_t sz = &arr[1][0][0] - &arr[0][0][0]; // sz = nx * ny = 15
```

Here *sx*, *sy*, and *sz* are the *strides* along the three dimensions, with $sx < sy < sz$.

Of course, C vs. Fortran ordering matters only for multidimensional arrays and when the array dimensions (*nx*, *ny*, *nz*) are not all equal.

Note that zfp *fields* also support strides, which can be used to represent more general layouts than C and Fortran order, including non-contiguous storage, reversed dimensions via negative strides, and other advanced layouts. With the default strides, however, it is correct to think of zfp as using Fortran order.

For uncompressed data stored in C order, one easily translates to zfp Fortran order by reversing the order of dimensions or by specifying appropriate *strides*. We further note that zfp provides *nested views* of arrays that support C indexing syntax, e.g., `view[z][y][x]` corresponds to `arr(x, y, z)`.

Note: The zfp *NumPy interface* uses the strides of the NumPy array to infer the correct layout. Although NumPy arrays use C order by default, zfp handles such arrays correctly regardless of their memory layout. The actual order of dimensions for compressed storage are, however, reversed so that NumPy arrays in C order are traversed sequentially during compression.

Why does zfp use Fortran order when C is today a far more common language? This choice is somewhat arbitrary yet has strong proponents in either camp, similar to the preference between *little and big endian* byte order. We believe that a single 2D array storing an (*x*, *y*) image is most naturally extended to a sequence of *nt* time-varying images by *appending* (not prepending) a time dimension *t* as (*x*, *y*, *t*). This is the convention used in mathematics, e.g., we use (*x*, *y*) coordinates in 2D and (*x*, *y*, *z*) coordinates in 3D. Using Fortran order, each time slice, *t*, is still a 2D contiguous image, while C order (`arr[x][y][t]`) would suggest that appending the *t* dimension now gives us *nx* 2D arrays indexed by (*y*, *t*), even though without the *t* dimension the images would be indexed by (*x*, *y*).

Q1: *Can zfp compress vector fields?*

I have a 2D vector field

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

of dimensions $nx \times ny$. 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 assuredly 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 $nx \times ny$ 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 (size_t z = 0; z < nz; z++)
  for (size_t y = 0; y < ny; y++)
    for (size_t 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 $nx \times ny$ 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: Yes, but only in *reversible mode*.

zfp's lossy compression modes currently support only finite floating-point values. 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 *Utility Functions*). 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: Yes, this can safely be done in *reversible mode*.

In other (lossy) modes, the answer depends. 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 data, zfp supports a rate granularity of $1 / 4^d$ bits, i.e., the rate can be specified in increments of a fraction of a bit. Such fine rate selection is always available for sequential compression (e.g., when calling `zfp_compress()`).

Unlike in sequential compression, zfp's *read-write compressed-array classes* 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, 1 bit in 3D, and 0.25 bits in 4D. *Read-only compressed arrays* support the same fine granularity as sequential compression.

To achieve finer granularity, build 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. When decompressing one block at a time, it is possible to use more tightly constrained `zfp_stream` parameters during decompression than were used during compression. For instance, one may use a 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. Furthermore, the parameters must agree between compression and decompression when calling the high-level API function `zfp_decompress()`.

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: First, zfp does not support *fixed-accuracy mode* for integer data and will ignore any tolerance requested via *zfp_stream_set_accuracy()* or associated *expert mode* parameter settings. So this FAQ pertains to floating-point data only.

The short answer is that, given finite precision, the zfp and IEEE floating-point number systems represent distinct subsets of the reals (or, in case of zfp, blocks of reals). Although these subsets have significant overlap, they are not equal. Consequently, there are some combinations of floating-point values that zfp cannot represent exactly; conversely, there are some zfp blocks that cannot be represented exactly as IEEE floating point. If the user-specified tolerance is smaller than the difference between the IEEE floating-point representation to be compressed and its closest zfp representation, then the tolerance necessarily will be violated (except in *reversible mode*). In practice, absolute tolerances have to be extremely small relative to the numbers being compressed for this issue to occur, however.

Note that this issue is not particular to zfp but occurs in the conversion between any two number systems of equal precision; we may just as well fault IEEE floating point for not being able to represent all zfp blocks accurately enough! By analogy, not all 32-bit integers can be represented exactly in 32-bit floating point. The integer 123456789 is one example; the closest float is 123456792. And, obviously, not all floats (e.g., 0.5) can be represented exactly as integers.

To further demonstrate this point, let us consider a concrete example. zfp does not store each floating-point scalar value independently but represents a group of values (4, 16, 64, or 256 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 rounded result of the addition, giving `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. Even when write random access is not requested, the compressed stream is written in units of *wordsize*. Hence, once the stream is flushed, either by a `zfp_compress()` or `zfp_stream_flush()` call, to output any buffered bits, its size will be a multiple of *wordsize* 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.

Aside from these caveats, zfp is guaranteed to meet the exact rate specified.

Q19: *Can zfp perform compression in place?*

A: 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 * sizeof(type) * 8`, where *d* is the array dimensionality (1, 2, 3, or 4) 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: *Can zfp bound the point-wise relative error?*

A: Yes, but with some caveats. First, we define the relative error in a value f approximated by g as $|f - g| / |f|$, which converges to $|\log(f / g)| = |\log(f) - \log(g)|$ as g approaches f, where $\log(f)$ denotes the natural logarithm of f. Below, we discuss three strategies for relative error control that may be applicable depending on the properties of the underlying floating-point data.

If all floating-point values to be compressed are normalized, i.e., with no nonzero subnormal values smaller in magnitude than $2^{-126} \approx 10^{-38}$ (for floats) or $2^{-1022} \approx 10^{-308}$ (for doubles), then the relative error can be bounded using zfp's *expert mode* settings by invoking *reversible mode*. This is achieved by truncating (zeroing) some number of least significant bits of all floating-point values and then losslessly compressing the result. The q least significant bits of n-bit floating-point numbers (n = 32 for floats and n = 64 for doubles) are truncated by zfp by specifying a maximum precision of $p = n - q$. The resulting point-wise relative error is then at most $2^{q - 23}$ (for floats) or $2^{q - 52}$ (for doubles).

Note: For large enough q, floating-point exponent bits will be discarded, in which case the bound no longer holds, but then the relative error is already above 100%. Also, as mentioned, the bound does not hold for subnormals; however, such values are likely too small for relative errors to be meaningful.

To bound the relative error, set the expert mode parameters to:

```
minbits = 0
maxbits = 0
maxprec = p
minexp = ZFP_MIN_EXP - 1 = -1075
```

For example, using the **zfp** command-line tool, set the parameters using `-c 0 0 p -1075`.

Note that while the above approach respects the error bound when the above conditions are met, it uses zfp for a purpose it was not designed for, and the compression ratio may not be competitive with those obtained using compressors designed to bound the relative error.

Other forms of relative error control can be achieved using zfp's lossy compression modes. In *fixed-accuracy mode*, the *absolute error* $|f - g|$ is bounded by a user-specified error tolerance. For a field whose values are all positive (or all negative), we may pre-transform values by taking the natural logarithm, replacing each value f with $\log(f)$ before compression, and then exponentiating values after decompression. This ensures that $|\log(f) - \log(g)| = |\log(f / g)|$ is bounded. (Note, however, that many implementations of the math library make no guarantees on the accuracy of the logarithm function.) For fields whose values are signed, an approximate bound can be achieved by using $\log(f) \approx \text{asinh}(f / 2)$, where asinh is the inverse of the hyperbolic sine function, which is defined for both positive and negative numbers. One benefit of this approach is that it de-emphasizes the importance of relative errors for small values that straddle zero, where relative errors rarely make sense, e.g., because of round-off and other errors already present in the data.

Finally, in *fixed-precision mode*, the precision of zfp transform coefficients is fixed, resulting in an error that is no more than a constant factor of the largest (in magnitude) value, *fmax*, within the same zfp block. This can be thought of as a weaker version of relative error, where the error is measured relative to values in a local neighborhood.

In fixed-precision mode, 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*).

As alluded to, it is possible to bound the error relative to the largest 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, 3D, and 4D we have:

```
k(1) = 20
k(2) = 125
k(3) = 1125/4
k(4) = 16876/16
```

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
p(n, 4) = n + 11
```

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?*

A: Yes. As of zfp 0.5.5, bit-for-bit lossless compression is supported via the *reversible compression mode*. This mode supports both integer and floating-point data.

In addition, it is sometimes possible to ensure lossless compression using zfp's fixed-precision and fixed-accuracy modes. For integer data, zfp can with few exceptions ensure lossless compression in *fixed-precision mode*. 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 integer compression in fixed-precision mode 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 ($1 \leq d \leq 4$). 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
4	15	47

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. Therefore, we recommend using the *reversible mode* when lossless compression is desired.

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 usually requires *reversible mode*, as the other compression modes are unlikely to guarantee bit-for-bit exact reconstructions. However, 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. Finally, to preserve special values such as negative zero, plus and minus infinity, and NaNs, reversible mode is needed.

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

A: 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 (see *Q30*), 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.

Q23: *Are parallel compressed streams identical to serial streams?*

A: Yes, it matters not what execution policy is used; the final compressed stream produced by `zfp_compress()` depends only on the uncompressed data and compression settings.

To support future parallel decompression, in particular variable-rate streams, it will be necessary to also store an index of where (at what bit offset) each compressed block is stored in the stream. Extensions to the current zfp format are being considered to support parallel decompression.

Regardless, the execution policy and parameters such as number of threads do not need to be the same for compression and decompression.

Q24: *Are zfp's compressed arrays and other data structures thread-safe?*

A: Yes, compressed arrays can be made thread-safe; no, data structures like `zfp_stream` and `bitstream` are not necessarily thread-safe. As of zfp 0.5.4, thread-safe read and write access to compressed arrays via OpenMP threads is provided through the use of *private views*, although these come with certain restrictions and requirements such as the need for the user to enforce cache coherence. Please see the documentation on *views* for further details.

As far as C objects, zfp's parallel OpenMP compressor assigns one `zfp_stream` per thread, each of which uses its own private `bitstream`. Users who wish to make parallel calls to zfp's *low-level functions* are advised to consult the source files `ompcompress.c` and `parallel.c`.

Finally, the zfp API is thread-safe as long as multiple threads do not simultaneously call API functions and pass the same `zfp_stream` or `bitstream` object.

Q25: *Why does parallel compression performance not match my expectations?*

A: zfp partitions arrays into chunks and assigns each chunk to an OpenMP thread. A chunk is a sequence of consecutive d-dimensional blocks, each composed of 4^d values. If there are fewer chunks than threads, then full processor utilization will not be achieved.

The number of chunks is by default set to the number of threads, but can be modified by the user via `zfp_stream_set_omp_chunk_size()`. One reason for using more chunks than threads is to provide for better load balance. If compression ratios vary significantly across the array, then threads that process easy-to-compress blocks may finish well ahead of threads in charge of difficult-to-compress blocks. By breaking chunks into smaller units, OpenMP is given the opportunity to balance the load better (though the effect of using smaller chunks depends on OpenMP thread scheduling). If chunks are too small, however, then the overhead of allocating and initializing chunks and assigning threads to them may dominate. Experimentation with chunk size may improve performance, though chunks ought to be at least several hundred blocks each.

In variable-rate mode, compressed chunk sizes are not known ahead of time. Therefore the compressed chunks must be concatenated into a single stream following compression. This task is performed sequentially on a single thread, and will inevitably limit parallel efficiency.

Other reasons for poor parallel performance include compressing arrays that are too small to offset the overhead of thread creation and synchronization. Arrays should ideally consist of thousands of blocks to offset the overhead of setting up parallel compression.

Q26: *Why are compressed arrays so slow?*

A: This is likely due to the use of a very small cache. Prior to zfp 0.5.5, all arrays used two 'layers' of blocks as default cache size, which is reasonable for 2D and higher-dimensional arrays (as long as they are not too 'skinny'). In 1D, however, this implies that the cache holds only two blocks, which is likely to cause excessive thrashing.

As of version 0.5.5, the default cache size is roughly proportional to the square root of the total number of array elements, regardless of array dimensionality. While this tends to reduce thrashing, we suggest experimenting with larger cache sizes of at least a few kilobytes to ensure acceptable performance.

Note that compressed arrays constructed with the *default constructor* will have an initial cache size of only one block. Therefore, users should call `array::set_cache_size()` after *resizing* such arrays to ensure a large enough cache.

Depending on factors such as rate, cache size, array access pattern, array access primitive (e.g., indices vs. iterators), and arithmetic intensity, we usually observe an application slow-down of 1-10x when switching from uncompressed to compressed arrays.

Q27: *Do compressed arrays use reference counting?*

A: It is possible to reference compressed-array elements via proxy *references* and *pointers*, through *iterators*, and through *views*. Such indirect references are valid only during the lifetime of the underlying array. No reference counting and garbage collection is used to keep the array alive if there are external references to it. Such references become invalid once the array is destructed, and dereferencing them will likely lead to segmentation faults.

Q28: *How large a buffer is needed for compressed storage?*

A: `zfp_compress()` requires that memory has already been allocated to hold the compressed data. But often the compressed size is data dependent and not known a priori. The function `zfp_stream_maximum_size()` returns a buffer size that is guaranteed to be large enough. This function, which should be called *after* setting the desired compression mode and parameters, computes the largest possible compressed data size based on the current compression settings and array size. Note that by the pigeonhole principle, any (lossless) compressor must expand at least one input, so this buffer size may be larger than the size of the uncompressed input data. `zfp_compress()` returns the actual number of bytes of compressed storage.

When compressing individual blocks using the *low-level API*, it is useful to know the maximum number of bits that a compressed block can occupy. In addition to the `ZFP_MAX_BITS` macro, the following table lists the maximum block size (in bits) for each scalar type, whether *reversible mode* is used, and block dimensionality.

type	rev.	1D	2D	3D	4D
int32		131	527	2111	8447
	✓	136	532	2116	8452
float		140	536	2120	8456
	✓	146	542	2126	8462
int64		259	1039	4159	16639
	✓	265	1045	4165	16645
double		271	1051	4171	16651
	✓	278	1058	4178	16658

Q29: *How can I print array values?*

Consider the following seemingly reasonable piece of code:

```
#include <cstdio>
#include "zfp/array1.hpp"

int main()
{
  zfp::array1<double> a(100, 16.0);
  printf("%f\n", a[0]); // does not compile
  return 0;
}
```

The compiler will complain about `a[0]` being a non-POD object. This is because `a[0]` is a *proxy reference* object rather than a `double`. To make this work, `a[0]` must be explicitly converted to `double`, e.g., using a cast:

```
printf("%f\n", (double)a[0]);
```

For similar reasons, one may not use `scanf` to initialize the value of `a[0]` because `&a[0]` is a *proxy pointer* object, not a `double*`. Rather, one must use a temporary variable, e.g.

```
double t;
scanf("%lf", &t);
a[0] = t;
```

Note that using `iostream`, expressions like

```
std::cout << a[0] << std::endl;
```

do work, but

```
std::cin >> a[0];
```

does not.

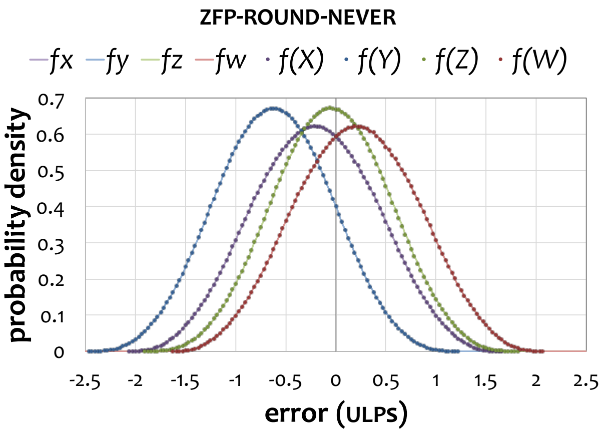
Q30: *What is known about zfp compression errors?*

A: Significant effort has been spent on characterizing compression errors resulting from zfp, as detailed in the following publications:

- P. Lindstrom, "Error Distributions of Lossy Floating-Point Compressors," JSM 2017 Proceedings.
- J. Diffenderfer, A. Fox, J. Hittinger, G. Sanders, P. Lindstrom, "Error Analysis of ZFP Compression for Floating-Point Data," SIAM Journal on Scientific Computing, 2019.
- D. Hammerling, A. Baker, A. Pinard, P. Lindstrom, "A Collaborative Effort to Improve Lossy Compression Methods for Climate Data," 5th International Workshop on Data Analysis and Reduction for Big Scientific Data, 2019.
- A. Fox, J. Diffenderfer, J. Hittinger, G. Sanders, P. Lindstrom. "Stability Analysis of Inline ZFP Compression for Floating-Point Data in Iterative Methods," SIAM Journal on Scientific Computing, 2020.

In short, zfp compression errors are roughly normally distributed as a consequence of the central limit theorem, and can be bounded. Because the error distribution is normal and because the worst-case error is often much larger than errors observed in practice, it is common that measured errors are far smaller than the absolute error tolerance specified in *fixed-accuracy mode* (see *Q22*).

It is known that zfp errors can be slightly biased and correlated (see Fig. 18.1 and the third paper above). Recent work has been done to combat such issues by supporting optional *rounding modes*.



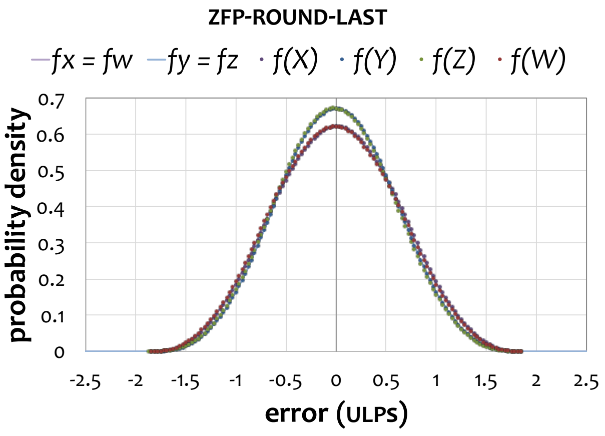


Fig. 18.1: zfp errors are normally distributed. This figure illustrates the agreement between theoretical (lines) and observed (dots) error distributions (X, Y, Z, W) for 1D blocks. Without proper rounding (left), errors are biased and depend on the relative location within a zfp block, resulting in errors not centered on zero. With proper rounding (right), errors are both smaller and unbiased.

CHAPTER

NINETEEN

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. *Have fill values, NaNs, and infinities been removed?*
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, and a time-varying volume is a 4D 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, 3D, and 4D 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 $100 \times 4 \times 4 = 1600$ 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]
```

As of version 0.5.4, zfp supports compression of 4D arrays. Since all dimensions in this example are likely to be correlated, the 4D array can be compressed directly. Alternatively, the data could be organized by the three "smoothest" dimensions and compressed as a 3D array. Given the organization above, the array could be treated as 3D:

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

Again, do **not** compress this as a 3D array with the *innermost* dimensions unfolded:

```
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 likely suffer greatly. See *this FAQ* for more details.

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 $4 \times 4 \times 4$ 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 $(8 \times 16 \times 4) / (5 \times 14 \times 2) \sim 3.66$ 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:

```
0.000000000000000000000000000000314159... * 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 lossy *modes of compression* that trade storage and accuracy, plus one *lossless mode*. 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.

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.

As of zfp 0.5.5, reversible (lossless) compression is available. The amount of lossless reduction of floating-point data is usually quite limited, however, especially for double-precision data. Unless a bit-for-bit exact reconstruction is needed, we strongly advocate the use of lossy compression.

Finally, 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

TWENTY

LIMITATIONS

zfp has evolved over the years from a research prototype to a production quality library. 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.

- Special floating-point values like infinity and NaN are supported in reversible mode but not in zfp's lossy compression modes. Subnormal floating-point numbers are, however, correctly handled. There is an implicit assumption that floating point conforms to IEEE-754, though extensions to other floating-point formats should be possible with minor effort.
- The optional zfp *header* supports arrays with at most 2^{48} elements. The zfp header limits each dimension to $2^{48/d}$ elements in a d-dimensional array, i.e., 2^{48}, 2^{24}, 2^{16}, and 2^{12} for 1D through 4D arrays, respectively. Note that this limitation applies only to the header; array dimensions are otherwise limited only by the size supported by `size_t`.
- The *compressed-array classes* have additional size restrictions. The *cache* supports at most 2^{p-1} - 1 blocks, where p is the number of bits in a `uint` (usually p = 32). Consequently, the number of elements in a d-dimensional compressed array is at most $4^d \times (2^{p-1} - 1)$, or about 8 billion elements for 1D arrays.
- 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.
- The *read-only array classes* do not yet support (de)serialization.
- 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 floating 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. The zfp compressed-array classes do not yet support integer scalar types.
- 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.
- Version 0.5.3 adds support for OpenMP compression. However, OpenMP decompression is not yet supported.
- Version 0.5.4 adds support for CUDA compression and decompression. However, only the fixed-rate compression mode is so far supported. The CUDA implementation is further subject to *additional limitations*.

- The cfp *C wrappers* for zfp's compressed arrays support only a subset of the C++ API. zfp 1.0.0 adds support for proxy references, pointers, and iterators, but views and read-only arrays are not yet supported. Furthermore, cfp works only with the zfp codec.
- The Python and Fortran bindings do not yet support zfp's compressed-array classes. Moreover, only a select subset of the *high-level API* is available via Python.

CHAPTER

TWENTYONE

FUTURE DIRECTIONS

zfp is actively being developed and plans have been made to add a number of important features, including:

- **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 supported by zfp only in *reversible mode*. In fact, for all lossy compression modes, 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.
- **Support for more general data types**. zfp currently does not directly support half and quad precision floating point. Nor is there support for 8- and 16-bit integers. With the emergence of new number representations like *posits* and *bfloat16*, we envision the need for a more general interface and a single unified zfp representation that would allow for *conversion* between zfp and *any* number representation. We are working on developing an uncompressed interchange format that acts like an intermediary between zfp and other number formats. This format decouples the zfp compression pipeline from the external number type and allows new number formats to be supported via user-defined conversion functions to and from the common interchange format.
- **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 available that produces compressed streams that are identical to serially compressed streams. However, parallel decompression is not yet supported. zfp also supports compression and decompression on the GPU via CUDA. However, only fixed-rate mode is so far supported.
- **Variable-rate arrays**. zfp currently offers only fixed-rate compressed arrays with random-access write support; zfp 1.0.0 further provides read-only variable-rate arrays. Fixed-rate arrays waste 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. A candidate solution has been developed for variable-rate arrays that support read-write random access with modest storage overhead. We expect to release this capability in the near future.
- **Array operations**. zfp's compressed arrays currently support basic indexing and initialization, but lack array-wise operations such as arithmetic, reductions, etc. Some such operations can exploit the higher precision (than

IEEE-754) supported by zfp, as well as accelerated blockwise computations that need not fully decompress and convert the zfp representation to IEEE-754.

- **Language bindings**. The main compression codec is written in C89 to facilitate calls from other languages. zfp's compressed arrays, on the other hand, are written in C++. zfp 0.5.4 and 0.5.5 add C wrappers around compressed arrays and Fortran and Python bindings to the high-level C API. Work is planned to provide additional language bindings for C, C++, Fortran, and Python to expose the majority of zfp's capabilities through all of these programming languages.

Please contact us with requests for features not listed above.

CHAPTER TWENTYTWO

CONTRIBUTORS

- zfp development team
  - Peter Lindstrom
  - Danielle Asher
- Major contributors
  - Chuck Atkins
  - Stephen Herbein
  - Mark Kim
  - Matt Larsen
  - Mark Miller
  - Markus Salasoo
  - David Wade
  - Haiying Xu

For a full list of contributors, see the GitHub Contributors page.

CHAPTER TWENTYTHREE

RELEASE NOTES

23.1 1.0.1 (2023-12-15)

This patch release primarily addresses minor bug fixes and is needed to update the zfpy Python wheels.

Added

- A new build macro, `BUILD_TESTING_FULL`, specifies that all unit tests be built; `BUILD_TESTING` produces a smaller subset of tests. Full tests and documentation are now included in releases.

Fixed

- #169: *libm* dependency is not always correctly detected.
- #171: *ptrdiff_t* is not always imported in Cython.
- #176: cfp API is not exposed via CMake configuration file.
- #177: Full test suite is not included in release.
- #181: *rpath* is not set correctly in executables.
- #204: Array strides are not passed by value in zFORp.
- #220: Errors reported with scikit-build when building zfpy.

23.2 1.0.0 (2022-08-01)

This release is not ABI compatible with prior releases due to numerous changes to function signatures and data structures like `zfp_field`. However, few of the API changes, other than to the cfp C API for compressed arrays, should impact existing code. Note that numerous header files have been renamed or moved relative to prior versions.

Added

- `zfp::const_array`: read-only variable-rate array that supports fixed-precision, fixed-accuracy, and reversible modes.
- Compressed-array classes for 4D data.
- `const` versions of array references, pointers, and iterators.
- A more complete API for pointers and iterators.
- cfp support for proxy references and pointers, iterators, and (de)serialization.
- Support for pointers and iterators into array views.

- `zfp::array::size_bytes()` allows querying the size of different components of an array object (e.g., payload, cache, index, metadata, . . .).
- Templated C++ wrappers around the low-level C API.
- A generic codec for storing blocks of uncompressed scalars in zfp's C++ arrays.
- Additional functions for querying `zfp_field` and `zfp_stream` structs.
- `zfp_config`: struct that encapsulates compression mode and parameters.
- Rounding modes for reducing bias in compression errors.
- New examples: `array`, `iteratorC`, and `ppm`.

Changed

- Headers from `array/`, `cfp/include/`, and `include/` have been renamed and reorganized into a common `include/` directory.
  - The libzfp API is now confined to `zfp.h`, `zfp.hpp`, and `zfp.mod` for C, C++, and Fortran bindings, respectively. These all appear in the top-level `include/` directory upon installation.
  - C++ headers now use a `.hpp` suffix; C headers use a `.h` suffix.
  - C++ headers like `array/zfparray.h` have been renamed `zfp/array.hpp`.
  - C headers like `cfp/include/cfparrays.h` have been renamed `zfp/array.h`.
- `size_t` and `ptrdiff_t` replace `uint` and `int` for array sizes and strides in the array classes and C/Fortran APIs.
- `zfp_bool` replaces `int` as Boolean type in the C API.
- `bitstream_offset` and `bitstream_size` replace `size_t` to ensure support for 64-bit offsets into and lengths of bit streams. Consequently, the `bitstream` API has changed accordingly.
- All array and view iterators are now random-access iterators.
- Array inspectors now return `const_reference` rather than a scalar type like `float` to allow obtaining a `const_pointer` to an element of an immutable array.
- `zfp::array::compressed_data()` now returns `void*` instead of `uchar*`.
- The array (de)serialization API has been revised, resulting in new `zfp::array::header` and `zfp::exception` classes with new exception messages.
- The array `codec` class is now responsible for all details regarding compression.
- The compressed-array C++ implementation has been completely refactored to make it more modular, extensible, and reusable across array types.
- Array block shapes are now computed on the fly rather than stored.
- The cfp C API now wraps array objects in structs.
- The zfPy Python API now supports the more general `memoryview` over `bytes` objects for decompression.
- The zFORp Fortran module name is now `zfp` instead of `zforp_module`.
- Some command-line options for the `diffusion` example have changed.
- CMake 3.9 or later is now required for CMake builds.

Removed

- `zfp::array::get_header()` has been replaced with a `zfp::array::header` constructor that accepts an array object.
- `ZFP_VERSION_RELEASE` is no longer defined (use `ZFP_VERSION_PATCH`).

Fixed

- #66: `make install` overwrites googletest.
- #84: Incorrect order of parameters in CUDA `memset()`.
- #86: C++ compiler warns when `__STDC_VERSION__` is undefined.
- #87: `CXXFLAGS` is misspelled in `cfp/src/Makefile`.
- #98: `zfp_stream_maximum_size()` underestimates size in reversible mode.
- #99: Incorrect `private_view` reads due to missing writeback.
- #109: Unused CPython array is incompatible with PyPy.
- #112: PGI compiler bug causes issues with memory alignment.
- #119: All-subnormal blocks may cause floating-point overflow.
- #121: CUDA bit offsets are limited to 32 bits.
- #122: `make install` does not install zfp command-line utility.
- #125: OpenMP bit offsets are limited to 32 bits.
- #126: `make install` does not install Fortran module.
- #127: Reversible mode reports incorrect compressed block size.
- #150: cmocka tests do not build on macOS.
- #154: Thread safety is broken in `private_view` and `private_const_view`.
- `ZFP_MAX_BITS` is off by one.
- `diffusionC`, `iteratorC` are not being built with `gmake`.

23.3 0.5.5 (2019-05-05)

Added

- Support for reversible (lossless) compression of floating-point and integer data.
- Methods for serializing and deserializing zfp's compressed arrays.
- Python bindings for compressing NumPy arrays.
- Fortran bindings to zfp's high-level C API.

Changed

- The default compressed-array cache size is now a function of the total number of array elements, irrespective of array shape.

Fixed

- Incorrect handling of execution policy in zfp utility.
- Incorrect handling of decompression via header in zfp utility.
- Incorrect cleanup of device memory in CUDA decompress.
- Missing tests for failing mallocs.
- CMake does not install CFP when built.

- `zfp_write_header()` and `zfp_field_metadata()` succeed even if array dimensions are too large to fit in header.

23.4 0.5.4 (2018-10-01)

Added

- Support for CUDA fixed-rate compression and decompression.
- Views into compressed arrays for thread safety, nested array indexing, slicing, and array subsetting.
- C language bindings for compressed arrays.
- Support for compressing and decompressing 4D data.

Changed

- Execution policy now applies to both compression and decompression.
- Compressed array accessors now return Scalar type instead of `const Scalar&` to avoid stale references to evicted cache lines.

Fixed

- Incorrect handling of negative strides.
- Incorrect handling of arrays with more than 2^{32} elements in zfp command-line tool.
- `bitstream` is not C++ compatible.
- Minimum cache size request is not respected.

23.5 0.5.3 (2018-03-28)

Added

- Support for OpenMP multithreaded compression (but not decompression).
- Options for OpenMP execution in zfp command-line tool.
- Compressed-array support for copy construction and assignment via deep copies.
- Virtual destructors to enable inheritance from zfp arrays.

Changed

- `zfp_decompress()` now returns the number of compressed bytes processed so far, i.e., the same value returned by `zfp_compress()`.

23.6 0.5.2 (2017-09-28)

Added

- Iterators and proxy objects for pointers and references.
- Example illustrating how to use iterators and pointers.

Changed

- Diffusion example now optionally uses 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.
- Rewrote documentation in reStructuredText and added complete documentation of all public functions, classes, types, and macros.

Fixed

- Issue with setting stream word type via CMake.

23.7 0.5.1 (2017-03-28)

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.
- Example that illustrates in-place compression.
- Support for CMake builds.
- Documentation that discusses common issues with using zfp.

Changed

- Separated library version from CODEC version and added version string.
- Corrected inconsistent naming of `BIT_STREAM` macros in code and documentation.
- Renamed some of the header bit mask macros.
- `stream_skip()` and `stream_flush()` now return 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` and `uint` types in places to use `ptrdiff_t` and `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.

- Modified directory structure.

Fixed

- Bug that prevented defining uninitialized arrays.
- Incorrect computation of array sizes in `zfp_field_size()`.
- Minor issues that prevented code from compiling on Windows.
- Issue with fixed-accuracy headers that caused unnecessary storage.

23.8 0.5.0 (2016-02-29)

This version introduces backwards incompatible changes to the CODEC.

Added

- 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.
- 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

- 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.
- 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.

23.9 0.4.1 (2015-12-28)

Added

- Added `simple.c` as a minimal example of how to call the compressor.

Changed

- Changed compilation of diffusion example to output two executables: one with and one without compression.

Fixed

- Bug that caused segmentation fault when compressing 3D arrays whose dimensions are not multiples of four. Specifically, arrays of dimensions $nx \times ny \times nz$, 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.

23.10 0.4.0 (2015-12-05)

This version contains 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.
- New examples for evaluating the throughput of the (de)compressor and for compressing grayscale images in the pgm format.
- Frequently asked questions.

Changed

- 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.

Removed

- Support for general fixed-point decorrelating transforms.

23.11 0.3.2 (2015-12-03)

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.

23.12 0.3.1 (2015-05-06)

Fixed

- Rare bug caused by exponent underflow in blocks with no normal and some subnormal numbers.

23.13 0.3.0 (2015-03-03)

This version modifies 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. This change is not backwards compatible.

Added

- Compile-time support for parameterized transforms, e.g., to support other popular transforms like DCT, HCT, and Walsh-Hadamard.

- Floating-point traits to reduce the number of template parameters. It is now possible to declare a 3D array as `Array3<float>`, for example.
- Functions for setting the array scalar type and dimensions.
- `testzfp` for regression testing.

Changed

- 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.
- Consolidated several header files.

23.14 0.2.1 (2014-12-12)

Added

- Win64 support via Microsoft Visual Studio compiler.
- Documentation of the expected output for the diffusion example.

Changed

- Made several minor changes to suppress compiler warnings.

Fixed

- Broken support for IBM's `xlc` compiler.

23.15 0.2.0 (2014-12-02)

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.

Added

- 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.

Fixed

- 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.

23.16 0.1.0 (2014-11-12)

Initial beta release.

PYTHON MODULE INDEX

INDEX