

# Lossy data compression

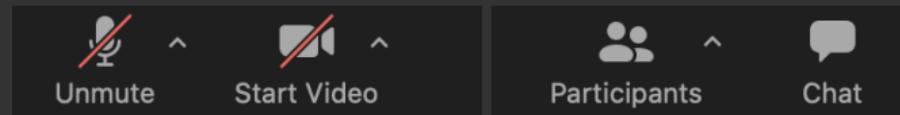
ALEX RAZOUMOV  
alex.razoumov@westdri.ca



Digital Research  
Alliance of Canada

# Zoom controls

- Please mute your microphone and camera unless you have a question
- To ask questions at any time, type in Chat, or Unmute to ask via audio
  - please address chat questions to "Everyone" (not direct chat!)
- Raise your hand in Participants



- Email [training@westdri.ca](mailto:training@westdri.ca)
- Our winter/spring training schedule <https://bit.ly/wg2024a>
  - webinars, courses, summer school at SFU on June 3-7

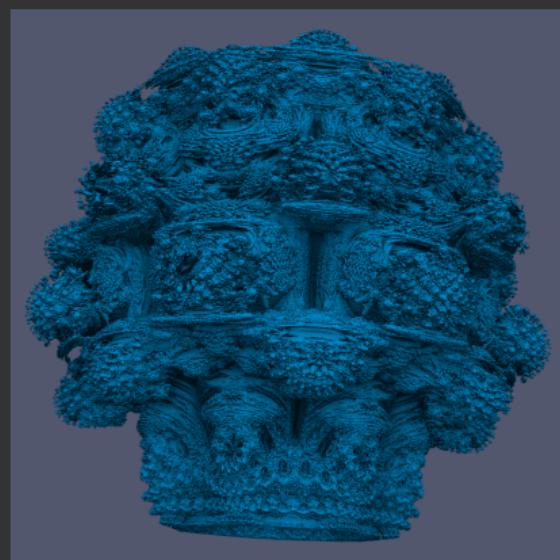
# Numerical simulations produce too much data!

(challenge to write and store long-term)

- Store every 100th timestep
- Store only selected variables
- In-situ visualization
- Data compression

## Compression techniques: lossless compression of 3D data

- Tools like gzip, bzip2, xz and many others
  - Use general-purpose algorithms; replace recurrent bit patterns in the data by references to a single copy of the pattern
  - Often included into the file format (NetCDF, HDF5, VTK)
  - For typical datasets (such as CFD)  $\sim 40 - 50\%$  reduction in size
  - Very high compression rates when a high redundancy is present in the data
    - e.g. the Mandelbulb on the right:  $800^3$  in single precision  $\Rightarrow$  1.9G, actual NetCDF file 12M
    - effectively, a litmus test for the efficiency of your data storage (if stored uncompressed)



## Compression techniques: lossy compression of 3D data

- Lower resolution and/or precision
  - ✓ Orthogonal transformation-based algorithms, e.g. zfp (based on local block transforms, similar to discrete cosine transforms)
  - ✓ Topological compression
  - Compression via ML

- 1D / 2D / 3D scalar fields

- Can this be applied to other data, e.g. MD trajectories?

## **zfp: compressed floating-point and integer arrays**

<https://computing.llnl.gov/projects/zfp>

- Developed at LLNL, supported by the U.S. DOE's Exascale Computing Project
  - Open source <https://github.com/LLNL/zfp>
  - Lossless and lossy
  - Very good documentation <https://zfp.readthedocs.io>
  - Written in C/C++
  - Bindings in C, C++, Python, Fortran; Python interface is called zfPy
  - Built into ~63 projects  
<https://computing.llnl.gov/projects/floating-point-compression/related-projects>
  - For faster (de)compression supports several backends: OpenMP, CUDA, HIP (C++ runtime for AMD and NVIDIA GPUs), FPGAs
  - Reported throughputs up to 2 GB/s per CPU core, 800 GB/s aggregate throughput on GPUs

## Algorithm

- Details at <https://zfp.readthedocs.io/en/releasel.0.1/algorithm.html#lossy-compression>
  - Relies on orthogonal transforms, keeping only significant transform coefficients
  - The transform coefficients are encoded into a losslessly-compressed bit stream that can be truncated in one of three ways, giving a user three “non-expert” compression modes:
    - **fixed-rate mode** controlled by a double-precision parameter (*rate*) giving a fixed number of bits per floating number
    - **fixed-precision mode** controlled by an integer parameter (*precision*) specifying the number of bit planes for the transform coefficients
    - **fixed-accuracy mode** controlled by a double-precision parameter (*tolerance*) specifying the max point-wise variable error

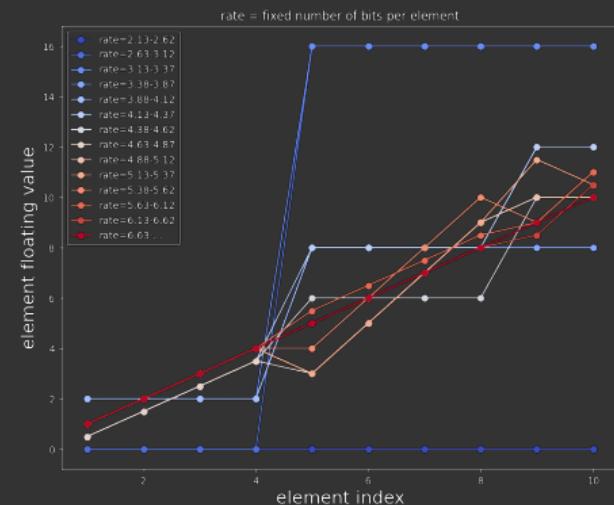
# Using zfp via Python: tiny 1D array

```
import numpy as np, zfpy

# first, lossless compression
initial = np.arange(1, 11, dtype=np.float32)
compressed = zfpy.compress_numpy(initial)
decompressed = zfpy.decompress_numpy(compressed)
np.testing.assert_array_equal(initial, decompressed)
    # they are equal

# next, lossy compression
rate = 4    # number of bits per each floating number
initial = np.arange(1, 11, dtype=np.float32)
compressed = zfpy.compress_numpy(initial, rate=rate)
decompressed = zfpy.decompress_numpy(compressed)
print("rate=", rate, "-->", str(len(compressed))+"B", "decompressed=", decompressed)

file = open("compressed.zfp", "wb")
file.write(compressed)
file.close()
```



Note: the compressed file also includes some overhead beyond individual elements, the relative size of which will decrease for bigger data (next slide)

# Using zfp via Python: large 3D array

- 3D sine envelope wave function defined inside a unit cube ( $x_i \in [0, 1]$ )

$$f(x_1, x_2, x_3) = \sum_{i=1}^2 \left[ \frac{\sin^2 \left( \sqrt{\xi_{i+1}^2 + \xi_i^2} \right) - 0.5}{\left[ 0.001(\xi_{i+1}^2 + \xi_i^2) + 1 \right]^2} + 0.5 \right], \text{ where } \xi_i \equiv 15(x_i - 0.5)$$

- Discretized at  $300^3$  Cartesian grid in double precision `double300.nc`  $\Rightarrow$  206M
- Lossless compression: `double300.nc.gz` - 125M, `double300.nc.bz2` - 119M
- NetCDF's `compression='zlib'` 159M
- NetCDF's `compression='zlib'`, `significant_digits=3` (very aggressive compression) 33M

```
import netCDF4 as nc, zfp

f = nc.Dataset('double300.nc')
rho = f.variables['density'][:]
compressed = zfp.compress_numpy(rho, rate=2.0)
print(len(compressed)) # in bytes => 6.4M

file = open("compressed.zfp", "wb")
file.write(compressed)
file.close()
```

- Demo `rate=2.0, rate=1.0, rate=0.5`

```
import netCDF4 as nc, zfp

file = open("compressed.zfp", "rb")
compressed = file.read()
file.close()
decompressed = zfp.decompress_numpy(compressed)

f = nc.Dataset('compressed.nc', 'w', format='NETCDF4')
nx, ny, nz = decompressed.shape
f.createDimension('x', nx)
f.createDimension('y', ny)
f.createDimension('z', nz)
rho = f.createVariable('density', 'f4', ('x','y','z'))
rho[:, :, :] = decompressed
f.close()
```

Other notable storage formats supporting zfp

- <https://h5z-zfp.readthedocs.io>
  - VTK-m supports zfp for (de)compressing arrays
  - ZfpCompression.jl provides Julia bindings for zfp
  - TTK without topological compression (will demo next)

# Using zfp in Julia

- ZfpCompression.jl is a wrapper around the C zfp library
- Use one of tol::Real, precision::Int, and rate::Real
- Optionally pass nthreads=... argument for multithreading, if ZfpCompression compiled with OpenMP (not enabled on MacOS by default)

```
using ZfpCompression
initial = rand(Float32, 100, 100);

# (1) lossless compression
compressed = zfp_compress(initial);
print(sizeof(initial), "↔", sizeof(compressed)) # 40000 and 34040 bytes
decompressed = zfp_decompress(compressed);
initial == decompressed # true

# (2) lossy compression
compressed = zfp_compress(initial, tol=1e-3);
decompressed = zfp_decompress(compressed);
maximum(abs.(initial - decompressed)) # 0.0003338

print(sizeof(initial), "↔", sizeof(compressed))
# 40000 and 17512 bytes
typeof(compressed) # Vector{UInt8}

write("111.bin", initial) # 40000 bytes
write("111.zfp", compressed) # 17512 bytes
```

```
initial = Array{Float32,2}(undef, 100, 100);
read!("111.bin", initial);
compressed = Array{UInt8,1}(undef, 17512);
read!("111.zfp", compressed);

decompressed = zfp_decompress(compressed);
maximum(abs.(initial - decompressed)) # 0.0003338
```

Using zfp via TTK's TCWriter without topological compression

## In ParaView GUI

## TopologicalCompressionWriter

1. Enable TTK plugin (more on it later in the talk)
  2. File | Save Data, for file type select TTK Compressed Image Data (\*.ttk)
  3. Check ZFP compressor only, set ZFP Relative Error Tolerance

Using zfp via TTK's TCWriter without topological compression

In pypython

```
from paraview.simple import *
data = NetCDFReader(FileNames='double300.nc')
SaveData('double300.ttk', proxy=data, ScalarField=['POINTS', 'density'],
         ZFPRelativeErrorToleranceextra=50,
         UseZFPcompressoronlynottopologicalcompression=1)
```

- To enable parallel mode, rebuild TTK with “-DTTK\_ENABLE\_OPENMP=ON”

| ZFPRelativeToleranceextra | 50% (default) | 20%         | 10%         | 5%                              | 3%        |
|---------------------------|---------------|-------------|-------------|---------------------------------|-----------|
| File size                 | 1.6M          | 2.0M        | 2.5M        | 3.1M                            | 4.4M      |
| Compression ratio         | 129           | 103         | 82          | 66                              | 47        |
| Quality                   | noisy         | quite noisy | small noise | very small noise,<br>acceptable | excellent |

# Deep water impact dataset

IEEE Vis 2018 contest <https://sciviscontest2018.org>

- One timestep, all variables, single-precision compressed VTK: 1.3G
- As far as I can tell, TTK's TopologicalCompressionWriter requires double precision input  $\Rightarrow$  stored sound speed as a double-precision VTK file (`snd.vti`)

| uncompressed (500 <sup>3</sup> ) | gzip-compressed | LZMA-compressed | ZLib-compressed |
|----------------------------------|-----------------|-----------------|-----------------|
| 954M                             | 245M            | 178M            | 258M            |

```
from paraview.simple import *
data = XMLImageDataReader(FileName='snd.vti')
SaveData('zfp50.ttk', proxy=data, ScalarField=['POINTS', 'sound'],
          ZFPRelativeErrorToleranceextra=50,
          UseZFPcompressoronlynotopologicalcompression=1)
```

| ZFPRelativeErrorToleranceextra | 90%  | 50% (default) | 30%  | 20%  | 10%  | 5%  |
|--------------------------------|------|---------------|------|------|------|-----|
| File size                      | 1.2M | 2.0M          | 3.3M | 5.3M | 8.1M | 12M |
| Compression ratio              | 795  | 477           | 289  | 180  | 117  | 80  |

- Telltale sign of zealous zfp compression: negative sound speeds (check `zfp50.ttk` and `zfp10.ttk`)

## Topology-based analysis

This slide is based on Attila Gyulassy's introductory topology tutorial (U. of Utah)

- Phenomenon of interest in a scalar field in the data  $\Rightarrow$  derive its measurable topological equivalent or abstraction
  - Measurable topological attributes:
    - measurement of connectedness: pick any two points + find an inside path to connect them
    - count non-contractable cycles: pick a closed loop inside that you can't squeeze down to a point
    - and many others
  - Applications in:
    - molecular analysis: e.g. pick up atomic bonds, atoms from the 3D electronic probability density
    - materials analysis: porosity measurement, battery design, defect identification, cavity deformation
    - neural pathways (connectomics)
    - CFD: turbulence and vorticity, bubble formation rate, ocean eddies
    - combustion analysis: ignition kernels
    - geological features
    - image segmentation

# Persistence intervals of a 1D scalar function

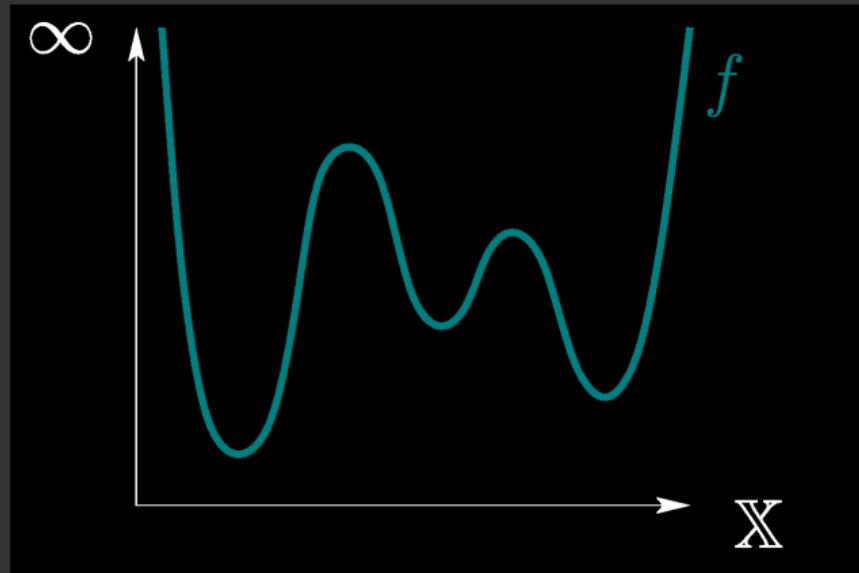


Figure copied from David Cohen-Steiner's slides on Topological Persistence

- Evolution of the topology of connected components during a filtration from  $-\infty$  to  $+\infty$
- Pair thresholds (critical points) that create components with those that destroy them
- Component persistence = difference in function value between component's birth and death

# Persistence intervals of a 1D scalar function

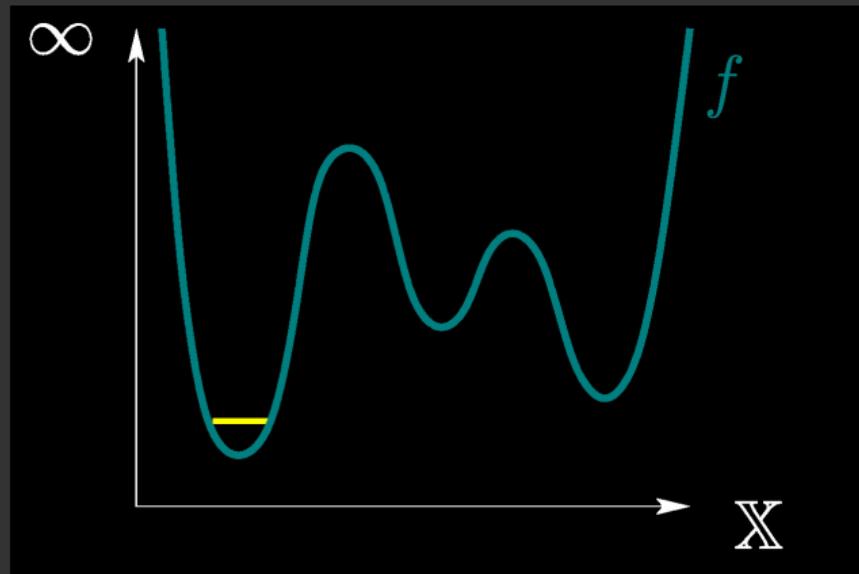


Figure copied from David Cohen-Steiner's slides on Topological Persistence

- Evolution of the topology of connected components during a filtration from  $-\infty$  to  $+\infty$
- Pair thresholds (critical points) that create components with those that destroy them
- Component persistence = difference in function value between component's birth and death

# Persistence intervals of a 1D scalar function

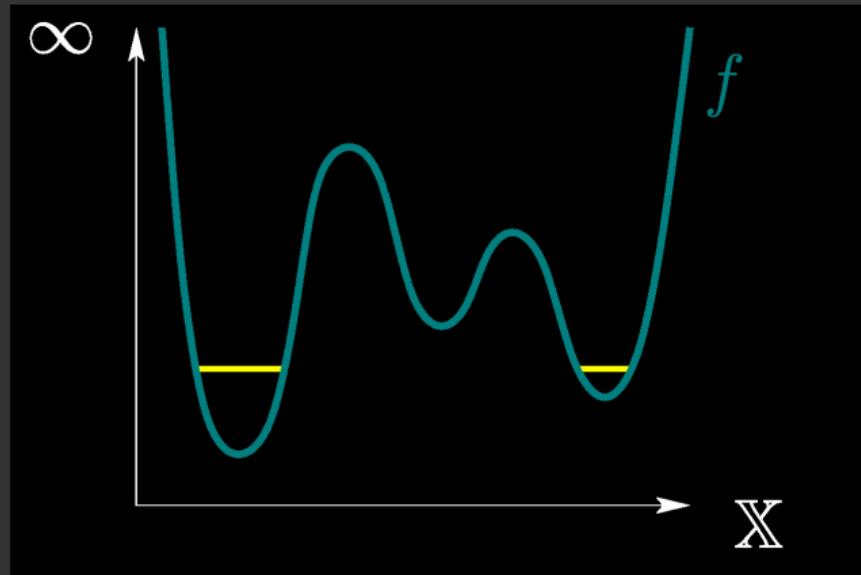


Figure copied from David Cohen-Steiner's slides on Topological Persistence

- Evolution of the topology of connected components during a filtration from  $-\infty$  to  $+\infty$
- Pair thresholds (critical points) that create components with those that destroy them
- Component persistence = difference in function value between component's birth and death

# Persistence intervals of a 1D scalar function

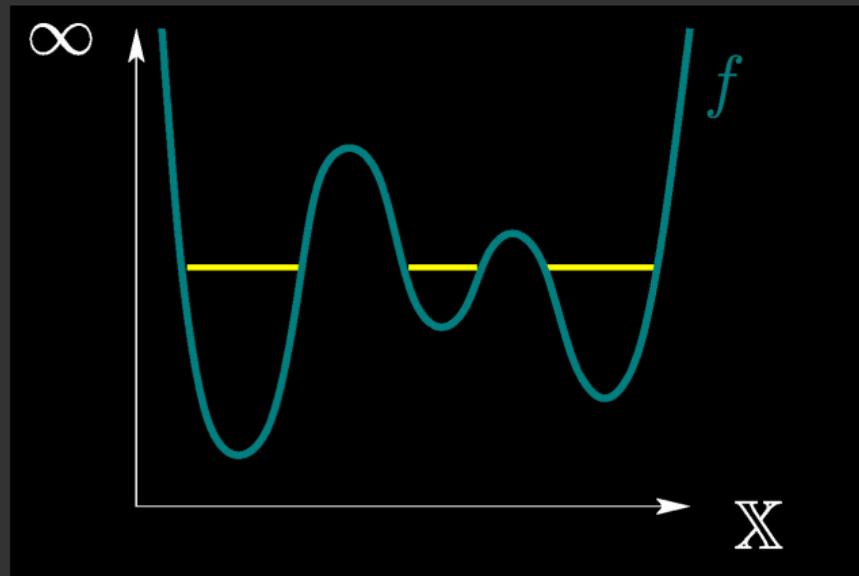


Figure copied from David Cohen-Steiner's slides on Topological Persistence

- Evolution of the topology of connected components during a filtration from  $-\infty$  to  $+\infty$
- Pair thresholds (critical points) that create components with those that destroy them
- Component persistence = difference in function value between component's birth and death

# Persistence intervals of a 1D scalar function

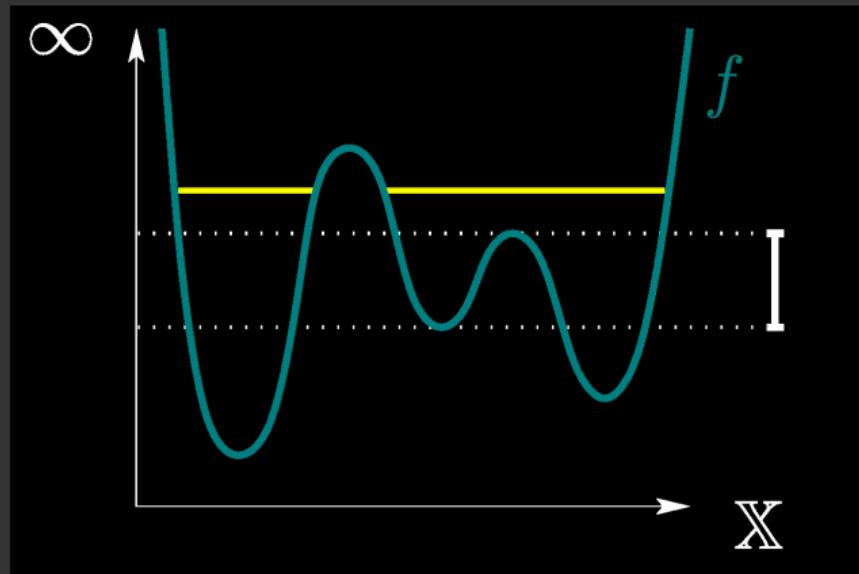


Figure copied from David Cohen-Steiner's slides on Topological Persistence

- Evolution of the topology of connected components during a filtration from  $-\infty$  to  $+\infty$
- Pair thresholds (critical points) that create components with those that destroy them
- Component persistence = difference in function value between component's birth and death

# Persistence intervals of a 1D scalar function

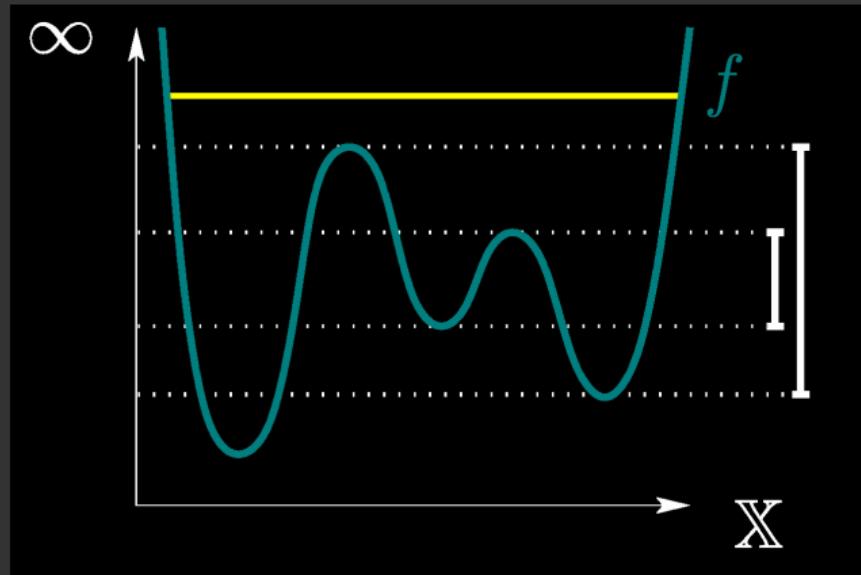


Figure copied from David Cohen-Steiner's slides on Topological Persistence

- Evolution of the topology of connected components during a filtration from  $-\infty$  to  $+\infty$
- Pair thresholds (critical points) that create components with those that destroy them
- Component persistence = difference in function value between component's birth and death

# Persistence intervals of a 1D scalar function

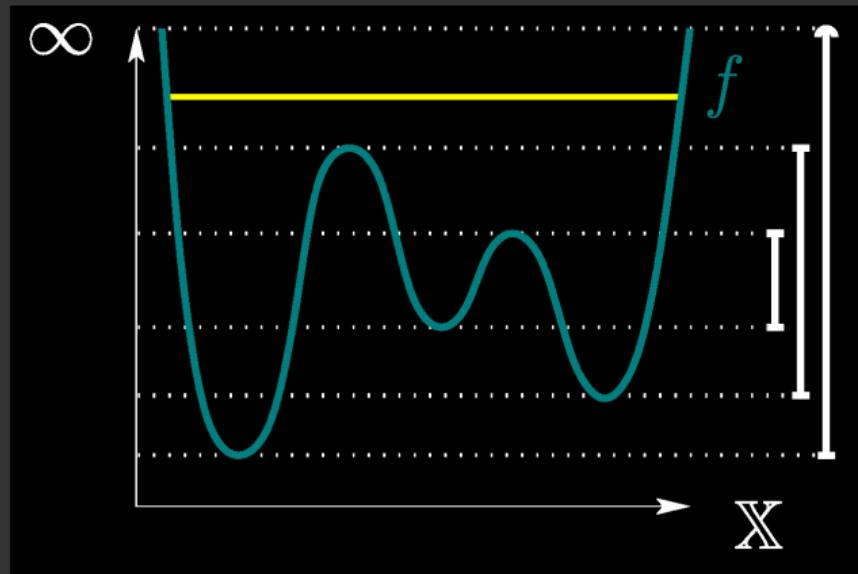


Figure copied from David Cohen-Steiner's slides on Topological Persistence

- Evolution of the topology of connected components during a filtration from  $-\infty$  to  $+\infty$
- Pair thresholds (critical points) that create components with those that destroy them
- Component persistence = difference in function value between component's birth and death

# Persistence diagram

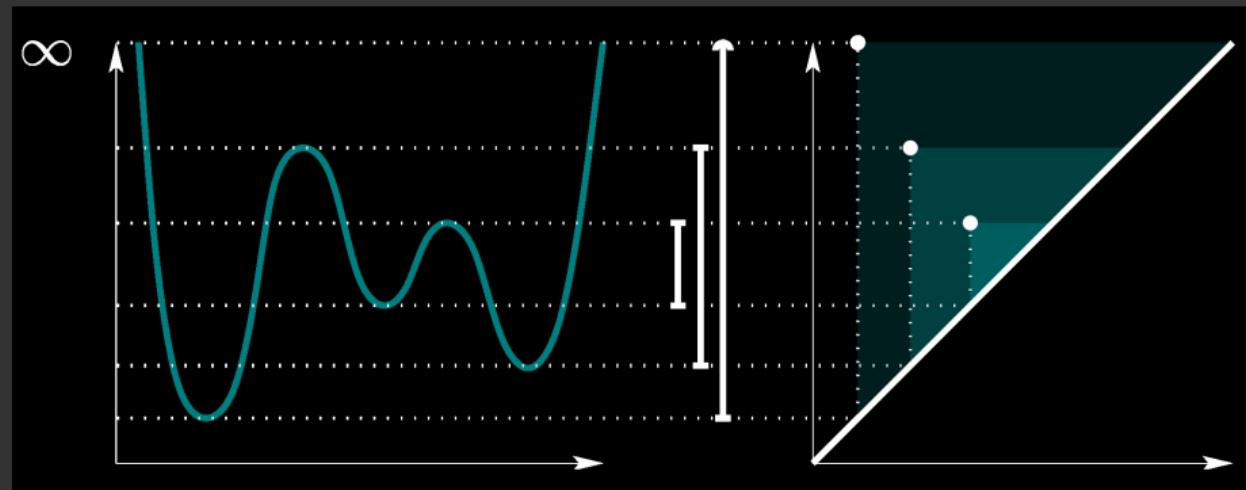


Figure copied from David Cohen-Steiner's slides on Topological Persistence

- Persistence diagram: x = function value at feature's birth, y = function value at feature's death
- Small features are mapped closer to the diagonal
- Critical points in 1D: **minima** and **maxima**
- Domain segmentation (monotone regions) between critical points, based on the gradient sign
- **Mountains** = monotone pieces around maxima, **basins** = monotone pieces around minima

# Scalar function in 2D



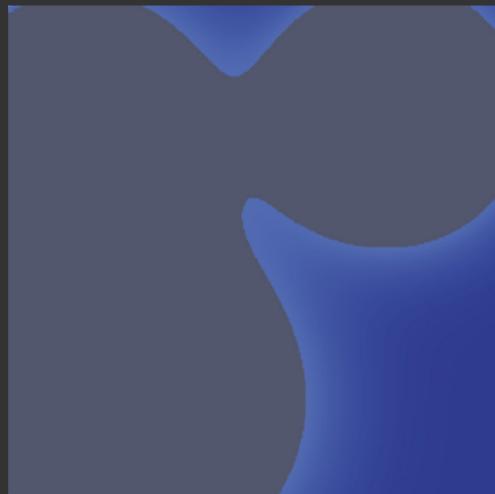
- Sweep in function value from  $-\infty$  to  $+\infty$
- Coloured regions shows the subdomain with function value lower than the sweep value
- This time single component with multiple holes that
  - split at the saddles
  - disappear at the maxima
- Critical points in 2D: **minima**, **maxima**, and **saddle points**
- Domain segmentation (monotone regions) between integral lines (orthogonal to the contours), based on the gradient sign
- Same definition for **mountains** and **basins**, but there are also **ridge lines** and **valley lines**

# Scalar function in 2D



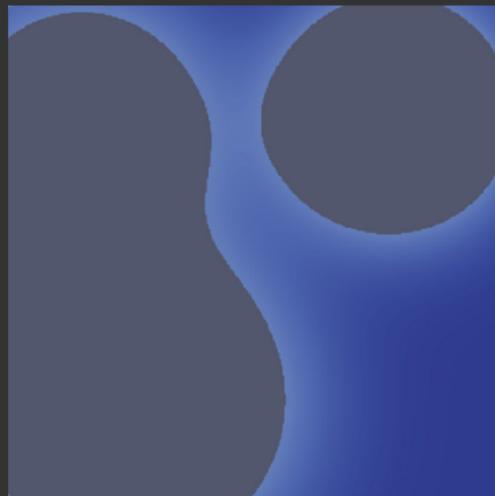
- Sweep in function value from  $-\infty$  to  $+\infty$
- Coloured regions shows the subdomain with function value lower than the sweep value
- This time single component with multiple holes that
  - split at the saddles
  - disappear at the maxima
- Critical points in 2D: **minima**, **maxima**, and **saddle points**
- Domain segmentation (monotone regions) between integral lines (orthogonal to the contours), based on the gradient sign
- Same definition for **mountains** and **basins**, but there are also **ridge lines** and **valley lines**

# Scalar function in 2D



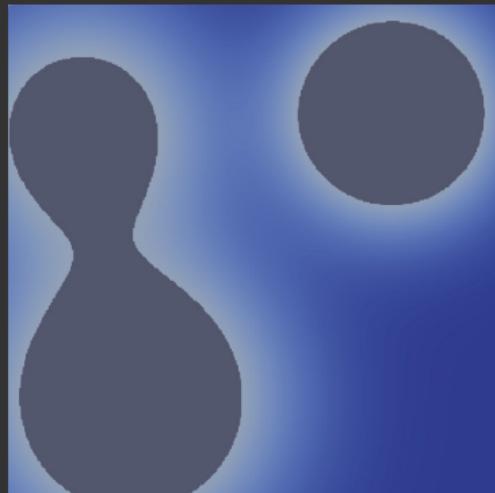
- Sweep in function value from  $-\infty$  to  $+\infty$
- Coloured regions shows the subdomain with function value lower than the sweep value
- This time single component with multiple holes that
  - split at the saddles
  - disappear at the maxima
- Critical points in 2D: **minima**, **maxima**, and **saddle points**
- Domain segmentation (monotone regions) between integral lines (orthogonal to the contours), based on the gradient sign
- Same definition for **mountains** and **basins**, but there are also **ridge lines** and **valley lines**

# Scalar function in 2D



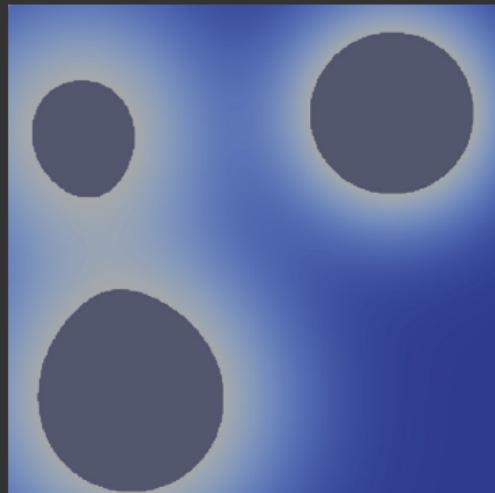
- Sweep in function value from  $-\infty$  to  $+\infty$
- Coloured regions shows the subdomain with function value lower than the sweep value
- This time single component with multiple holes that
  - split at the saddles
  - disappear at the maxima
- Critical points in 2D: **minima**, **maxima**, and **saddle points**
- Domain segmentation (monotone regions) between integral lines (orthogonal to the contours), based on the gradient sign
- Same definition for **mountains** and **basins**, but there are also **ridge lines** and **valley lines**

# Scalar function in 2D



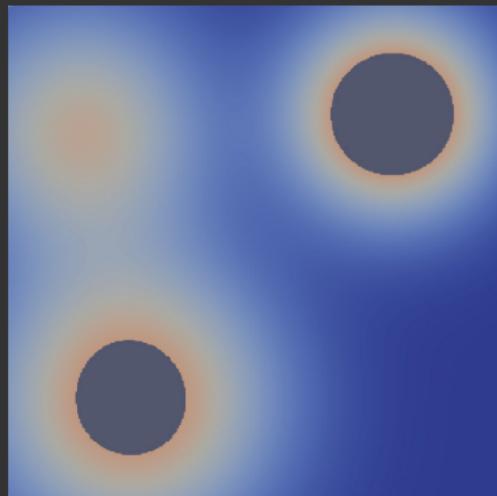
- Sweep in function value from  $-\infty$  to  $+\infty$
- Coloured regions shows the subdomain with function value lower than the sweep value
- This time single component with multiple holes that
  - split at the saddles
  - disappear at the maxima
- Critical points in 2D: **minima**, **maxima**, and **saddle points**
- Domain segmentation (monotone regions) between integral lines (orthogonal to the contours), based on the gradient sign
- Same definition for **mountains** and **basins**, but there are also **ridge lines** and **valley lines**

# Scalar function in 2D



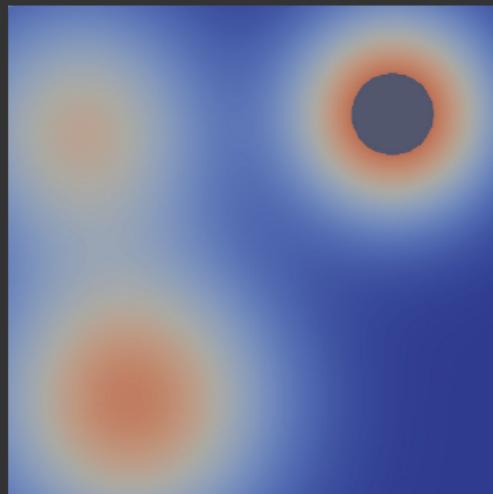
- Sweep in function value from  $-\infty$  to  $+\infty$
- Coloured regions shows the subdomain with function value lower than the sweep value
- This time single component with multiple holes that
  - split at the saddles
  - disappear at the maxima
- Critical points in 2D: **minima**, **maxima**, and **saddle points**
- Domain segmentation (monotone regions) between integral lines (orthogonal to the contours), based on the gradient sign
- Same definition for **mountains** and **basins**, but there are also **ridge lines** and **valley lines**

# Scalar function in 2D



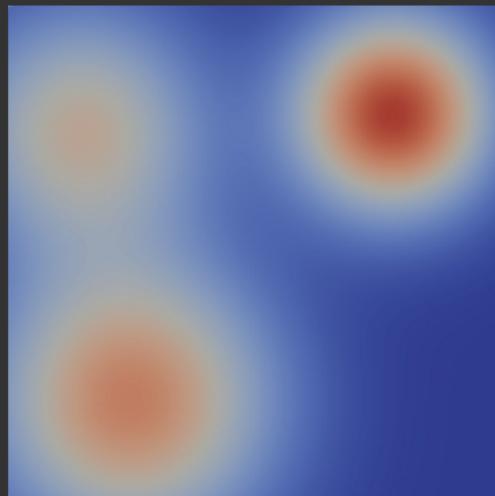
- Sweep in function value from  $-\infty$  to  $+\infty$
- Coloured regions shows the subdomain with function value lower than the sweep value
- This time single component with multiple holes that
  - split at the saddles
  - disappear at the maxima
- Critical points in 2D: **minima**, **maxima**, and **saddle points**
- Domain segmentation (monotone regions) between integral lines (orthogonal to the contours), based on the gradient sign
- Same definition for **mountains** and **basins**, but there are also **ridge lines** and **valley lines**

# Scalar function in 2D



- Sweep in function value from  $-\infty$  to  $+\infty$
- Coloured regions shows the subdomain with function value lower than the sweep value
- This time single component with multiple holes that
  - split at the saddles
  - disappear at the maxima
- Critical points in 2D: **minima**, **maxima**, and **saddle points**
- Domain segmentation (monotone regions) between integral lines (orthogonal to the contours), based on the gradient sign
- Same definition for **mountains** and **basins**, but there are also **ridge lines** and **valley lines**

# Scalar function in 2D



- Sweep in function value from  $-\infty$  to  $+\infty$
- Coloured regions shows the subdomain with function value lower than the sweep value
- This time single component with multiple holes that
  - split at the saddles
  - disappear at the maxima
- Critical points in 2D: **minima**, **maxima**, and **saddle points**
- Domain segmentation (monotone regions) between integral lines (orthogonal to the contours), based on the gradient sign
- Same definition for **mountains** and **basins**, but there are also **ridge lines** and **valley lines**

# Scalar function in 3D

- The topology becomes more complex
- 4 types of critical points: **minima**, **maxima**, and two kinds of **saddles**
  - 1-saddles are located between 2 minima, 2-saddles are located between 2 maxima
- Domain segmentation (monotone regions) between integral surfaces, based on the gradient sign
- Monotone regions combined into **mountains** and **basins**, with **ridge lines** and **valley lines** and **saddle connector lines**, ridge surfaces (separating the mountains) and valley surfaces (separating the basins)

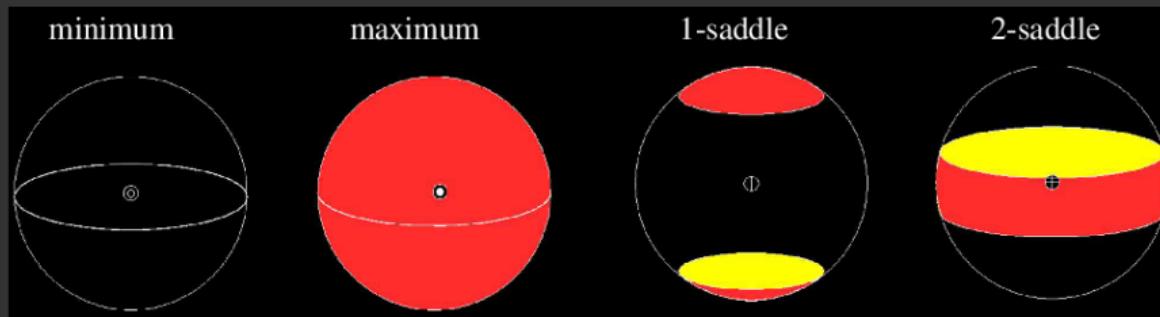


Figure copied from Guoning Chen's slides on Morse-Smale Complex

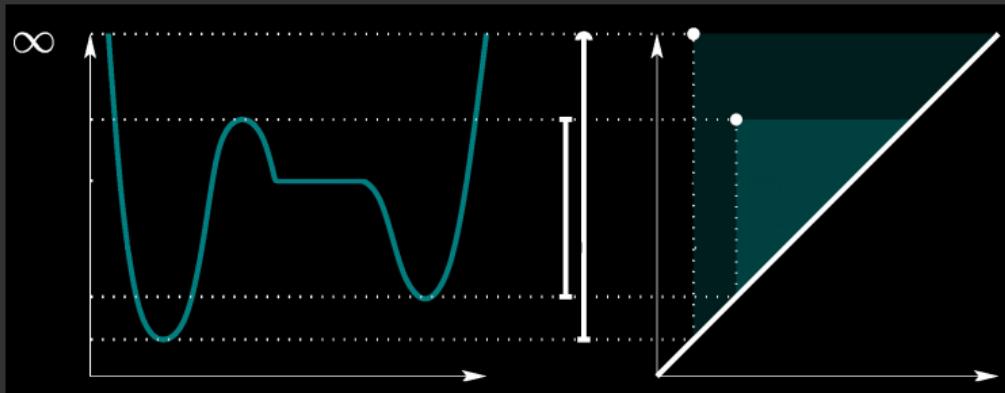
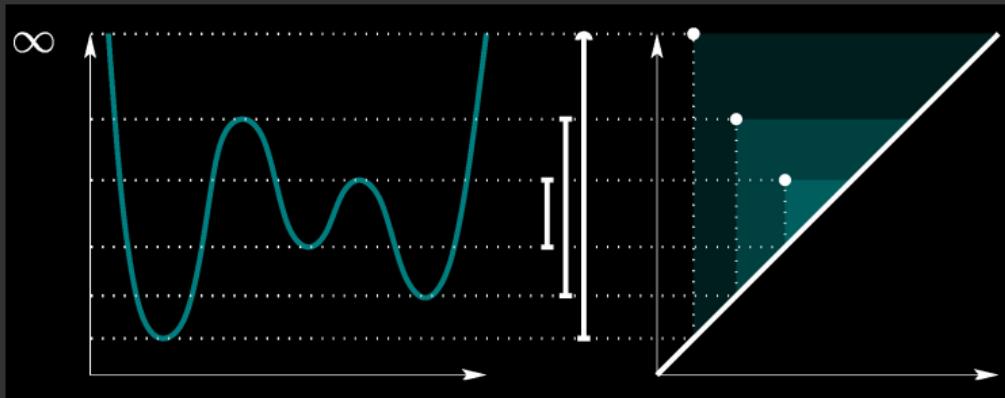
# TTK = the Topology ToolKit

- Topological analysis of multi-dimensional scalar functions
- <https://topology-tool-kit.github.io>
  - great documentation, many tutorials and step-by-step videos
- Open source, development since 2014, first public release in 2017
- Lead author Julien Tierny + active development community
- Some workflows in this presentation were taken from the excellent half-day TTK tutorial at IEEE VIS 2020
- Interfaces:
  - pure C++
  - VTK/C++ (~3X shorter code)
  - ParaView Python (~5X shorter code) and ParaView plugin, and ParaView GUI

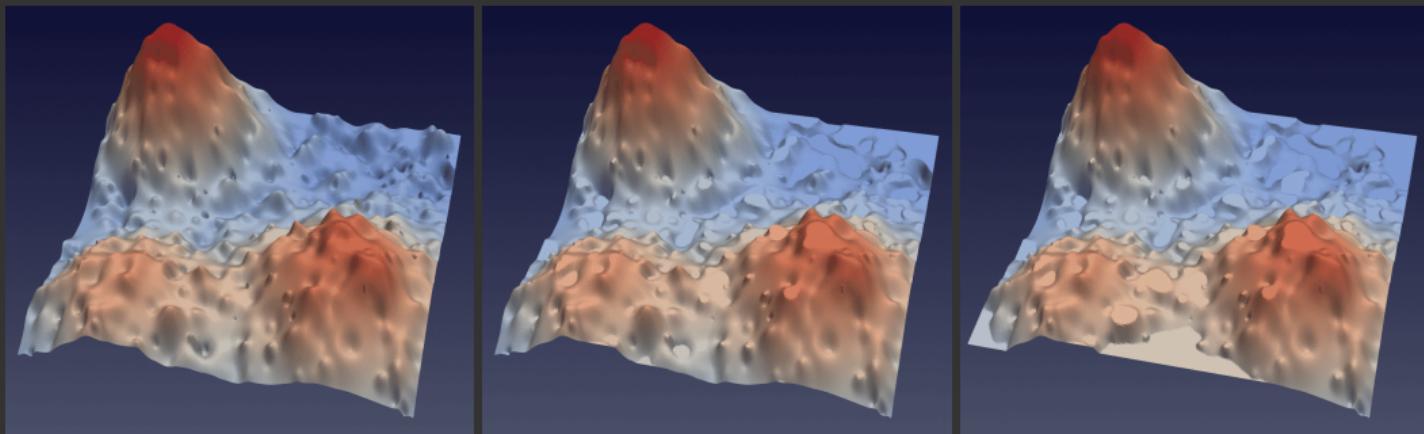
# Topological compression algorithm

1. Topological simplification
  - remove all topological features below a user persistence tolerance  $\epsilon$
2. Domain quantization
  - break the domain into regions by the number of components
  - initially, in each region the data are represented by a constant plane (that could span multiple components)
3. Approximate data in each domain via iterative refinement
  - each plane's vertical value is set to the data average over that region
  - each plane's horizontal extent is set to the data extent over that region
  - compute the intersections of these planes with the data surface
  - keep the original critical points in non-simplified regions
  - stepwise interpolation to connect critical points and the intersections
  - compute the max pointwise error on the grid, if larger than the specified error  $\Rightarrow$  double the number of planes in that region only and repeat the procedure
4. Lossless compression of all elements
5. Optionally combine with zfp compression

# Topological simplification in 1D



# Topological simplification in 2D



On presenter's laptop:

```
cd ~/tmp/lossy  
open s???.png
```

# Topological compression in 1D and 2D

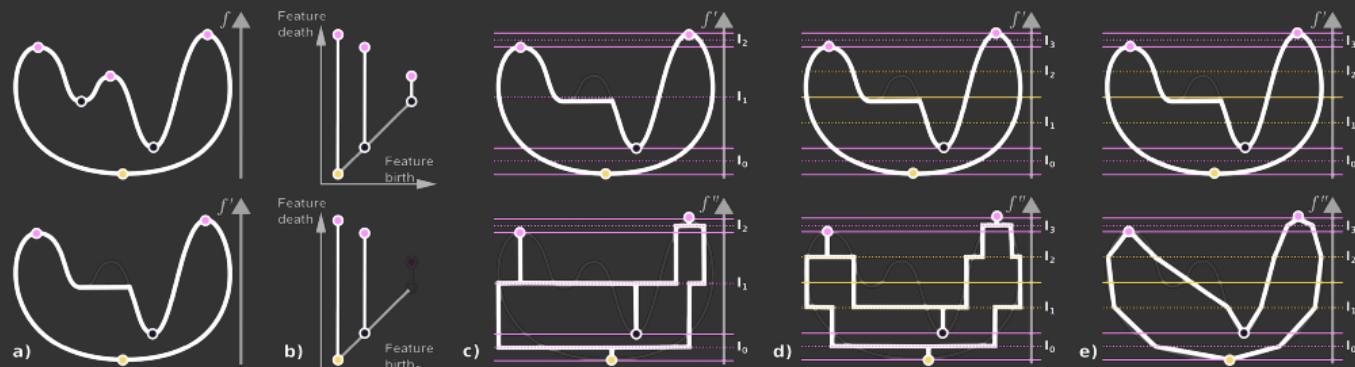


Figure copied from Soler et al. 2018

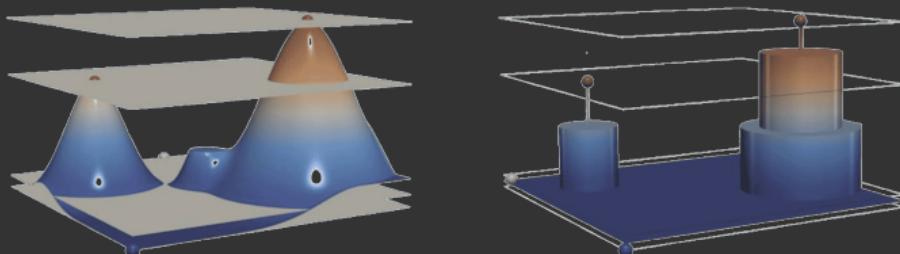


Figure copied from Soler et al., IEEE PacificVis, 2018

# Using TTK's topological compression via ParaView GUI

1. Enable TTK plugin
2. Load `double300.nc`
3. File | Save Data, for file type select TTK Compressed Image Data (\*.ttk)
4. Select **topological persistence tolerance** and **max pointwise error**
5. Optionally select ZFP Relative Error Tolerance

## Using TTK's topological compression via pypython

```
from paraview.simple import *
data = NetCDFReader(FileNames='double300.nc')
# compress & save to TTK Topological Compression format
SaveData("double300.ttk", proxy=data, ScalarField=["POINTS", "density"],
          Topologicallosspersistencepercentage=10,
          Maximumpointwiseerrorpercentage=10,
          ZFPRelativeErrorToleranceextra=50)
```

# Using TTK's topological compression via C++/VTK

Step 1: install TTK (could not use x86-compiled MacOS binary ⇒ compile)

```
wget https://github.com/topology-tool-kit/ttk/archive/1.2.0.tar.gz
tar xvfz 1.2.0.tar.gz && cd ttk-1.2.0
brew install vtk libomp llvm cmake
mkdir ttk_install build && cd build
FLAGS=(
  -DTTK_BUILD_PARAVIEW_PLUGINS=OFF
  -DCMAKE_INSTALL_PREFIX=$HOME/tmp/ttk
)
export OpenMP_ROOT=$(brew --prefix)/opt/libomp
cmake .. "${FLAGS[@]}"
make -j4
make install
```

# Using TTK's topological compression via C++/VTK (cont.)

## Step 2: write the code and compile it against TTK

```
#include <CommandLineParser.h>
#include <vtkSmartPointer.h>
#include <vtkNew.h>
#include <vtkImageData.h>
#include <vtkPointData.h>
#include <vtkXMLGenericDataObjectReader.h>
#include <ttkTopologicalCompressionWriter.h>
#include <string>

int main(int argc, char **argv) {
    ttk::CommandLineParser parser;
    std::string inputFile, outputFile, tolerance, error;
    parser.setArgument("i", &inputFile, "Path to input VTI file");
    parser.setArgument("o", &outputFile, "Path to output TTK file");
    parser.setArgument("t", &tolerance,
        "Topological persistence tolerance percentage", true);
    parser.setArgument("e", &error, "Maximum error", true);
    parser.parse(argc, argv);

    // read the data
    auto reader = vtkSmartPointer<vtkXMLGenericDataObjectReader>::New();
    reader->SetFileName(inputFile.data());
    reader->Update();
    auto inputDataObject = reader->GetOutput();
    if(!inputDataObject) {
        cout << "Unable to read input file " + inputFile << endl;
        return 1;
    }

    auto inputAsVtkDataSet = vtkDataSet::SafeDownCast(inputDataObject);
    auto pointData = inputAsVtkDataSet->GetPointData();
    cout << "Read '" << pointData->GetArrayName(0) << "' array" << endl;

    vtkNew<ttkTopologicalCompressionWriter> topoWriter{};
    topoWriter->SetInputArrayToProcess(0, 0, 0,
        vtkDataObject::FIELD_ASSOCIATION_POINTS, pointData->GetArrayName(0));
    topoWriter->SetInputData(inputAsVtkDataSet);

    // set parameters
    if (tolerance.length() > 0) // persistence %; default 10
        topoWriter->SetTolerance(std::stod(tolerance));
    if (error.length() > 0) // relative error; default 10
        topoWriter->SetMaximumError(std::stod(error));
    topoWriter->SetZFPThreshold(-1); // no zfp compression
    topoWriter->SetSubdivide(true);
    topoWriter->SetUseTopologicalSimplification(true);

    cout << "Topological persistence tolerance percentage = "
        << topoWriter->GetTolerance() << endl;
    cout << "Maximum error = " << topoWriter->GetMaximumError() << endl;

    // write compressed TTK file
    topoWriter->SetFileName(outputFile.c_str());
    topoWriter->Write();

    return 0;
}
```

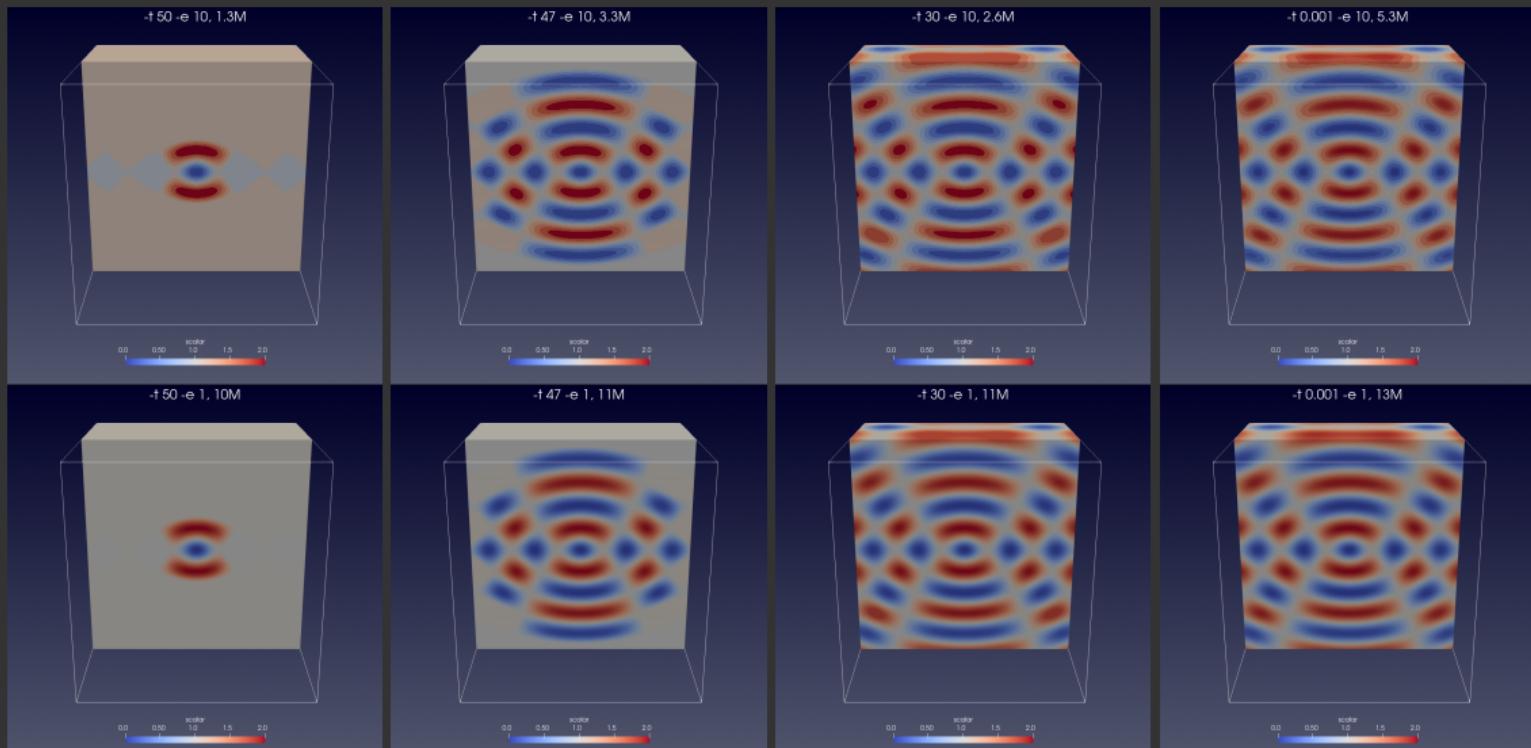
## Using TTK's topological compression via C++/VTK (cont.)

Step 3: run the code

```
(cd ~/tmp/lossy && make)
./main -i double300.vti -t 10 -e 10 -o double300.ttk
```

Two parameters: topological persistence tolerance and max error

- Both as percentages (defaults: 10% and 10%)
  - Original dataset: 206M, rendering similar to the lower right panel



## Deep water impact dataset

IEEE Vis 2018 contest <https://sciviscontest2018.org>

- Recall: double-precision, single-variable VTK file (`rnd.vti`)

| uncompressed ( $500^3$ ) | gzip-compressed | LZMA-compressed | ZLib-compressed |
|--------------------------|-----------------|-----------------|-----------------|
| 954M                     | 245M            | 178M            | 258M            |

- ## • Demoing with C++/VTK:

```
cd ~/tmp/deepWaterImpact  
./lossy/main -i snd.vti -t 10 -e 10 -o t10e10.ttk
```

| Arguments         | <code>-t 10 -e 10</code> (default)     | <code>-t 5 -e 10</code>                 | <code>-t 10 -e 5</code>                 | <code>-t 20 -e 10</code> | <code>-t 30 -e 10</code>                 |
|-------------------|--|---|---|--------------------------|--|
| File name         | t10e10.ttk                             | t5e10.ttk                               | t10e5.ttk                               | t20e10.ttk               | t30e10.ttk                               |
| File size         | 9.6M                                   | 16M                                     | 9.8M                                    | 4.2M                     | 1.7M                                     |
| Compression ratio | 99                                     | 60                                      | 97                                      | 227                      | 561                                      |
| Quality           | looks great, colour a little quantized | looks identical, so t10 is already good | looks identical, so e10 is already good | somewhat compressed      | quantized colours, definitely compressed |

- ~1 min to compress and ~1 min to uncompress (or load into ParaView)
  - Compare two 3D distributions (`snd.vti` vs. `t10e10.ttk`): `paraview --script=compare.py`
  - Variable bounds kept constant

- Both methods (zfp and TC) are quite comparable in output file quality and sizes
  - Both can *reliably achieve* ~20-30X compression, without any visible data degradation – and sometimes as high as ~80-200X
  - Compression artifacts show in very different ways
  - Can combine the two for improved quality and compression (they operate on different bits), see some comparison at <https://topology-tool-kit.github.io/examples/persistenceDrivenCompression>
  - TC is much slower, both during compression and decompression
    - however, it provides more control (two parameters vs. one)
    - it preserves variable bounds
    - in certain cases it can achieve much higher compression, esp. if interested only in certain components
  - Both algorithms act on scalar arrays, not files
  - zfp: C/C++, Fortran, Python, built into many third-party packages, can compress integer, single- and double-precision datasets
  - TC: only one widely accessible implementation (TTK), available via C++, ParaView (GUI and scripting), double precision only?
  - In principle, TC should be able to compress AMR (multi-resolution) data in one go, as it does not rely on spatial wavelengths, although this would require some effort

# Questions?