CT Simulation environment

(User Manual and Code Explication)

CTS Version 1.0.0.1-BETA

UNI FR

Contents

Chapter 1: Introduction

CTS (CT Simulation) is a modular, open-source environment for simulating and segmenting X-ray CT volume data. It is intended as a free alternative to commercial CT slice viewers and segmentation tools, targeting academic researchers and students in materials science and petrophysics. In line with platforms like 3D Slicer (a free open-source medical image toolkit) (Fedorov et al., 2012)[1], CTS provides interactive volume visualization and annotation without licensing costs. Key design goals include modularity (allowing plugin modules and scripts) and extensibility via modern AI methods. For example, CTS integrates state-of-the-art segmentation models: the Segment Anything Model 2 (SAM2) for promptable image/video segmentation (Ravi et al., 2024)[2], its microscopy-focused extension µSAM (Archit et al., 2025)[3], and the Grounding DINO object detector (Liu et al., 2023)[4] that uses text prompts for open-vocabulary segmentation. These AI modules enable advanced automated labeling within the CTS environment. The software is being developed at the University of Fribourg (Switzerland) using a .NET technology stack (SharpDX[5]/DirectX for graphics, ILGPU[6] for GPU compute, ONNX Runtime[7] for neural network inference, Math.NET[8] for math, and OpenTK[9]). CTS is currently in test mode and continuously evolving with new features and models. CTS provides an integrated environment for simulations on CT scanned datasets. Key points of the software are:

- **Open Source, Research Focus:** CTS is released under an open-source license and is freely available. It is aimed at educational and research use, mirroring other free imaging platforms.

[1] Fedorov, A., Beichel, R., Kalpathy-Cramer, J., Finet, J., Fillion-Robin, J.-C., Pujol, S., Bauer, C., Jennings, D., Fennessy, F., Sonka, M., Buatti, J., Aylward, S., Miller, J. V., Pieper, S., & Kikinis, R. (2012). 3D Slicer as an image computing platform for the Quantitative Imaging Network. *Magnetic Resonance Imaging*, 30(9), 1323–1341.
[2] Ravi, N., Gabeur, V., Hu, Y.-T., Hu, R., Ryali, C., Ma, T., Khedr, H., Rädle, R., Rolland, C., Gustafson, L., Mintun, E., Pan, J., Alwala, K. V., Carion, N., Wu, C.-Y., Girshick, R., Dollár, P., & Feichtenhofer, C. (2024). *SAM 2: Segment Anything in Images and Videos*. arXiv:2408.00714.
[3] Archit, A., Freckmann, L., Nair, S., Khalid, N., Hilt, P., Rajashekar, V., Freitag, M., Teuber, C., Buckley, G., von Haaren, S., Gupta, S., Dengel, A., Ahmed, S., & Pape, C. (2025). *Segment Anything for Microscopy*. Nature Methods, 22, 579–591.
[4] Liu, S., Li, F., Zhang, L., et al. (2023). Grounding DINO: Marrying DINO with grounded pre-training for open-set object detection. arXiv:2303.05499.
[5] SharpDX Contributors. (2019). SharpDX (Version 4.2) [Software]. https://github.com/sharpdx/SharpDX
[6] Lüthi, M. (2017). ILGPU: A high-performance GPU compiler for .NET (Version 0.9) [Software]. https://github.com/m4rs-mt/ILGPU
[7] Bai, Y., Feng, T., Li, Q., Rao, V., Seetharaman, G., & Rasheed, R. (2019). ONNX Runtime: Graph optimizations and scalable deep-learning inference [White paper]. Microsoft.
[8] Christensen, C., Mahieu, J., & Math.NET Developers. (2023). Math.NET Numerics (Version 5.0) [Software]. https://numerics.mathdotnet.com
[9] OpenTK Team. (2024). OpenTK: The Open Toolkit Library (Version 4.8) [Software]. https://opentk.net

- **Modular Architecture:** The system is organized as independent modules (e.g., data loaders, segmentation tools) accessible from a central interface, allowing future extensions.
- **Advanced AI Segmentation:** Includes integration of Meta's SAM2 foundation model, the μSAM adaptation for microscopy, and the language-driven Grounding DINO detector, all of which improve segmentation capabilities.
- **Development Platform:** Built at University of Fribourg (CH) on top of the C# programming language, SharpDX (DirectX), ILGPU (GPU), ONNX Runtime, Krypton GUI[10], Math.NET, and OpenTK.
- **Current Status:** CTS is in active development and testing; users should expect occasional updates and experimental features.

[10] Component Factory. (2024). Krypton Toolkit (Version 6.0) [software]. https://github.com/Krypton-Suite/Standard-Toolkit

Chapter 2: Main Interface and Modules

MainForm and ControlForm

The CTS user interface consists of two main windows: the **MainForm** (Figure 1) for data visualization and the **ControlForm** for controls and modules. The MainForm displays the CT volume in four synchronized views: three orthogonal slice views (XY, XZ, YZ planes) and a 3D rendered orthogonal volume view. These are arranged in a 2 × 2 grid: top-left shows the XY (axial) slice, top-right the YZ (coronal) slice, bottom-left the XZ (sagittal) slice, and bottom-right the 3D volume rendering. Each slice view is a scrollable image panel with a colored title label (e.g. "XY View" in yellow). All views support zoom and pan. In each view, the user can navigate through slices via sliders or spinboxes in the ControlForm.

The ControlForm contains menus and tool panels for data operations and segmentation. Key elements include:

- **File Menu:** New (restart), Load Folder (load a folder of images), Import .bin (load a saved CT data file), Save .bin (save volume+labels), Export Images (save all slices as images), and Compress/Decompress Volume for the CTS volume compression.
- **Edit Menu:** Manage materials (Add/Delete/Rename/Merge materials, Extract selection into new material) and threshold operations (Add/Subtract thresholded voxels to the current material). It also provides a "Segment Anything" command that launches the SAM2-based segmentation tool.
- **Tools Menu:** Interactive segmentation modes and utilities. Basic tools are Pan, Eraser, Brush, Measurement, Thresholding, and Lasso (each is toggleable via the menu). Additional options include Brightness/Contrast adjustment and a Node Editor. The Tools menu also has an **Artificial Intelligence** submenu that groups the AI modules:
  - *Segment Anything CT:* Launches the SAM2-based (Ravi et al., 2024) segment-anything tool on the current volume and material.
  - *MicroSAM:* Launches the microscopy-optimized SAM (µSAM) tool (Archit et al., 2025).

- *Grounding DINO:* Launches the DINO-based object detector with grounding, allowing text-prompted segmentation (Liu et al., 2023).
- *Texture Classifier:* A Grey Level Co-Occurrence Matrix (GLCM) machine-learning classifier for material textures (Sebastian V. et al., 2012)[11].

- **Other Modules:** Additional menu items include Integrate/Resample (downsample or resample volumes), Band Detection (artifact/band removal), and Transform Dataset (apply rigid transforms), each opening a specialized dialog.

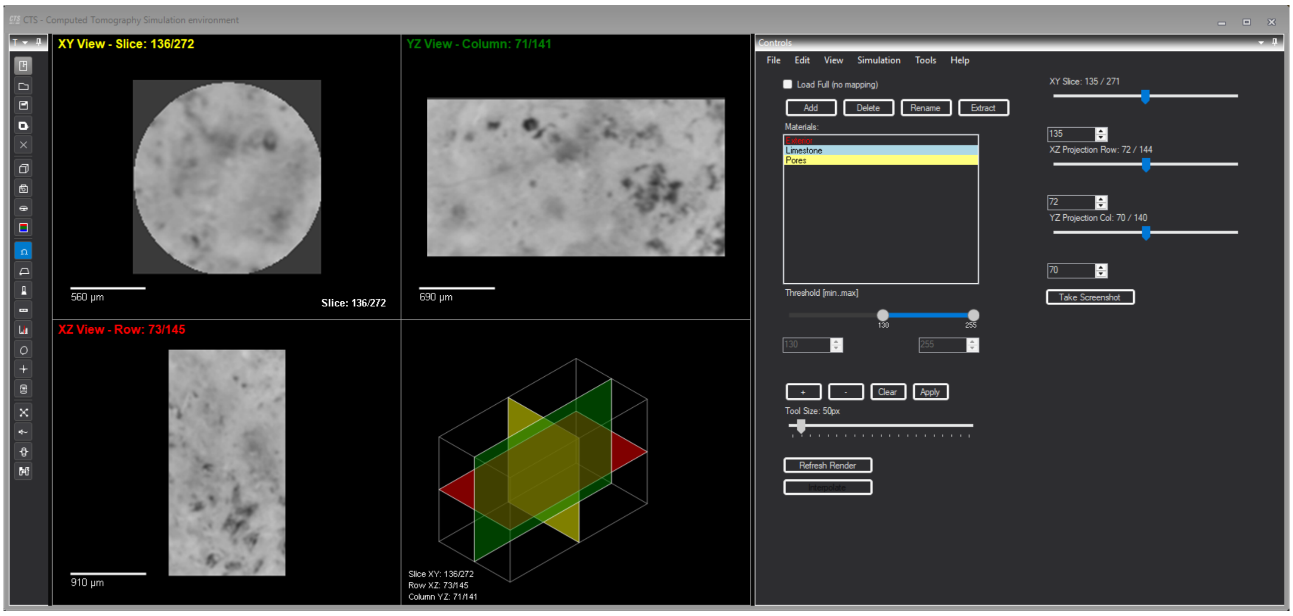


Figure 1. CTS – Main window and control panel

Data Chunking and Compression

CTS uses a chunked 3D volume format. The raw volume is subdivided into cubic chunks of size $256 \times 256 \times 256$ voxels (**constant CHUNK_DIM = 256**). This enables out-of-core loading via memory-mapped files for large datasets. When a folder of images is loaded, CTS creates a single volume.bin containing all chunk data and writes a header file volume.chk with volume metadata. The header is a small binary storing: width, height, depth, chunkDim, and voxel size (in physical units).

[11] Sebastian, V., B., Unnikrishnan, A., & Balakrishnan, K. (2012). Gray Level Co-Occurrence Matrices: Generalisation and Some New Features. CoRR, abs/1205.4831.

For example, in code:

```
bw.Write(volWidth);
bw.Write(volHeight);
bw.Write(volDepth);
bw.Write(chunkDim);
bw.Write(pixelSize);

```

writes the header fields. A corresponding labels.bin file holds the segmentation labels, with its header first specifying the same chunkDim and the counts of chunks in X, Y, Z, followed by blank label data (initialized to zero). For instance, the code does:

```
bw.Write(chunkDim);
bw.Write(cntX); bw.Write(cntY); bw.Write(cntZ);
byte[] emptyChunk = new byte[chunkDim*chunkDim*chunkDim];
for (int i = 0; i < totalChunks; i++)
    bw.Write(emptyChunk, 0, emptyChunk.Length);

```

During editing, assigning voxels to materials writes the material ID into the label volume's chunks.

CTS also provides a custom 3D predictive compression algorithm. This encoder predicts each voxel's intensity from its neighbors in all three dimensions and encodes the residuals; run-length encoding (RLE) can be applied to further compress repeated values. Predictive coding leverages the strong spatial redundancy in CT data, yielding high compression ratios without loss. The user can select compression level and options via the Volume Compression dialog; the program will output a compressed .cts3d file. Decompression restores the original volume exactly when using lossless mode.

Binning (Downsampling)

When loading a dataset, the user may specify an integer $binning\ factor\ > 1$ to downsample the data. If binning is enabled, CTS renames the original volume.bin (making a backup) and then creates a downsampled volume by averaging or maxing blocks of voxels. After binning, CTS replaces volume.bin with the new file and updates the header. The code updates the header by calling CreateVolumeChk with the new dimensions and scales the pixel size by the bin factor. For example:

```
CreateVolumeChk(path, newWidth, newHeight, newDepth, chunkDim, pixelSize * binFactor);

```

This ensures the header correctly reflects the reduced resolution and adjusted physical spacing after binning.

Material Labels

CTS treats each material as a unique byte value. The list of Materials (each with a name, color, and value range) is maintained in memory on the MainForm. The label volume (labels.bin) assigns each voxel an ID corresponding to a material. As noted, the label file's header contains the chunk size and the number of chunks in each dimension. Because label values are bytes, the system supports up to 255 materials (value 0 is reserved for the exterior/background). Saving the project can export a combined file containing volume and label chunks together. When the user merges or deletes materials, CTS updates the label data accordingly and may rewrite labels.bin.

Segmentation Tools

CTS offers several interactive segmentation modes:

- **Brush/Eraser:** In Brush mode, the user paints directly on the current slice (XY/XZ/YZ) to add voxels to the selected material. The Eraser tool removes material labels from voxels. Brush size is adjustable via a slider.
- **Lasso:** Freehand region selection on a slice. The drawn polygon's interior voxels are added to (or erased from) the current material.
- **Thresholding:** The user sets a grayscale intensity range (min and max) using sliders in the ControlForm. CTS identifies all voxels in the displayed slice (or optionally in the entire volume) whose intensities fall within this range. Clicking the "+" button adds these voxels to the current material's label; "–" removes them. These operations run in the background (via MaterialOperations.AddVoxelsByThreshold) to avoid UI freezes.
- **Slice Navigation:** Sliders and numeric controls labeled XY, XZ, YZ in the ControlForm let the user change the current slice index for each view. As the index changes, the corresponding 2D view updates its image and any overlay (mask, brush cursor).
- **Pan/Zoom:** Clicking and dragging in Pan mode lets the user move the slice image. The 2D picture boxes automatically handle zoom (mouse wheel) and scrollbars.
- **Measurement:** When Measurement mode is active, the user can draw a line segment on a slice to measure distance in micrometers, using the known pixelSize.

- **Mask Rendering:** The View menu has an option to toggle rendering of a translucent overlay showing the labeled regions in each slice. This helps visualize segmented volumes.
- **Segment Anything (AI) Module:** Selecting the SAM2 module ("Segment Anything") launches a specialized window where the user can place prompts (points or boxes) on a slice and the AI will suggest a mask for the current material. This utilizes the integrated SAM2/µSAM backend to assist segmentation.

Screenshot and View Capture

CTS supports an enhanced screenshot function to document results. Invoking **Save Screenshot** prompts the user to select a file, then programmatically captures the contents of all four view panels. The code composites the XY, YZ, XZ slice bitmaps and the 3D render into a single image (with appropriate scaling), draws scale bars and labels, and saves the result to PNG or JPEG. This provides a publication-ready figure of the segmented volume.

Main View Layout Summary: The MainForm's grid has:

- *Top-left:* XY (axial) slice view
- *Top-right:* YZ (coronal) slice view
- *Bottom-left:* XZ (sagittal) slice view
- *Bottom-right:* 3D volume rendering (OrthogonalViewPanel)

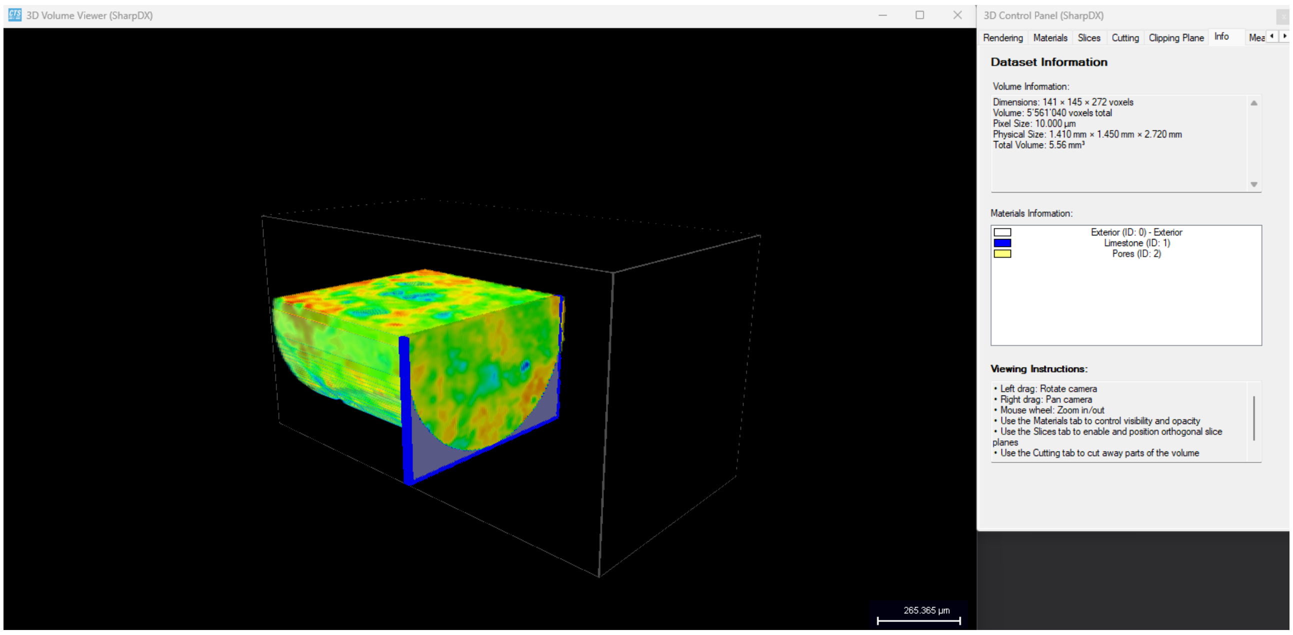
Each 2D view includes a title bar (e.g. "XY View" in yellow) and scrollbars. The 3D view shows the volume as a translucent cube with labeled regions highlighted.

Chapter 3: 3D Viewer and Volume Rendering

3.1 Overview

The CTS 3D Viewer (Figure 2) provides an interactive GPU-accelerated rendering environment (Engel et al., 2006)[12] for visualizing volumetric datasets using DirectX 11 via SharpDX. The viewer allows users to explore CT volumes in real time with options for grayscale volume display, colored label overlays, slice planes, clipping, cutting, and measurement. It includes a control panel that enables fine-grained tuning of rendering settings and export operations.

The 3D Viewer is implemented as a separate window (SharpDXViewerForm) that integrates a rendering panel (SharpDXVolumeRenderer) and a tabbed control interface (SharpDXControlPanel). The rendering backend supports both full-volume rendering and a **streaming mode** designed for extremely large datasets.



Dataset Information

Volume Information:

Dimensions: 141 × 145 × 272 voxels
Volume: 5'561'040 voxels total
Pixel Size: 10.000 µm
Physical Size: 1.410 mm × 1.450 mm × 2.720 mm
Total Volume: 5.56 mm³

Materials Information:

⬜	Exterior (ID: 0) - Exterior
🟦	Limestone (ID: 1)
🟨	Pores (ID: 2)

Viewing Instructions:

- Left drag: Rotate camera
- Right drag: Pan camera
- Mouse wheel: Zoom in/out
- Use the Materials tab to control visibility and opacity
- Use the Slices tab to enable and position orthogonal slice planes
- Use the Cutting tab to cut away parts of the volume

Figure 2 – CTS 3D Viewer and Control Panel

[12] Engel, K., Hadwiger, M., Kniss, J., Rezk-Salama, C., & Weiskopf, D. (2006). Real-Time Volume Graphics. A K Peters/CRC.

3.2 Launch and Initialization

When the 3D Viewer is launched (from the ControlForm or Tools menu), a new SharpDXViewerForm window is created. Internally, it initializes:

- A **rendering panel** (Panel renderPanel) to host the SharpDX surface
- A SharpDXVolumeRenderer instance that sets up GPU shaders, textures, and DirectX pipeline
- A SharpDXControlPanel floating window with controls for rendering, materials, slicing, cutting, and measurement

To support responsiveness, a **render timer** with 33ms ticks (~30 FPS) keeps the visualization updated based on interaction or changes to rendering state.

3.3 Rendering Pipeline

The volume renderer (D3D11VolumeRenderer) constructs a real-time ray-marching pipeline (Levoy, 1988)[13] with:

- **DirectX 11 device and swap chain**
- **3D textures**: grayscale volume, label volume, LOD volumes
- **Pixel shader**: performs volume raycasting, material coloring, threshold clipping, and cutplane handling
- **LOD system**: dynamic adjustment of ray step size for interactive performance (especially for large data)
- **Streaming renderer**: for datasets >8 GB, texture chunks are loaded and rendered on demand to reduce VRAM load
- **Automatic adaptation**: The system will automatically detect the GPU in use and adapt the quality to best fit GPU resources.

A specialized constant buffer (ConstantBufferData) communicates per-frame parameters like slice positions, camera orientation, clipping planes, and material opacities to the shader.

3.4 User Interaction and Controls

[13] Levoy, M. (1988). Display of surfaces from volume data. IEEE Computer Graphics and Applications, 8(3), 29–37.

The 3D Viewer supports the following interactions:

- **Mouse rotate / pan / zoom**: Right-click to rotate, middle-drag to pan, scroll wheel to zoom
- **Cutting planes**: Enable/disable cutting planes for X, Y, or Z with forward/backward directions and position, cutting planes also allow for slice planes
- **Clipping plane**: Enable a planar clip with adjustable normal and distance, slab mode supported
- **Material control**: Per-material visibility and opacity settings with immediate visual feedback
- **Threshold control**: Global intensity min/max thresholds to hide background noise or focus on relevant structures
- **LOD system**: Adaptive ray step size during interaction to ensure smooth performance

All options are exposed in the D3D11ControlPanel through tabbed controls (Rendering, Materials, Slices, Cutting, Clipping Plane, Measurements, Info).

3.5 Measurement Tools

The viewer supports **3D measurement tools**:

- Users can click two points to draw a line segment in 3D space
- The line is rendered over the volume with a digital label indicating its physical length (in µm or mm depending on pixel size)
- All measurements are stored as MeasurementLine objects and listed in the Measurement tab
- Users can **toggle visibility**, **delete**, or **export** measurements to CSV via the control panel.

This enables precise quantification of features directly in the 3D rendered view.

3.6 Streaming Renderer

The **streaming renderer** is activated automatically for volumes exceeding ~8 GB or can be toggled manually. Features include:

- Loads visible chunks of the volume dynamically during camera movement
- Maintains only a subset (e.g., 32) of chunks in GPU memory
- Reduces memory footprint to enable real-time rendering of 30–100 GB volumes on commodity GPUs
- Uses DirectX texture and resource management with background chunk loading and unloading

This approach improves scalability for large datasets at the cost of occasional streaming delay or lower detail for distant areas.

3.7 Export Capabilities

The 3D Viewer supports exporting:

- **Screenshots** (PNG or JPEG) of the current 3D view, with slices, labels, and measurements visible
- **Mesh exports** (via VoxelMeshExporter) to:
  - OBJ (Surface or Voxel mesh)
  - Binary STL
- Users can control mesh type, surface resolution, voxel size, and whether to exclude "Exterior" (label 0)
- The mesh export dialog (MeshExportDialog) exposes options for:
  - Threshold range
  - Material inclusion
  - Surface facet count (for decimation)
  - Clipping and cutting planes

The exporter supports both full memory-resident and **producer-consumer streaming** approaches using ProducerConsumerFileWriter to reduce RAM during large mesh generation.

N.B. The Mesh Exporter only export visible voxels in case a cutting plane is enabled.

3.8 Clipping and Cutting Planes

CTS offers advanced clipping and cutting tools for selective exploration:

- **Clipping Plane:** Infinite plane that clips voxels based on their distance and direction. Adjustable via:
  - Normal vector (XY, YZ, XZ)
  - Distance
  - Mirror (invert side)
- **Cutting Planes:** Axis-aligned planes with position and direction, supporting cuts along:
  - X, Y, or Z
  - Forward or backward

  - Controlled in real time with sliders in the ControlPanel

These tools allow researchers to reveal internal structures and explore cross-sections of the dataset in real time.

3.9 Debugging and Optimization

Debugging features include:

- **Debug Mode:** Enables wireframe or bounding-box rendering for geometry diagnostics
- **LOD Toggle:** Allows users to disable Level-of-Detail downsampling for full-resolution visualization
- **Streaming Toggle:** Turn streaming mode on/off depending on system performance and dataset size

Performance is automatically managed using:

- **Multi-threaded rendering** with render timer loop
- **Asynchronous loading** of texture chunks
- **Shader resource caching** for reuse across frames

Chapter 4: Modules

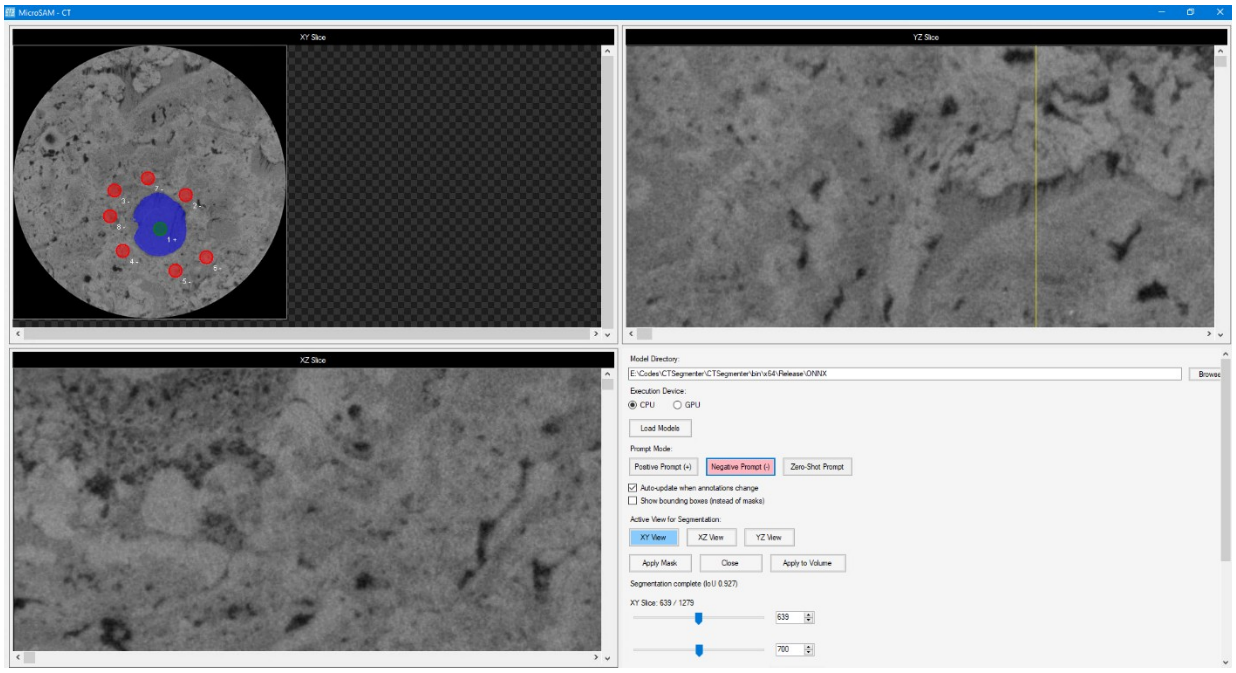
4.1 AI Segmentation Modules (ONNX Runtime-Based)

CTS integrates multiple AI-driven segmentation tools for CT volumes, all executed via ONNX Runtime for cross-platform inference. These modules share a common architecture: the CT volume is visualized in orthogonal views (XY, XZ, YZ) and users define *prompts* (points, boxes, or text) to guide segmentation. A first-stage **encoder** processes each slice (or view) once to produce image embeddings (cached in memory), and a second-stage **decoder** applies user prompts to generate binary masks. CUDA or CPU execution can be toggled via a radio button: on GPU, ONNX Runtime's CUDA provider is used; on CPU, an optimized CPU execution provider is selected. Caches of recent slice bitmaps prevent redundant rendering when navigating views.

Segment Anything CT (SAM2): This module is based on Meta's Segment Anything Model 2 (SAM2) (Ravi et al., 2024)[4], based on the evolution of Segment Anything Model (SAM) (Kirillov et al., 2023)[14]. In CTS, the user loads a pair of ONNX models (encoder and decoder) via the UI (Figure 3), toggles GPU/CPU, and clicks *Load Models*. The interface then lets the user select positive or negative point prompts (blue vs. red markers) on any orthogonal slice, or even draw a bounding box via an integrated helper. Clicking *Apply* runs the decoder ONNX session on the current embedding and prompts to produce one or more object masks. CTS automatically picks the mask with highest confidence score (IoU prediction) and rasterizes it in the active view's slice. A "Slice Alignment" feature ensures the mask aligns correctly if the user switches view orientation. The status bar shows segmentation progress and IoU scores for debugging.

For multi-slice volumes, CTS offers a **3D propagation** workflow: after segmenting one slice, the user can propagate the mask forward and/or backward over a user-defined range of slices. In sequential or parallel mode, each new slice is segmented using the previous slice's mask as a weak prior (via SAM decoder), stopping when confidence drops below threshold. The resulting 3D mask is applied back into the volume. Internally, a progress dialog shows the slice-by-slice propagation, and masks are applied to the CT volume (overwriting voxel labels). This enables an initial 2D prompt to bootstrap a near-3D segmentation. All intermediate encoder embeddings are cached and reused, so only the lightweight decoder runs per slice.

[14] Kirillov, A., Mintun, E., Ravi, N., Mao, H., Rolland, C., Gustafson, L., Xiao, T., Whitehead, S., Berg, A. C., Lo, W.-Y., Dollár, P., & Girshick, R. (2023). Segment Anything. arXiv:2304.02643



Model Directory: E:\Codes\CTSegmenter\CTSegmenter\bin\x64\Release\ONNX

Execution Device: [✓ CPU] GPU

[✓] Auto-update when annotations change
[ ] Show bounding boxes (instead of masks)

Segmentation complete (IoU 0.927)

XY Slice: 639 / 1279

639

700

Figure 3 – SAM pipeline. Processing a volume in CTS using SAM it’s quite easy. The user just needs to enter positive and negative prompts on the active view in order to feed the model and the segmented structure is automatically extracted. Clicking a second time on a point will delete it.

MicroSAM: µSAM (MicroSAM) (Archit et al., 2025)[3] is a microscopy-specialized variant of SAM trained for cellular/biological images. In CTS, MicroSAM supports the same ONNX encoder/decoder pipeline but adds **three prompt modes**: *positive*, *negative*, and *zero-shot*. The UI shows three toggle buttons. In positive/negative mode, the user clicks points on the slice to add or remove regions as with SAM2. In zero-shot mode, the tool automatically generates multiple object masks: pressing the “Zero-Shot” button triggers MicroSAM to predict all salient objects in the view and display them as separate translucent masks with bounding boxes. A *“Show Bounding Boxes”* checkbox toggles drawing of box outlines around each candidate. Users can then click on these boxes to toggle each object mask on/off (selected masks fill the material ID). All prompt changes can auto-update the mask if *Auto-Update* is checked. Feature extraction for microSAM is identical to SAM2 (encoder + decoder ONNX), with the addition that MicroSAM merges zero-shot boxes into a combined segmentation mask when requested. In implementation, MicroSAM maintains a dictionary of point-label pairs and caches the image embedding per slice. All ONNX sessions and cache handling mirror SAM2.

Grounding DINO: Grounding DINO (Liu et al., 2023)[4] is an open-vocabulary text-prompted object detector. In CTS, this module lets users enter a text query (e.g. “angular grain”, “pore”) and detect all matching structures on the current CT slice. The UI includes a text box for the prompt, a confidence slider, and *Detect*/**Apply** buttons. Upon *Detect*, CTS runs a custom ONNX Grounding DINO session: the text is tokenized, fed through the model together with the image slice, producing bounding boxes and confidence scores for all objects matching the prompt. These boxes are overlaid on the image viewer. Users can adjust the confidence threshold or select/deselect boxes, then click *Apply*. Several **mask creation methods** are offered (via a dropdown) to convert boxes into segmentation masks: e.g. taking the center point, corners, filled box, or grid of points within each box. Pressing *Send to SAM* uses the helper interface to pass chosen box-derived prompts into the SAM2 module, yielding refined polygon masks. In essence, Grounding DINO provides a text-guided detection front-end whose output can seed the SAM segmentation. The ONNX model is a slightly modified version of the official Grounding DINO (adjusted output tensor names for integration). GPU/CPU toggles allow using CUDA acceleration. UI elements include *Select All*, *Clear Selection*, and real-time status messages. This enables open-set detection: e.g. "find all screws" or "find cracks" in a scan, without retraining.

GLCM texture classifier: CTS quantifies local greyscale texture by analysing the statistical distribution of neighbouring grey-level pairs. For every user-selected patch (or for every sliding-window position during volume classification) the code constructs four grey-level co-occurrence matrices—at 0°, 45°, 90° and 135°—either on the full 256-level intensities or on a user-defined quantised scale for speed. From each matrix the classifier extracts a feature vector comprising classical Haralick statistics (contrast, dissimilarity, homogeneity, energy and correlation) and the extended moment-based metrics introduced by Sebastian V. et al. (2012), which improve sensitivity to subtle textural changes. The four orientation-specific vectors are averaged to obtain a rotation-robust signature. During training this signature is stored as the "reference" texture for the current material; during inference CTS computes the cosine- or histogram-intersection similarity between the reference vector and every new patch. Voxels whose similarity exceeds the user-set threshold are written into the material label mask, and the procedure can be executed slice-by-slice, across a slice range or over the whole volume with optional GPU acceleration (ILGPU stream kernels) to keep interactive speeds. In this way, the GLCM classifier adds a statistically grounded, orientation-aware texture channel that complements intensity- and geometry-based segmentation tools in CTS (Sebastian V. et al., 2012).[5]

4.2 Filtering and Resampling Modules

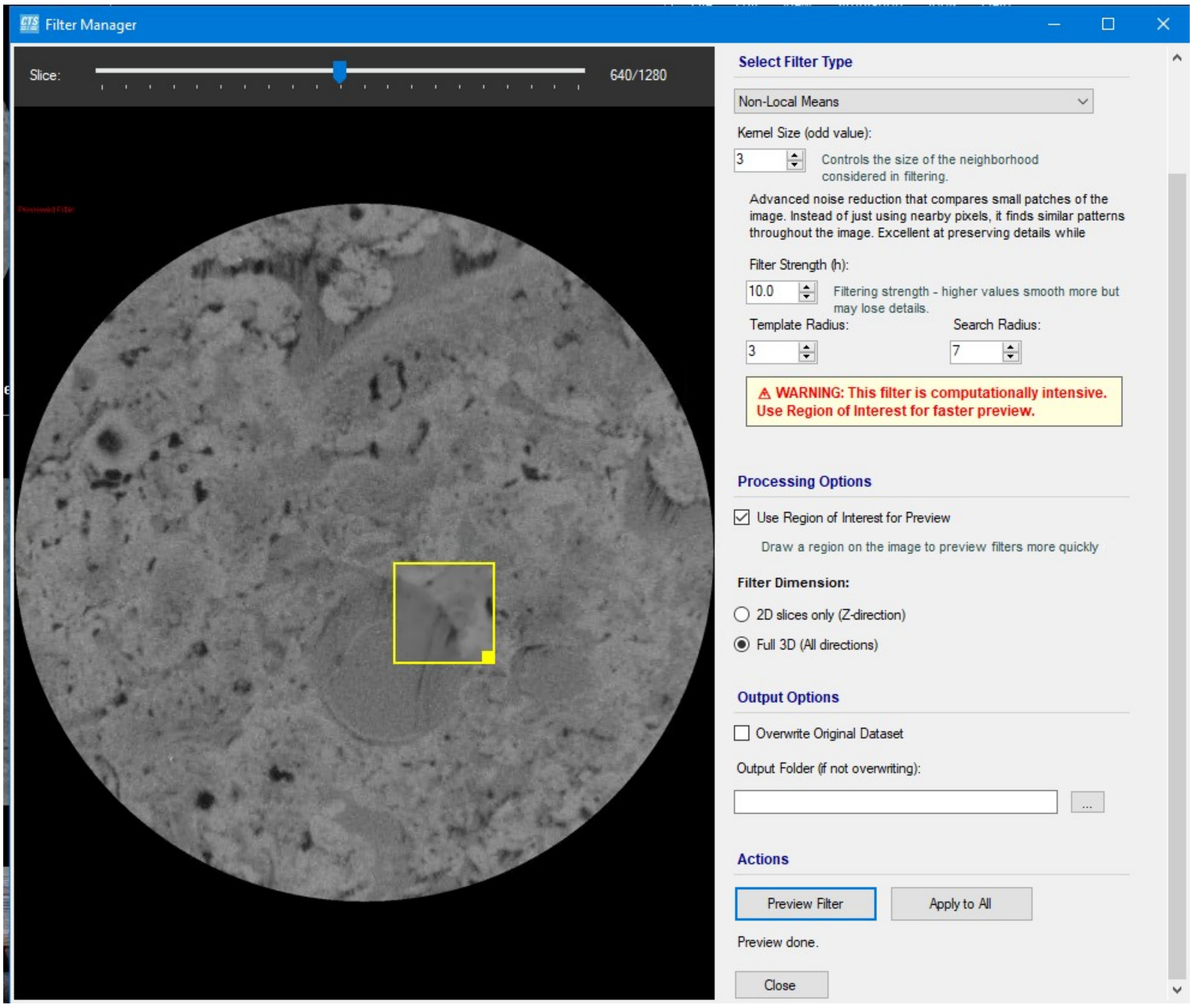
CTS's filtering subsystem (Figure 4) leverages ILGPU-based parallel kernels to accelerate common 2D/3D image filters on the GPU (with a CPU fallback). The **FilterManager** UI groups filter options and parameters. Available filters include **Gaussian smoothing**, **Median**, **Box (mean)**, **Non-Local**

Means (3D), **Bilateral**, **Unsharp (sharpening)**, and **Edge Detection**. Each filter appears as a choice in a drop-down or icon bar. When a filter is selected, controls for kernel size (e.g. radius or sigma) and other parameters are enabled. For Gaussian and Bilateral, σ values and radius are user-set; for Median, an odd-sized neighborhood is chosen; for Unsharp, a sharpening factor; etc. A *"2D/3D"* checkbox toggles whether to apply the filter on the current single slice or across all slices as a volume operation. An *ROI mode* checkbox enables filtering only within the current highlighted region. A preview panel on the UI shows a slice before/after filter if *Preview* is enabled.

3D Non-Local Means (NLM3D): The NLM3D filter (Buades et al., 2005)[15] computes a voxel-wise weighted average of the volume, weighting neighbors by patch similarity. Its full 3D variant is computationally intensive but implemented via ILGPU kernels. CTS automatically *chunks* the volume when GPU memory is limited: it splits the volume into overlapping sub-volumes ("blocks") that fit GPU memory, processes each block sequentially, and stitches the results. This strategy mirrors approaches in GPU CT processing where extra-large volumes are divided into blocks for computation. As each block is filtered, boundary regions are blended to maintain consistency. The NLM search radius and similarity sigma are set in the UI; default patch size might be $\sim 5 \times 5 \times 5$ voxels. Due to its cost, progress bars are shown during NLM execution.

After filtering, users choose to **overwrite** the current volume or **export** a new volume copy. The *Integrate/Resample* module (accessible via the same Filters dialog or a separate tool) can adjust the volume resolution. For example, a user may double the voxel size or resample to isotropic spacing. On GPU, this uses trilinear interpolation kernels (or averaging/integration if downsampling) to produce the new volume. The UI asks for target voxel size or scale factor and offers overwrite vs. export. Overwrite applies changes in memory to the open volume; export writes to a new file.

[15] Buades, A., Coll, B., & Morel, J.-M. (2005). A non-local algorithm for image denoising. In Proceedings of CVPR, Vol. 2, 60–65.


Filter Manager

Slice: 640/1280

Select Filter Type

Non-Local Means

Kernel Size (odd value): 3 — Controls the size of the neighborhood considered in filtering.

Advanced noise reduction that compares small patches of the image. Instead of just using nearby pixels, it finds similar patterns throughout the image. Excellent at preserving details while

Filter Strength (h): 10.0 — Filtering strength - higher values smooth more but may lose details.

Template Radius: 3

Search Radius: 7

⚠ WARNING: This filter is computationally intensive. Use Region of Interest for faster preview.

Processing Options

[✓] Use Region of Interest for Preview

Draw a region on the image to preview filters more quickly

Filter Dimension:

○ 2D slices only (Z-direction)

● Full 3D (All directions)

Output Options

☐ Overwrite Original Dataset

Output Folder (if not overwriting):

Actions

Preview Filter | Apply to All

Preview done.

Close

Figure 4 – Filter Manager. The filter manager allows the user to chose the kind of filter to apply to the dataset. A ROI preview box can be dragged and resized to have a preview of the applied filter on the XY slice.

4.3 Measurement Module

CTS supports manual 2D measurements directly on slice views. In any orthogonal viewer (XY, XZ, or YZ), the user can draw a *line segment* between two points. The software computes the real-world distance using the image pixel spacing. Internally, each **Measurement** record stores: a unique ID, a name/description, the view type (XY/XZ/YZ), the slice index, the integer start/end pixel coordinates, and the calculated distance (in meters). The distance is computed as the Euclidean distance in pixels times the voxel spacing: for example, in the XY plane,

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2} \quad * s_{xy}$$

Where s_{xy} is the XY pixel spacing (meters per pixel). For XZ and YZ views, one axis is actually the depth direction, but the same formula applies substituting the Z spacing accordingly. The computed distance is then formatted for display: distances <1 mm are shown in micrometers (µm), distances up to 1 m in millimeters (mm), and above 1 m in meters (m), matching the logic in the Measurement class.

Measurements are listed in the **MeasurementForm**, which contains a table view with columns ID, Name, View, Slice, Start (x,y), End (x,y), Distance, and Date. Each measurement's line is drawn on the slice in a distinct color (cycling through a preset palette). Double-clicking a row jumps the main view to that slice and highlights the line. A context menu on the list permits renaming or deleting entries. Buttons allow *Export CSV*, *Export Excel*, *Import CSV*, *Delete Selected*, and *Clear All*. Export functions write the full measurement table, including separate columns for distance in meters, micrometers, and millimeters. Import CSV can read a file with the same columns, recreating measurements (inheriting line colors by ID).

CTS's measurement UI provides an easy way to annotate lengths on slices. The underlying formulas use simple 2D geometry and the known pixel size. As implemented, a Measurement object stores the distance in meters and a formatted text (µm/mm/m) for display. Users can thus quantify anatomical features or artifacts by drawing lines across the volume and retrieving structured data for reporting or analysis.

Chapter 5: Particle Separator and Dataset Tools

The Particle Separator module (Figure 5) identifies and isolates discrete "particles" within a volumetric dataset by performing a 3D connected-component analysis on a binary or segmented image. Internally, the algorithm scans the volume (typically using 26-neighbor connectivity) and labels all contiguous foreground voxels as individual components. Once labeled, each component's properties are computed: its volume is the total count of voxels multiplied by the voxel volume, and its center (centroid) is the average of its voxel coordinates in physical space. Optionally, the module can also compute other shape descriptors (e.g. bounding box, principal axes) for each particle. These statistics (particle ID, volume, center coordinates) are output in a table and can be exported (e.g. CSV) for further analysis.

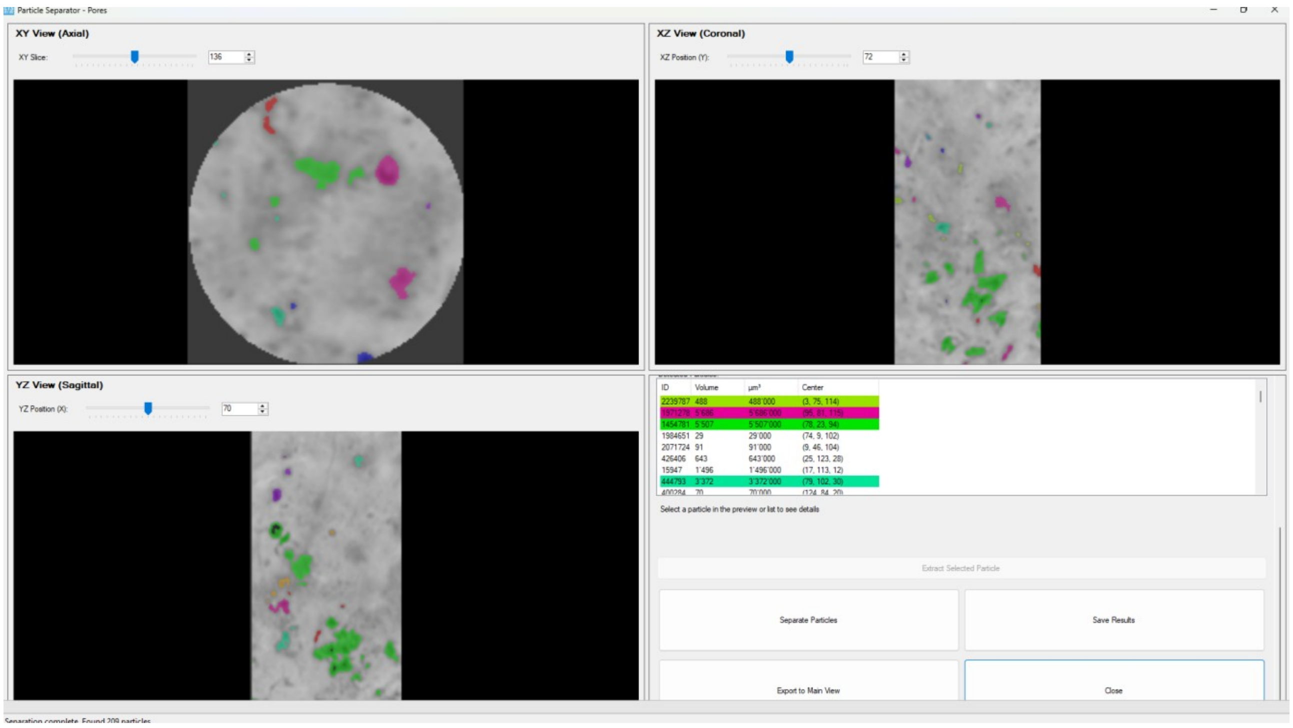
Performance Notes: The connected-components step can be memory- and compute-intensive for large tomographic volumes. To handle this, the implementation may divide (chunk) the dataset into overlapping sub-volumes, process each in parallel, then merge labels across chunk boundaries. If a GPU is available, a fast GPU-based kernel (via ILGPU) can accelerate the labeling; otherwise a multi-threaded CPU accelerator is used. In practice, ILGPU's CPU Accelerator enables debugging and provides parallel execution on multicore CPUs. For example, pseudocode for chunked processing might be:

```
Parallel.ForEach(chunks, chunk => { var labels = ComputeConnectedComponents(chunk); });
// merge chunk labels into global labeling map

```

If the GPU memory is insufficient or not present, the code "falls back" to CPU execution automatically, ensuring robustness.

A typical UI for the Particle Separator provides controls for Threshold, Minimum Object Size, and Connectivity (6-, 18-, or 26-neighbor). The user may also enable "GPU Acceleration" or "Multi-threading".

```
// Example: Compute centers and volumes of labeled particles
var stats = Image3D.RegionProperties(labels, voxelSize);
foreach(var region in stats) {
Console.WriteLine($"Particle {region.Label}: Volume={region.Volume:F2} mm^3,
Center=({region.Centroid.X:F2},{region.Centroid.Y:F2},{region.Centroid.Z:F2})");
}

```



ID	Volume	µm³	Center
2239787	488	488'000	(3, 75, 114)
1971278	5'686	5'686'000	(95, 81, 115)
1454781	5'507	5'507'000	(78, 23, 94)
1984651	29	29'000	(74, 9, 102)
2071724	91	91'000	(9, 46, 104)
426406	643	643'000	(25, 123, 28)
15947	1'496	1'496'000	(17, 113, 12)
444793	3'372	3'372'000	(79, 102, 30)
400284	70	70'000	(124, 84, 20)

Select a particle in the preview or list to see details

Separation complete. Found 209 particles

Figure 5 – Particle separator. The particle separator produces a list of non-connected objects per single XY slice or per 3D volume. The results can be exported. The module can also export the selected particles in the list in a separate material. Useful for separating big volumes from smaller particles.

Here RegionProperties might compute volume and centroid for each connected component label. The module can also export a 3D histogram of particle volumes or sizes, useful for characterizing particle size distributions. Dynamic charts show, for example, the count of particles versus volume (number-weighted or volume-weighted histograms).

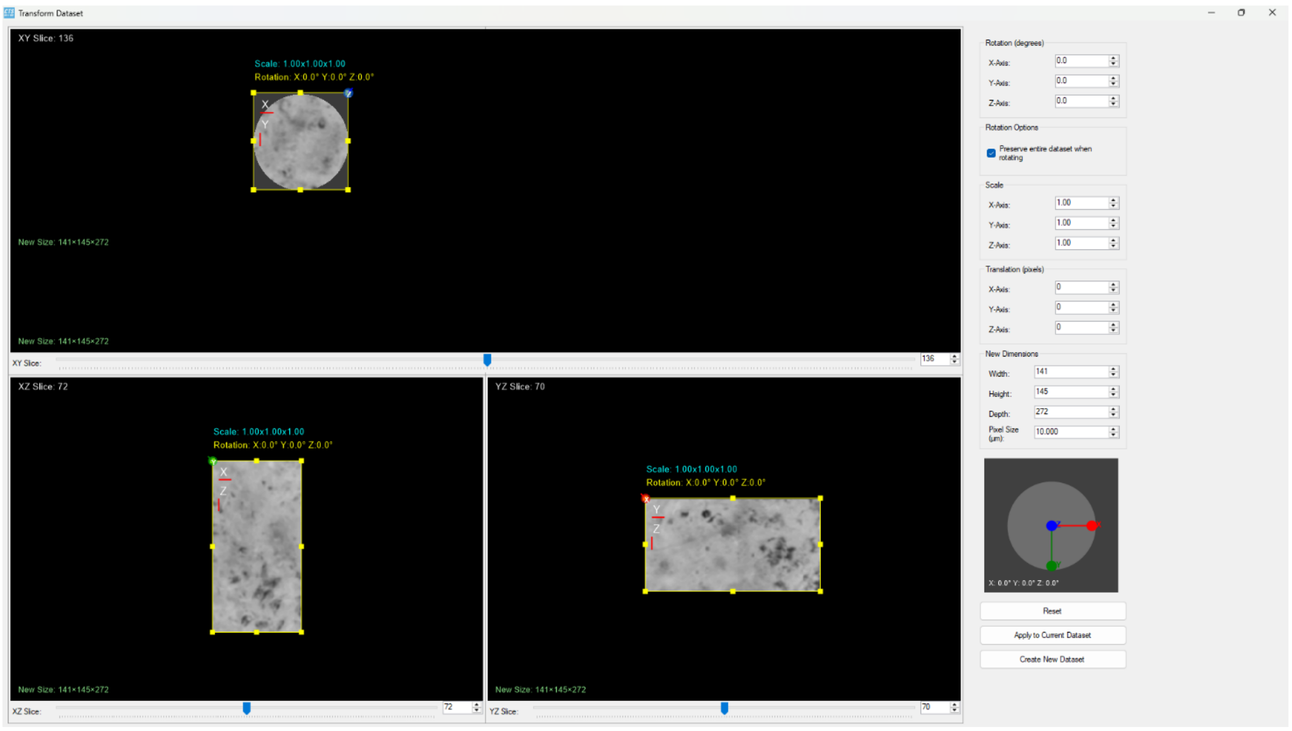
The Material Statistics module generalizes similar analysis to multi-material (labeled) datasets. Given a volume where each material is encoded by a distinct integer label, the module computes per-material metrics: the count of voxels, total volume ($count \times voxelVolume$), and surface area of that material. The surface area can be estimated by counting the number of label-boundary voxel faces or by more accurate methods (e.g. marching cubes). The output table lists each material's name (or label value), its volume fraction, and estimated surface area. A histogram of material volumes (or surface areas) can be generated for further analysis or quality control. For instance, a table might report that Material_1 has volume 1240 mm^3 and surface area 350 mm^2, etc. The histogram export facility allows saving the size distribution of connected volumes (similar to particle size analysis).

```python
# Pseudocode: compute material volumes and counts
import numpy as np
labels = dataset  # 3D array of material labels
unique, counts = np.unique(labels, return_counts=True)
voxelVolume = dx*dy*dz
for lab, count in zip(unique, counts):
    print(f"Material {lab}: Volume={count*voxelVolume:.2f} mm³, Count={count}")

```

The Transform Dataset tool allows the user to shift, rotate, or scale the entire volume in physical space. In the UI, fields or sliders are provided for X/Y/Z translation, three Euler rotation angles (pitch, yaw, roll in degrees), and a uniform Scale factor. Applying these transforms modifies the dataset's origin or sampling; for example, rotating by 90° around Z reorients the data volume. Internally, these transforms are applied via an affine transformation matrix. For example, a rotation about Z by θ uses a matrix

$$R_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

and translations add an offset to voxel coordinates. Interpolation (nearest or linear) is used to resample voxel values.


Transform Dataset

XY Slice: 136
Scale: 1.00x1.00x1.00
Rotation: X:0.0° Y:0.0° Z:0.0°
New Size: 141×145×272

XZ Slice: 72
Scale: 1.00x1.00x1.00
Rotation: X:0.0° Y:0.0° Z:0.0°
New Size: 141×145×272

YZ Slice: 70
Scale: 1.00x1.00x1.00
Rotation: X:0.0° Y:0.0° Z:0.0°
New Size: 141×145×272

Rotation (degrees)
X-Axis: 0.0
Y-Axis: 0.0
Z-Axis: 0.0

Rotation Options
[✓] Preserve entire dataset when rotating

Scale
X-Axis: 1.00
Y-Axis: 1.00
Z-Axis: 1.00

Translation (pixels)
X-Axis: 0
Y-Axis: 0
Z-Axis: 0

New Dimensions
Width: 141
Height: 145
Depth: 272
Pixel Size (µm): 10.000

X: 0.0° Y: 0.0° Z: 0.0°

Figure 6 – Transform Dataset. The transform dataset module allows the user to cut or rotate the dataset along the 3 directions. This is for example useful in order to vertically align cores (Z direction).

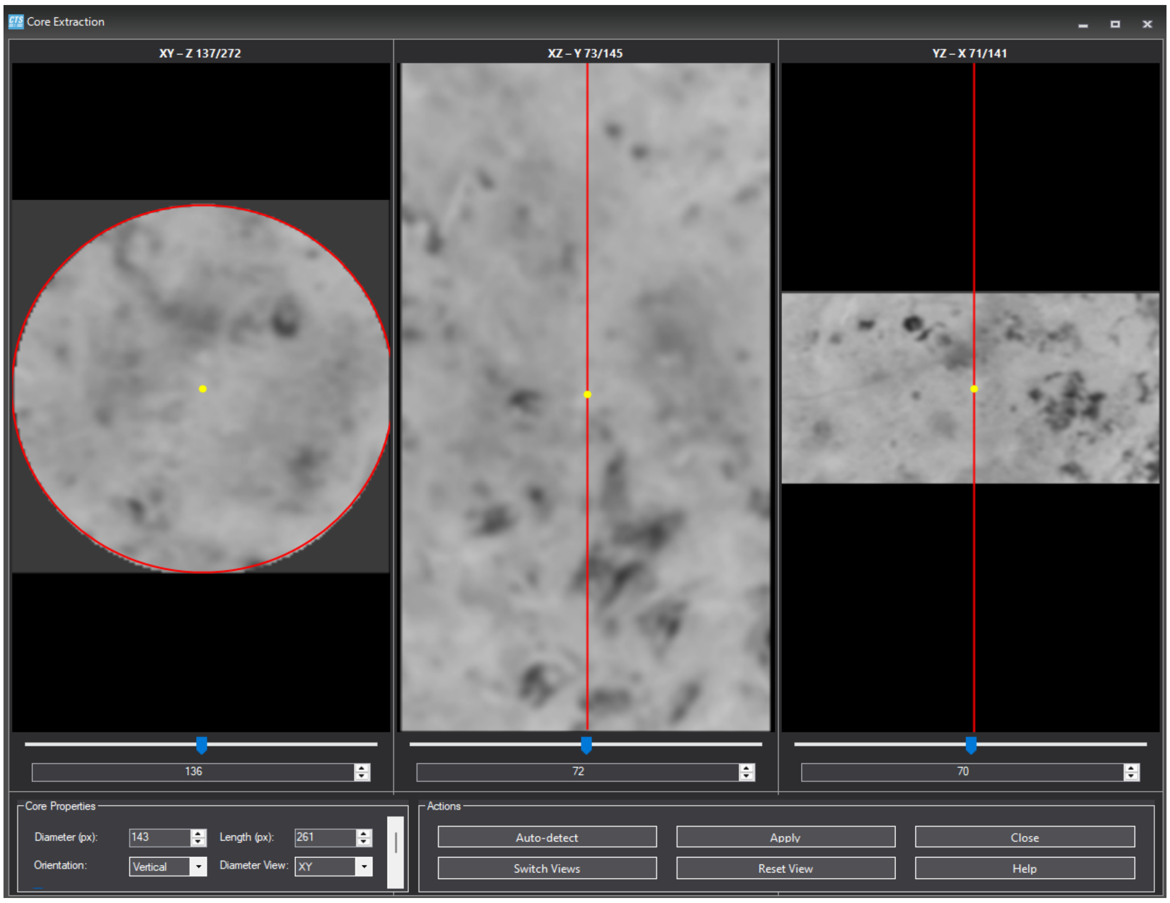
```csharp
// C# example: apply a 90° rotation about Z
var transform = AffineTransform.Rotation(axis: Vector3.UnitZ, angle: Math.PI/2)
                * AffineTransform.Translation(10.0, 0.0, 5.0);
var transformedVolume = dataset.ApplyTransform(transform, interpolation:
       InterpolationMode.Linear);

```

After transformation, metadata such as voxel origin and direction cosines are updated so that subsequent measurements remain in correct physical units.

Chapter 6: Petrophysical Tools

Core Extraction Tool

The Core Extraction tool isolates a cylindrical sub-volume (core) from the scanned data, emulating the laboratory core sample geometry. The user defines a cylindrical region by specifying either a 3D axis (two points or a line) and radius, or by clicking two points to establish the base and height of the cylinder. The algorithm then extracts all voxels within that cylinder. Internally, each voxel's position (x, y, z) is tested against the cylinder equation. For example, for a cylinder aligned along the Z-axis at $(x0, y0)$ with radius r, a voxel is inside if$(x - x_0)^2 + (y - y_0)^2 \leq r^2$. For arbitrary axis alignment, the algorithm transforms coordinates to the cylinder's reference frame first.



Diameter (px): 143 Length (px): 261
Orientation: Vertical Diameter View: XY

Figure 7 – Core Extraction Tool. The core extraction tool interface allows for a quick extraction (suppression of surrounding air voxels) of a cylindrical core from the scan. Good to be applied after filtering, transform and segmentation. Air voxels in the corner will be assigned to exterior.

Extraction parameters include Core Radius, Core Length (or endpoints), and Centering method (e.g. centroid of input mask or manual offset). The tool supports trimming beyond dataset bounds. Once defined, the tool outputs a new dataset of just the core, potentially with an applied mask.

```
# Pseudocode for core extraction along Z-axis
core_center = (cx, cy)
radius = R
core_mask = ((X - cx)**2 + (Y - cy)**2) <= R**2
core_volume = original_volume.copy()
core_volume[~core_mask] = background_value

```

This is often used in petrophysics to reduce a full rock scan to a cylindrical core shape for flow simulations or comparison with laboratory core measurements.

Pore Network Modeling

The Pore Network Modeling module constructs a network of pores and throats from a segmented volume, often representing the pore space of a rock. The process typically follows these steps:

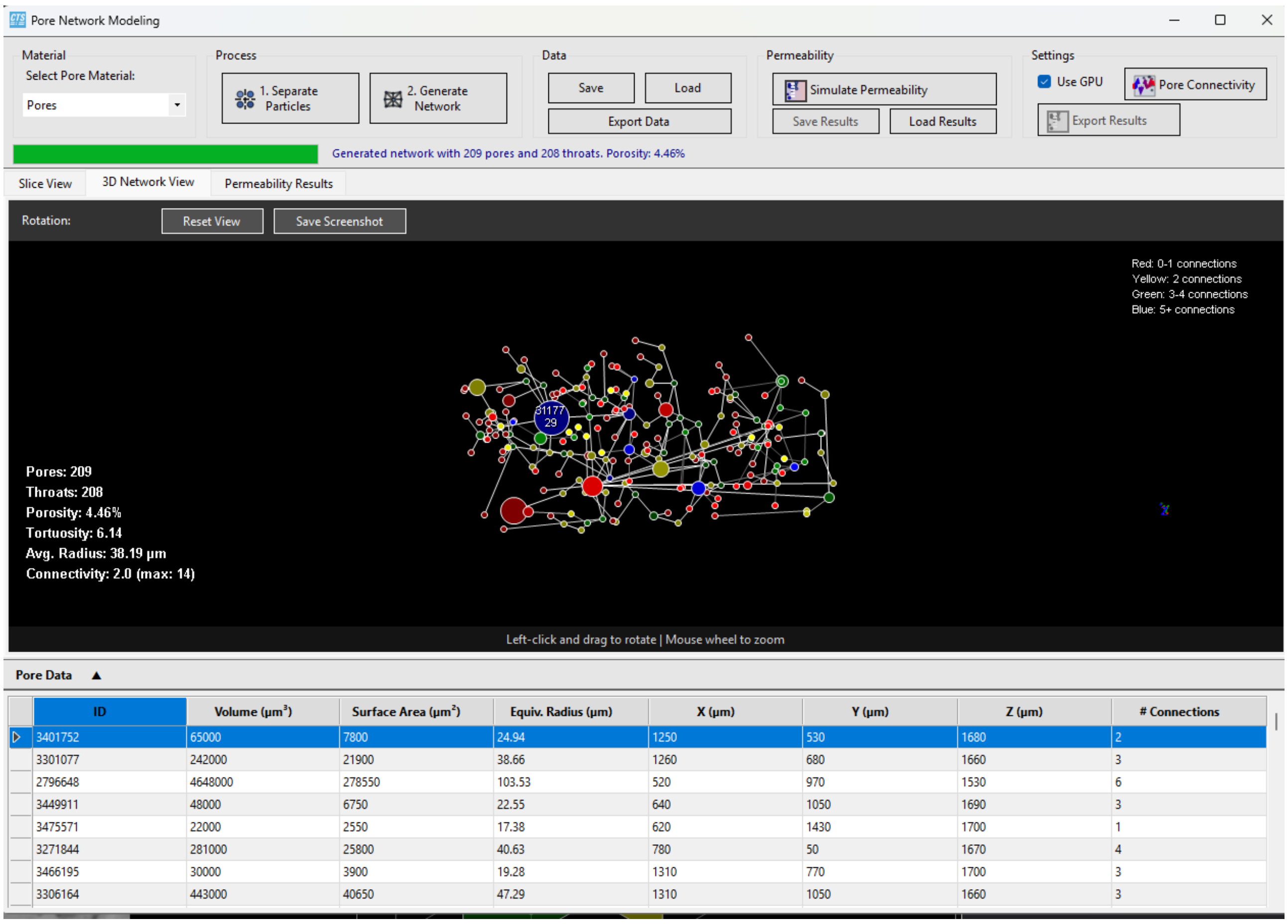
1. Pore (Particle) Identification: Each pore body (or “particle”) in the segmented image is treated as a network node. This can be done by computing the distance transform within the pore space and identifying maxima, or by labeling connected pore regions and computing their centroids. This conversion from voxels to pore centers is analogous to extracting “particles” in the pore phase.
2. Throat Generation: Potential throats are generated between nearby pore centers. One approach is to consider pairs of pores within a cutoff distance max throat length factor times the sum of their radii. If two pores’ volumes overlap or nearly touch (within an overlap tolerance), a throat is created connecting them. The throat’s properties (e.g. cross-sectional area) are estimated based on the geometry of the overlap region.
3. Connectivity Rules: The algorithm may use a max throat length factor to limit connectivity to physically reasonable neighbors, and an overlap factor to ensure throats only when pores are sufficiently close. For example, one might allow a throat if the distance between pore centers is less than $L = 2.5 * (R_i + R_j)$ (where R_i is an effective pore radius) and if the throat cross-section area exceeds a threshold.
4. Tortuosity Calculation: After the network is built, the code traces shortest paths between inlet and outlet pores along the X, Y and Z axes, compares the average path length (L_e) with the straight sample length (L) and reports the geometric tortuosity $\tau = (L_e/L)^2$. The three

directional values are averaged to give a single tortuosity that can be fed into Kozeny-Carman or other transport correlations.

The core computation, finding neighbors and computing throat sizes, is parallelizable. Both a CPU and GPU version are implemented via ILGPU: the GPU kernel can rapidly evaluate many pore pairs in parallel. ILGPU's JIT kernels enable writing the same C# code to run on CUDA or CPU. For example, the GPU kernel might assign one thread per pore to examine nearby candidates. If a GPU is available, the network assembly is greatly accelerated; otherwise the CPU path (possibly multi-threaded) is used.

```
// Simplified pseudocode for throat generation (CPU or ILGPU)
foreach (pore i) {
  for (each pore j > i) {
    double d = Distance(pore[i].Center, pore[j].Center);
    if (d < maxFactor*(r[i]+r[j])) {
      // compute overlap cross-section area
      double A = ComputeThroatArea(pore[i], pore[j]);
      if (A > areaThreshold)
        network.AddThroat(i,j, area:A);
    }
  }
}

```

In the UI, the user steps through: (a) select pore (particle) image, (b) set throat parameters (max length factor, overlap), (c) run Generate Network, (d) preview network connectivity, and (e) export as a file (e.g. an OpenPNM network JSON).



Generated network with 209 pores and 208 throats. Porosity: 4.46%

Pores: 209
Throats: 208
Porosity: 4.46%
Tortuosity: 6.14
Avg. Radius: 38.19 µm
Connectivity: 2.0 (max: 14)

ID	Volume (µm^3)	Surface Area (µm^2)	Equiv. Radius (µm)	X (µm)	Y (µm)	Z (µm)	# Connections
3401752	65000	7800	24.94	1250	530	1680	2
3301077	242000	21900	38.66	1260	680	1660	3
2796648	4648000	278550	103.53	520	970	1530	6
3449911	48000	6750	22.55	640	1050	1690	3
3475571	22000	2550	17.38	620	1430	1700	1
3271844	281000	25800	40.63	780	50	1670	4
3466195	30000	3900	19.28	1310	770	1700	3
3306164	443000	40650	47.29	1310	1050	1660	3

Figure 8 – PNM Interface. Pore Network modeling results after network generation. Details about pores are displayed in the bottom table and in the visualization. Each pore has an ID (written in the circles). Average Tortuosity on the network is also displayed and reused in permeability simulation (see Permeability Simulation chapter).

The generated pore network can be further analyzed or used in flow simulations. For reference, this approach is similar to methods in the PoreSpy toolkit (which provides Python tools for pore network extraction).

Permeability Simulation

The Permeability Simulation tool solves fluid flow through the pore network using Darcy's Law. The fundamental governing equation is

$$Q = -\frac{kA}{\mu} \cdot \frac{\Delta P}{L}$$

where Q is volumetric flow rate, k is intrinsic permeability, A is cross-sectional area, μ is fluid viscosity, and $\frac{\Delta P}{L}$ is the pressure gradient. In a pore network, each throat between pores acts like a conduit with conductance $g \propto \frac{A}{\ell}$ (where A is throat area and ℓ throat length) (Pellerin & Zinszner, 2008; Bear, 1972)[16]. The network of pores and throats yields a system of linear equations (Kirchhoff's laws) for flow: at each pore node, $sum\ of\ flows\ =\ 0$. Boundary conditions are set by fixing the pressure at inlet pores (e.g. $P = P_{in}$) and outlet pores ($P = P_{out}$), then solving for internal pore pressures.

Once pressures are obtained (via e.g. a sparse linear solver), the total flow Q is computed. Then the effective permeability k_{eff} is calculated from Darcy's Law rearranged: $k_{\mathrm{eff}} = -\frac{Q\mu L}{A\Delta P}$. To account for porosity and pore geometry, a Kozeny–Carman correction can be applied: the classical Kozeny–Carman relation predicts permeability from porosity ϕ and specific surface area S, often written

$$k \approx \frac{\phi^3}{c(1-\phi)^2} \cdot \frac{d^2}{C},$$

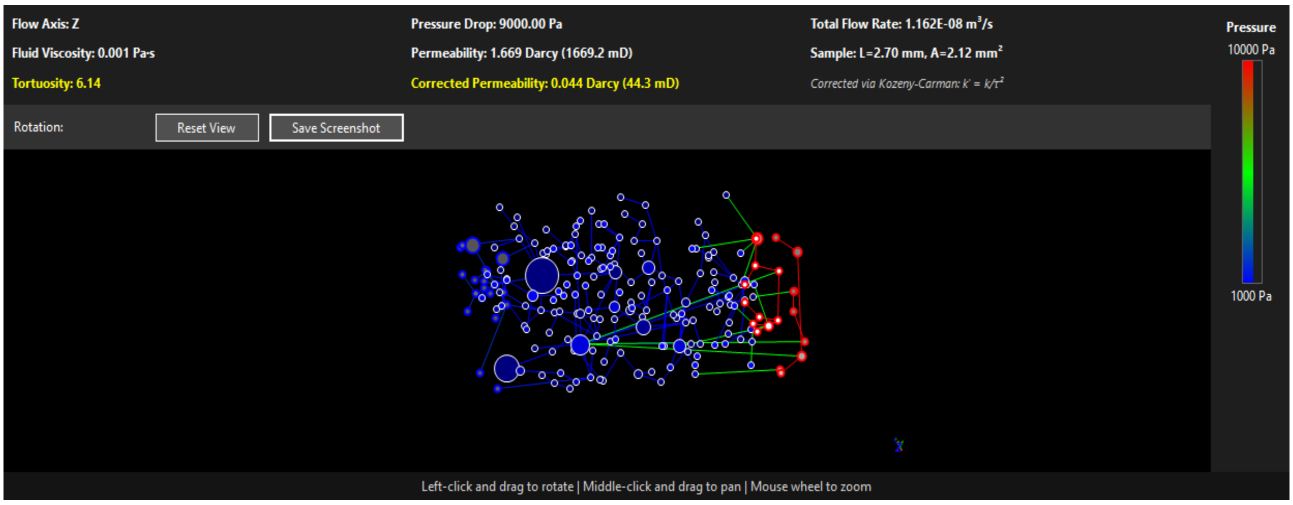
where c is a Kozeny constant and d a mean grain diameter. In practice, the simulator outputs both the raw network permeability and a Kozeny–Carman estimate for comparison (Pellerin & Zinszner, 2008; Bear, 1972)[10].

The solver is implemented in both CPU and GPU variants. On the GPU, ILGPU kernels can perform the sparse matrix-vector products and Jacobi/CG iterations, taking [17]advantage of parallelism. On CPU, a multi-threaded sparse solver is used (e.g. Conjugate Gradient or direct solver). Performance testing shows the GPU solver can be orders of magnitude faster for large networks, but the CPU solver remains robust for modest sizes. Importantly, ILGPU's CPU Accelerator allows using the same code path for debugging or when GPUs are unavailable.

```python
# Simplified flow solve (conceptual)
K = assemble_conductance_matrix(pores, throats)
b = set_boundary_conditions(num_pores, P_in, P_out)
P = scipy.sparse.linalg.spsolve(K, b)  # pressures at pores
Q = compute_flow(P, throats)
k_eff = -Q * mu * length / (area * (P_in - P_out))

```

[16] Pellerin, F.-M., & Zinszner, B. (2008). *A Geoscientist's Guide to Petrophysics*. Editions Technip.
Bear, J. (1972). *Dynamics of Fluids in Porous Media*. Dover Publications.

The geometric tortuosity τ, determined from shortest-path analysis of the reconstructed network, represents the increased transport length and yields a corrected permeability $k_{\mathrm{corr}} = \frac{k_{\mathrm{raw}}}{\tau^2}$. This quadratic reduction captures the additional viscous dissipation imposed by elongated flow paths and aligns with the Kozeny–Carman formulation, where τ^2 appears alongside porosity and specific surface area. Presenting both $k_{raw}$and k_{corr} offers a direct assessment of the raw network outcome and a tortuosity-normalised permeability suitable for comparison with core-plug measurements or incorporation into continuum-scale reservoir simulations.


Flow Axis: Z
Fluid Viscosity: 0.001 Pa·s
Tortuosity: 6.14
Pressure Drop: 9000.00 Pa
Permeability: 1.669 Darcy (1669.2 mD)
Corrected Permeability: 0.044 Darcy (44.3 mD)
Total Flow Rate: 1.162E-08 m^3/s
Sample: L=2.70 mm, A=2.12 mm^2
Corrected via Kozeny-Carman: $k' = k/\tau^2$

Left-click and drag to rotate | Middle-click and drag to pan | Mouse wheel to zoom

Figure 9 – Permeability Simulation results with corrected permeability after Kozeny-Carman / Tortuosity correction.

This permeability tool is analogous to routines in PoreSpy[18]'s "flow" or OpenPNM[19], but fully integrated into the CTS UI. By exposing parameters (fluid viscosity, boundary pressures) and providing log outputs of convergence, users can analyze flow through the pore structure in a research context.

Lattice Boltzmann method (LBM)

In addition to the Darcy-based solver, the CTS engine can solve flow on a voxelised version of the pore network with a 3-D, 19-velocity-direction (D3Q19) Lattice Boltzmann scheme. Each voxel is flagged as solid or fluid; bounce-back boundary conditions enforce no-slip at the grain surface, while Zou-He pressure boundaries impose the inlet and outlet pressures. Collision is handled with a single-relaxation-time BGK operator ($\tau = \nu / c_s^2$) , and streaming updates voxel distributions in parallel.

[18] Gostick, J. T., Khan, Z. A., Tranter, T. G., Kok, M. D. R., Agnaou, M., Sadeghi, M., & Jervis, R. (2019). PoreSpy: A Python toolkit for quantitative analysis of porous media images. *Journal of Open Source Software, 4*(37), 1296.
[19] Gostick, J. T., Khan, Z. A., Tranter, T. G., Kok, M. D. R., Agnaou, M., Sadeghi, M., & Jervis, R. (2016). OpenPNM: A pore network modeling package. Computing in Science & Engineering, 18(4), 60–74.

When the residual drop in velocity between successive iterations falls below 10^{-6} (or a user-defined maximum iteration count is reached) the code integrates the velocity field over a mid-sample cross-section to obtain the total volumetric flux and derives an effective permeability with Darcy's law. Because the algorithm is local, it maps efficiently to GPUs; for high-resolution reconstructions the ILGPU backend typically brings one to two orders of magnitude speed-up compared with a multi-core CPU run, yet a CPU Accelerator path is retained for debugging or hardware-limited deployments.

Navier–Stokes / Forchheimer solver

Where higher Reynolds numbers or inertial effects are suspected, a lightweight continuum approach is also available. The code analyses the throat statistics to estimate an average hydraulic diameter and, using the imposed pressure gradient, solves a one-dimensional form of the steady Navier–Stokes equations augmented with a Forchheimer ($\beta\rho v^2$) term to capture non-Darcy drag. An iterative scheme updates the superficial velocity until convergence of $10^{-6}\ m\ s^{-1}$ is achieved, yielding an apparent permeability that naturally transitions from linear (Darcy) to quadratic (inertial) regimes. The routine reports the representative Reynolds number, the fitted Forchheimer coefficient, and flags cases where $Re > 10$ warning that full turbulence modelling may be required. As with the other methods, a tortuosity correction can be applied post-solve to facilitate comparison with core-plug measurements or Kozeny–Carman predictions.

Chapter 7: Band Detection Tool

The Band Detection module identifies stratified bands in a volumetric core sample, such as density banding or bleaching patterns in coral or ice cores. It uses a variance-based detection pipeline: the 3D image is first optionally smoothed with a Gaussian blur (a low-pass filter that reduces noise). Gaussian blurring (with user-defined sigma) ensures that large-scale band structures are preserved while small-scale noise is suppressed. Mathematically, applying a Gaussian blur is equivalent to convolving the image with a 3D Gaussian kernel.

Next, a local variance filter is applied. For each voxel (or along a projected axis, e.g. the core depth), the local variance of intensity is computed over a sliding window or kernel. Variance is defined as $\sigma^2 = \frac{1}{N}\sum_i (x_i - \mu)^2$, where μ is the local mean. Peaks in the variance profile correspond to transitions between bands (where intensity changes are maximal). A GPU-accelerated variance kernel computes this filter in parallel across the volume: each thread sums squares and means in its neighborhood, then computes variance. The CUDA/ILGPU implementation drastically speeds up this convolution compared to CPU, enabling real-time previews.

After computing the variance image (or a 1D profile along the core axis), a peak detection step identifies prominent peaks using a peak prominence criterion. Peak prominence measures how much a peak stands out relative to its surroundings. Only peaks above a user-defined prominence threshold are kept, reducing false detections. The detected peaks are interpreted as band boundaries. The UI allows the user to adjust the Gaussian sigma and prominence threshold, and see an overlaid plot (variance vs. depth) with marked peaks. Formally, if $V(z)$ is the variance as a function of depth z, peaks are points where $V(z)$ is a local maximum and the difference to surrounding minima exceeds the prominence. This is analogous to SciPy's find_peaks logic. In practice, we compute variance on 2D slices or on a projection of the 3D core to 1D for speed and easier interpretation.

After peak detection, a band mask is generated: regions between detected boundaries are labeled as individual bands. This mask can be exported or combined with the original image for visualization. For example, the tool can produce a composite RGB slice where each band is highlighted in a different color overlay, helping in visual inspection. Use cases for this workflow include coral bleaching research, where density banding could indicate stress events. For instance, bleached layers may appear as high-contrast bands; this tool automates their detection to quantify band spacing or thickness. Ice-core and tree-ring analyses use banding detection to infer chronological layers (Winstrup et al., 2012[20]; Poláček et al., 2023[21]).

[20] Winstrup, M., Svensson, A. M., Rasmussen, S. O., Winther, O., Steig, E. J., & Axelrod, A. E. (2012). An automated approach for annual layer counting in ice cores. Climate of the Past, 8, 1881–1895.
[21] Poláček, M., Arizpe, A., Hüther, P., Weidlich, L., Steindl, S., & Swarts, K. (2023). Automation of tree-ring detection and measurements using deep learning. Methods in Ecology and Evolution, 14(9), 2233–2242.

Chapter 8: Acoustic Wave Simulation Framework

8.1 Overview of the Acoustic Module

The CTS acoustic module is designed to propagate elastic waves through CT-derived volumetric data, enabling high-fidelity ultrasonic testing and fracture analysis on segmented core samples. Its purpose is to model both P-waves (pressure waves) and S-waves (shear waves) in an elastic medium reconstructed from segmented CT volumes. In practice, users import segmented CT data (labelled voxels) into an AcousticVolume structure (containing voxel labels and density tags). The module simulates wave propagation to assess material properties, core integrity, and potential fracture zones (e.g. ultrasonic nondestructive testing, core evaluation, fracture prediction). The raw volume (byte array of labels) is paired with material densities and physical material-related parameters, (from a library or user input) to define the simulation domain.

Input data: segmented 3D CT volume (voxel labels) and assigned material densities/elastic moduli.
Applications: ultrasonic pulse propagation, velocity tomography, fracture modeling in rock cores.
Computing feature: direct wave simulation on segmented CT voxels. CTS is the only software integrating full-elastodynamic, plastic and brittle physics on such data.

8.2 Physical Modeling

CTS solves for each varying-density voxel the full elastodynamic equations for an isotropic solid. P-waves (compressional) and S-waves (shear) naturally emerge from the coupled stress-strain relations. The governing continuum equations are Newton's second law $\rho \frac{\partial^2 \boldsymbol{u}}{\partial t^2} = \nabla \cdot \boldsymbol{\sigma}$ and Hooke's law for elasticity $\boldsymbol{\sigma} = \lambda(\nabla \cdot \boldsymbol{u})\boldsymbol{I} + 2\mu\epsilon$ where λ, μ are Lamé parameters and ϵ is strain. From these one derives two wave modes: longitudinal (P) and transverse (S) waves. For a typical medium, the P-wave and S-wave speeds are

$$V_P = \sqrt{\frac{K + \frac{4}{3}G}{\rho}}, \quad V_S = \sqrt{\frac{G}{\rho}}$$

where ρ is density, K the bulk modulus, and G the shear modulus. Equivalently, in terms of Young's modulus E and Poisson's ratio ν:

$$V_P = \sqrt{\frac{E(1-\nu)}{\rho(1+\nu)(1-2\nu)}}, \quad V_S = \sqrt{\frac{E}{2\rho(1+\nu)}}$$

These formulas are implemented internally (via Lamé constants λ, μ; see below) (Aki & Richards, 2002)[22]. The wave equations are discretized on a 3D staggered grid using finite differences (Virieux, 1986)[23]. A first-order-in-time, second-order-in-space FD scheme updates stress and velocity on alternate half-steps. For each voxel in the chosen material region, the code computes velocity gradients (e.g. $\frac{\partial vx}{\partial x}$, etc.) by central differences, updates stresses via Hooke's law, then updates velocities from stress divergence. For example, in code:

```
double dvx_dx = (vx[x+1,y,z] - vx[x-1,y,z])/(2*pixelSize);
double dvy_dy = (vy[x,y+1,z] - vy[x,y-1,z])/(2*pixelSize);
double dvz_dz = (vz[x,y,z+1] - vz[x,y,z-1])/(2*pixelSize);
double volumetricStrain = dvx_dx + dvy_dy + dvz_dz;
double dsxx = dt * (lambda * volumetricStrain + 2 * mu * dvx_dx);
double dsxy = dt * (mu * (dvx_dy + dvy_dx));
// ...

```

These increments ds are added to the existing stress field. After the elastic predictor, optional plastic (Mohr–Coulomb) and brittle-damage corrections (Jaeger et al., 2007)[24] are applied: if enabled, the stresses are adjusted to enforce yield or tensile failure conditions. Finally, the velocity components (vx, vy, vz) are updated from the stress divergence via explicit time stepping. Each time the wave crosses a voxel, Vp and Vs are specifically recalculated on the base of that voxel's density, in this way volume density variability is preserved across the whole material.

Units throughout are SI: pressure in Pa, density in kg/m³, velocity in m/s. In code, user-entered Young's modulus (in MPa) is converted to Pa ($\times 10^6$). The pixel size (CT voxel spacing) is supplied

[22] Aki, K., & Richards, P. G. (2002). Quantitative Seismology: Theory and Methods (2nd ed.). University Science Books.
[23] Virieux, J. (1986). P-SV wave propagation in heterogeneous media: Velocity-stress finite-difference method. Geophysics, 51(4), 889–901.
[24] Jaeger, J. C., Cook, N. G. W., & Zimmerman, R. (2007). Fundamentals of Rock Mechanics (4th ed.). Wiley-Blackwell.

in meters (respecting the actual PixelSize from MainForm), so all spatial derivatives and distances use correct physical scaling.

8.3 Material Properties Input

Users define material properties through the GUI or library: each material has a density ρ (kg/m³), Young's modulus E (MPa), and Poisson's ratio ν. The code immediately computes the shear modulus $G = \frac{E}{2(1+\nu)}$ and bulk modulus $K = \frac{E}{3(1-2\nu)}$. Internally these yield the Lamé constants:

```
double E = YoungsModulus * 1e6;
mu0 = E/(2*(1+poissonRatio));
lambda0 = E*poissonRatio/((1+poissonRatio)*(1-2*poissonRatio));

```

The AcousticSimulationForm has fields/controls for these (see MaterialProperties class, which stores YoungsModulus and PoissonRatio in MPa). Users select which material ID (label) to excite; TX/RX transducer positions will be set in that material.

The material flags Elastic, Plastic, Brittle (checkboxes) enable the corresponding physics models. For example, checking Plastic activates the Mohr–Coulomb yield logic (scale deviatoric stress if $\tau + p\, sin\, \phi > c\, cos\, \phi$), and Brittle activates tensile damage. These settings are read into the simulator (useElasticModel, usePlasticModel, useBrittleModel in AcousticSimulator).

For completeness, wave speeds are computed by the module from the user inputs if needed. The user can check the "Auto Elastic Props" box in the form, which will compute ν from a desired Vp/Vs ratio or vice versa (via calibration), but otherwise the simulator uses the above formulas.

8.4 Calibration Module

CTS includes a calibration system (class CalibrationManager) for fitting material elastic parameters to experimental data. Users enter calibration points that contain measured velocities and densities for a material (or multiple materials) under known conditions. There are two input modes: (1) Vp/Vs ratio only, or (2) Separate Vp and Vs values. Each point also includes the measured density, confining pressure, and optionally volume fraction. From this data, CTS fits simple models to predict Young's modulus vs density and Poisson's ratio vs Vp/Vs. For each confining pressure group, it performs a linear regression so that:

$$E \approx a\rho + b, \quad \nu \approx c\left(\frac{V_p}{V_s}\right) + d$$

logging lines such as "Created density-to-Young's model: $E = a \cdot \rho + b \quad (R^2 = \dots)$ and "Created VpVs-to-Poisson model: $\nu = c \cdot \left(\frac{V_p}{V_s}\right) + d \quad (R^2 = \dots)$. These models allow automatic calibration: given a new material density and/or measured Vp/Vs ratio, the system predicts the best-fitting E and ν for the simulation. For instance, PredictPoissonRatio(double vpVsRatio) uses the fitted linear model (slope and intercept) to output ν. The calibration UI (CalibrationDialog) aids the user in entering data points, showing regression curves, and exporting models (as CSV or plots).

When a simulation finishes, the computed Vp and Vs are compared to the calibration; if a significant mismatch is found, the user can iteratively adjust parameters or rely on the system's auto-correction. In practice, calibration ensures that the simulated Vp/Vs ratio matches lab data for that material. The code even displays the current simulated Vp/Vs target in the dialog and updates SimulationResults.VpVsRatio if needed.

8.5 A* Pathfinding for Transducers

Defining an optimal travel path between a transmitter (TX) and receiver (RX) inside a heterogeneous volume is done via a 3D A* search (Hart et al., 1968)[25]. The user selects TX/RX voxel coordinates (e.g. by clicking the volume view), and picks which material ID the wave travels through (typically one contiguous region). CTS then calls AStar.FindPath(tx,ty,tz, rx,ry,rz). The A* algorithm treats each voxel as a node and uses 6-connected neighbors (±1 step in x, y, or z). It restricts the search to voxels matching the chosen material ID (so paths stay inside that region).

The cost of moving to a neighbor is the Euclidean distance between voxel centers: in code,

```
double movementCost = Math.Sqrt(dx*dx + dy*dy + dz*dz);
newG = current.GCost + movementCost;

```

[25] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum-cost paths. IEEE Transactions on Systems Science and Cybernetics, 4(2), 100-107.

with dx,dy,dz in $\{-1,0,1\}$. Since diagonal neighbors are not used (dx,dy,dz has only one nonzero at a time), the movement cost is 1 for axial moves. The heuristic is the straight-line (Euclidean) distance to the goal:

$$h = \sqrt{(x_{\text{goal}} - x_{\text{curr}})^2 + \cdots}$$

This finds the shortest path (and hence shortest travel time if speed is uniform) between TX and RX in the voxel grid. After completion, FindPath returns a list of Point3D points from start to goal; CTS then simplifies collinear points and logs key waypoints.

In the simulation form (AcousticSimulationForm), clicking "Calculate TX-RX Path" initiates this search: it constructs an AStar pathfinder = new AStar(labels, materialID), runs FindPath, and then stores the resulting point list. If either TX or RX lies outside the material, the code warns the user and instead creates a straight-line path for display. When a path is found, the GUI draws it over the volume and also computes its length for travel-time calculation (see below).

8.6 Simulation Architecture

The acoustic simulation is implemented in C# with two parallel execution paths: CPU and GPU. The AcousticSimulator class is the CPU implementation; AcousticSimulatorGPU uses ILGPU to run on the GPU (or multi-core CPU fallback). Both versions perform identical physics and inputs. The main form allows toggling "Run on GPU" for large models; otherwise the CPU code is used.

8.7 CPU Simulator (AcousticSimulator.cs)

The CPU simulator uses 3D double arrays for state (velocities and stresses) sized [width,height,depth]. For example:

```
private readonly double[,,] vx, vy, vz;
private readonly double[,,] sxx, syy, szz, sxy, sxz, syz;
private readonly double[,,] damage;

```

(allocated in the constructor). The time step loop increments stepCount until termination. Updates are done either serially or with Parallel.For over one dimension (z) to exploit multithreading (as seen in UpdateStress()). The loop alternates calls to UpdateStress and UpdateVelocity (not shown), each iterating over the 3D grid and computing finite differences.

The Courant condition determines dt:

```
dt = Math.Min(SafetyCourant * pixelSize / vpMax, dtFreq);

```

where vpMax is the maximum P-wave speed in the model and $dtFreq$ is set by the excitation frequency. SafetyCourant is ~0.4 for numerical stability. This ensures dt (in seconds) satisfies the CFL condition (P.D. Lax, 1967[26]; Courant-Friedrichs-Levy, 1928[27]). (A lower bound of $1e-8\,s$ is enforced to avoid underflow.) The code logs the final dt value for verification.

The simulator also tracks when the wave reaches the receiver (RX). At each timestep, it checks the receiver voxel's velocity components: when a P-peak is detected in vx it marks the P-wave arrival (pWaveTouchStep), and similarly for S-wave in vy/vz. After both are detected (plus a few extra steps), the simulation terminates. It logs:

```
P-velocity={pVelocity:F2} m/s, S-velocity={sVelocity:F2} m/s, Vp/Vs={vpVsRatio:F3}
Travel times: P-wave={pWaveTouchStep}, S-wave={sWaveTouchStep}, Total={stepCount}

```

with the computed speeds $v_p = \frac{\text{distance}}{pWaveTime\cdot dt}$.

Finally, it raises a SimulationCompleted event carrying the results (PWaveVelocity, SWaveVelocity, travel times, etc.).

8.8 GPU Simulator (AcousticSimulatorGPU.cs)

The GPU pipeline uses ILGPU to compile and launch kernels. In its constructor, a GPU context/accelerator is created:

```
context = Context.Create(builder => builder
    .Cuda()     // use CUDA if available
    .OpenCL()); // fallback to OpenCL

```

On this accelerator, the simulator allocates device buffers for the grid (material IDs, density, vx,vy,vz, stresses, damage). For example, MemoryBuffer1D<double> is created for each field. The code then loads two primary kernels: stressKernel and velocityKernel, each a stream kernel over all grid cells.

[26] P.D.Lax, 1967, Hyperbolic Difference Equations: A Review of the Courant-Friedrichs-Lewy Paper in the Light of Recent Developments, IBM Journal

[27] R. Courant, K.Friedrichs, H.Lewy, 1928, On Partial Difference Equations of Mathematical Physics, IBM Journal

These kernels mirror the CPU loops: they read neighbors' values from global memory and update stresses/velocities.

During simulation, at each time step the host invokes:

```
stressKernel(totalCells, materialBuffer.View, densityBuffer.View,
    vxBuffer.View, vyBuffer.View, vzBuffer.View,
    sxxBuffer.View, syyBuffer.View, szzBuffer.View,
    sxyBuffer.View, sxzBuffer.View, syzBuffer.View,
    damageBuffer.View, physParams, gridParams);
velocityKernel(totalCells, materialBuffer.View, densityBuffer.View,
    vxBuffer.View, vyBuffer.View, vzBuffer.View,
    sxxBuffer.View, syyBuffer.View, szzBuffer.View,
    sxyBuffer.View, sxzBuffer.View, syzBuffer.View,
    damageBuffer.View, physParams, gridParams);

```

ILGPU transparently launches these on the GPU (or multi-threaded CPU) with the given parameters. This yields massive parallelism: all voxels update concurrently. The GPU version includes the same physics flags (UsePlastic, UseBrittle in PhysicsParams) and stabilization. Memory transfers occur for the results and for any progress visualization.

Performance-wise, the GPU engine is recommended for large volumes (e.g. >10M voxels) where CPU becomes slow. The ILGPU context will automatically use the best available device. However, be aware of GPU memory limits: each double[,,] array is $\sim 8 \times W \times H \times D\ bytes$, and multiple buffers (9 stress + 3 velocity + others) multiply this footprint. If GPU memory is insufficient, ILGPU may fall back to CPU execution.

8.9 Visualization and Result Extraction

CTS provides real-time and post-processing visualization of the wavefield and results. During simulation (both CPU and GPU), the module periodically fires ProgressUpdated events with partial wavefields (floats) for the P- and S-wave fields (used to update the GUI). Once complete, the full time histories and final snapshots are available in an SimulationResults object.

The GUI "Results" tab displays:

- Waveforms: Plots of wave amplitude vs time at the midpoint and receiver. Midpoint data (e.g. center of TX-RX path) and receiver velocities are captured at each time step. The code logs these as "[MidpointMeasurement] P=…, S=…, Z=…" for debugging. The AcousticResults code uses the cached fields (cachedPWaveField, etc.) and travel times to draw pulse waveforms. It also shows the dead time (the time between P- and S-arrivals).

- Cross-sectional wavefields: The time-evolving 3D pressure (P-wave) and shear fields can be visualized as slices. In practice, CTS generates a 3D volume of the final velocity fields (vx, vy, vz) at receiver arrival. The Visualizer (via AcousticSimulationVisualizer) can display 2D cross-sections and 3D renderings. Users can scrub through X/Y/Z slices to see wave propagation inside the volume. A trackbar selects which slice is shown in the waveform.
- Velocity tomography & path profile: After computing travel times, CTS can invert for velocity along the A* path (simple tomography). It computes the average Vp and Vs along the path and shows a profile plot (distance vs velocity) in the results panel. Additionally, a histogram of voxel velocities (from calibration or input) is shown.

All displays allow export (PNG) or CSV data. The code also updates summary labels: P-wave speed (m/s), S-wave speed, Vp/Vs ratio, P- and S-wave travel times (steps and ms). For example, after simulation completion it sets:

```
simulationResults = new SimulationResults {
    PWaveVelocity = pWaveVel, SWaveVelocity = sWaveVel,
    VpVsRatio = vpVsRatio, PWaveTravelTime = e.PWaveTravelTime,
    SWaveTravelTime = e.SWaveTravelTime
};

```

These values are then formatted into the GUI labels. The code multiplies travel steps by dt to get seconds, converting to milliseconds for display.

Finally, AcousticVolume export: the AcousticVolume object (with vertices, indices, normals) is prepared for external export (e.g. to OBJ or STL) if needed. Its Vertices list holds mesh vertices, Indices holds triangle indices, and Normals holds per-vertex normals. The VoxelDensities dictionary maps voxel indices to density values (for colored export). This structure is filled after meshing the CT volume (not shown) and can be written out for CAD or FEM tools.

8.10 Wave Travel Time and Vp/Vs Results

Travel times are measured at the RX location by detecting when the propagating wave first significantly excites the receiver. In practice, after each timestep the simulator checks the receiver voxel's velocity. When a P-wave pulse arrives, pWaveTouchStep is set to the current step; similarly for S-wave. The travel times (in timesteps) are then converted to real time by multiplying by dt (e.g. if $dt = 1.0e - 7\ s$ and 50,000 steps, travel time ≈5 ms). The distance between TX and RX is taken as the straight-line path length (in $voxels \times pixelSize$). Finally, the wave speeds are computed as:

$$V_p = \frac{\text{dist}}{T_p \cdot dt}, \quad V_s = \frac{\text{dist}}{T_s \cdot dt}$$

The Vp/Vs ratio is simply $\frac{V_p}{V_s}$. These results are communicated via the AcousticSimulationCompleteEventArgs (with fields PWaveVelocity, SWaveVelocity, VpVsRatio, and travel times).

In the CTS GUI, these numeric results are shown to the user and can be exported to CSV or XML. The travel time (in steps and ms) is also plotted on the waveform panel. For example, the label text is set as "<N> steps (X.XXX ms)" using the formula $time_{ms} = steps * dt * 1000$. The dead time (difference between S and P arrival) is similarly computed and displayed.

To keep the scheme numerically stable and avoid energy buildup the code applies a light, artificial “viscous” damping at every timestep.
Inside each **VelocityUpdate** routine (CPU and GPU) a constant:

```
const double DAMPING_FACTOR = 0.05;   // 5 % per-step

```

is defined. Before the new stress-derived increment is added, the existing velocity in every component is premultiplied by

```
damping = 1 − DAMPING_FACTOR = 0.95

```

$$v_{new} = v_{old} \times \mathbf{0.95} + \Delta v$$

so, each step removes about 5 % of the current amplitude. This produces a first-order exponential decay with $time - constant \approx \frac{dt}{0.05}$, which is strong enough to suppress grid-scale ringing and runaway growth, yet weak enough that it does **not** measurably alter the arrival times or the resulting Vp , Vs and Vp/Vs ratios.

8.11 AcousticVolume Meshing and Normals

The AcousticVolume class (in AcousticVolume.cs) holds the geometric mesh used for exporting the CT volume. It contains:

```
public List<Vector3> Vertices { get; set; } = new List<Vector3>();
public List<int> Indices  { get; set; } = new List<int>();
public List<Vector3> Normals { get; set; } = new List<Vector3>();
public Dictionary<int,double> VoxelDensities { get; set; } = new Dictionary<int,double>();
public byte[,,] VolumeData { get; set; }
public Vector3 MinBounds, MaxBounds;

```

Here, Vertices/Indices/Normals define a triangular surface mesh of the volume (e.g. for a cropped material region). Each vertex's normal is stored in Normals. The Indices list specifies triangles by indexing into Vertices. VoxelDensities maps a voxel index (flattened) to its density value, enabling per-element density tagging in the mesh. After simulation, this mesh (with density tags) can be exported to standard formats for further analysis or for meshing in finite-element packages.

The meshing process itself uses a surface extraction algorithm. AcousticVolume provides the final structured geometry: MinBounds and MaxBounds record the physical extents of the mesh, and all data are ready for rendering or exporting.

8.12 Units and Scaling

All physical quantities are in SI units. PixelSize (CT voxel edge length) is input in meters, so that distances (e.g. TX–RX separation) are scaled correctly. Elastic moduli entered in MPa are internally multiplied by 10^6 to convert to Pa. Time stepping dt is in seconds, computed from the Courant condition. When displaying or exporting results, CTS converts time steps to seconds (and often milliseconds) by multiplying by dt. For example, waveform plots and result labels show milliseconds, using the formula $time\_ms = steps * dt * 1000$.

The code also contains scaling factors for visualization: a constant WAVE_VISUALIZATION_AMPLIFICATION ($1e10$) is used when rendering the wave amplitude for GUI display. This is purely cosmetic to bring small velocity perturbations into view (actual simulation values remain in physical units). On export, however, the raw physical values are written.

All code snippets above are taken from the CTS source, illustrating the algorithmic implementation.

Chapter 9: Using the Acoustic Simulator

This chapter is a practical, step-by-step guide to running acoustic simulations in CTS. It assumes the user has already loaded or segmented a CT dataset and is ready to simulate wave propagation through it. Unlike Chapter 8, which focused on the physics and code, this section is entirely user-facing and based on GUI interaction.

9.1 Getting Started

To begin:

1. Load your dataset: Open a .bin project file or import a segmented CT volume using ControlForm → File → Import .bin.
2. Ensure labels are defined: Your CT volume should have at least one material label (e.g., “Limestone”, “Fracture”).
3. Launch the Acoustic Simulator: Navigate to ControlForm → Tools → Acoustic Simulator.

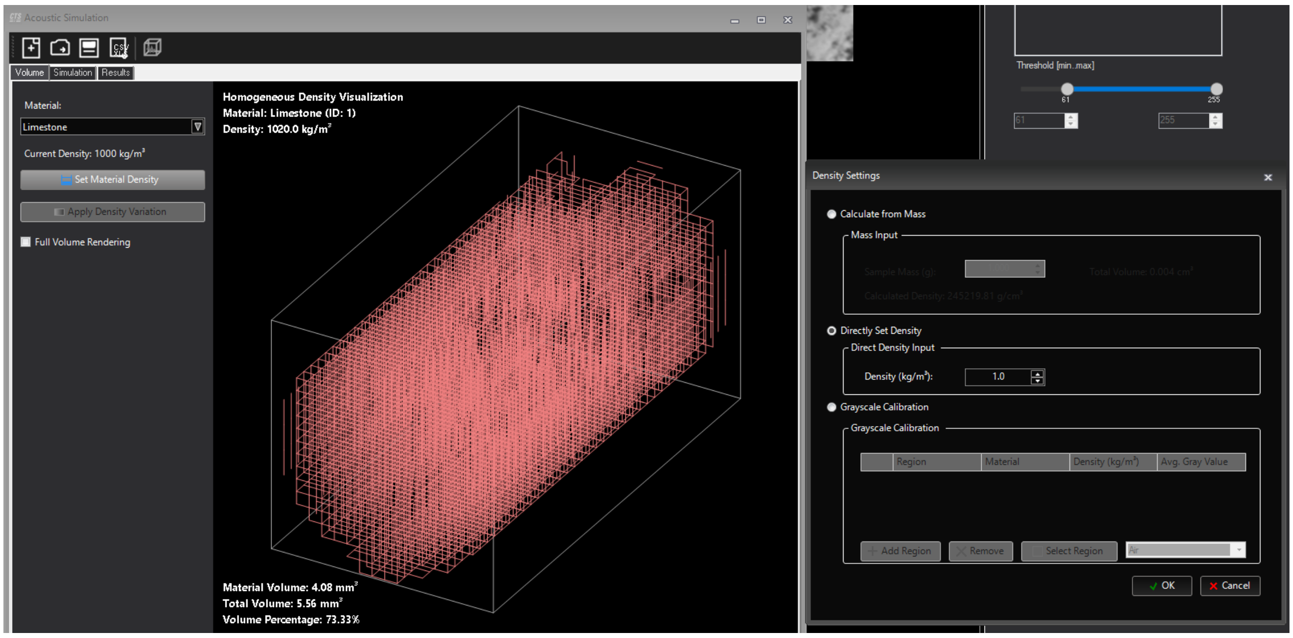
This will open the AcousticSimulationForm, which provides controls for defining the simulation, visualizing the wave propagation, and extracting results.

9.2 Defining the Simulation Setup

A. **Select Material to Simulate and calibrate density**

At the top-left of the simulation window:

- Choose the material label (by name or ID) where wave propagation will occur.
- Chose “Set Material Density” (Figure 9) and give the material its density, based on:
  - Sample weight (the volume is calculated by CTS);
  - Assigning mean density value;
  - Calibrating with an XY slice.
- Once density calibration is applied click on Apply Density Variation to assign a density value to every voxel of the volume.
- Only one material can be active at a time (e.g., simulate wave propagation in “Sandstone”).



Material: Limestone

Current Density: 1000 kg/m³

Homogeneous Density Visualization
Material: Limestone (ID: 1)
Density: 1020.0 kg/m³

Material Volume: 4.08 mm³
Total Volume: 5.56 mm³
Volume Percentage: 73.33%

Threshold [min..max]: 61 – 255

Calculate from Mass
Mass Input
Sample Mass (g): 1.000
Total Volume: 0.004 cm³
Calculated Density: 245219.81 g/cm³

[✓ Directly Set Density]
Direct Density Input
Density (kg/m³): 1.0

Grayscale Calibration

	Region	Material	Density (kg/m³)	Avg. Gray Value

Air

Figure 9 – Density calibration window

B. **Define TX and RX Manually (Transmitter and Receiver) (FUTURE UPDATE)**

1. Click “Set TX”, then click in one of the 2D slice views to place the transmitter.
2. Click “Set RX”, then click a different location to place the receiver.
3. The system will highlight both points in all orthogonal views.

Tip: TX and RX must lie within the selected material region. If either is outside, you will be warned.

At the current stage CTS allows for testing along a direction placing RX and TX automatically at the material boundaries. The “Calculate TX-RX Path” button will ensure the materials to be in the material and will position them at the best locations.

C. **Activate Physical Models (Optional)**

Check the boxes for:

- Elastic: simulate purely elastic (default).
- Plastic: simulate permanent deformation when stresses exceed yield.
- Brittle: simulate sudden failure if tensile limits are exceeded.

If unsure, use only Elastic.

D. **Choose Simulation Mode**

At the bottom:

- Check Run on GPU if your system has CUDA/OpenCL support.
- Otherwise, CTS will use a multi-threaded CPU simulation.
- Chose the transceiver contact type: CTS is able to simulate point emission or full face TX-RX in order to simulate laboratory procedures where normally a full face transducer is used.

9.3 Setting Simulation Parameters

Click “Advanced Settings” (or “Simulation Parameters”) to configure:

- Wave Frequency (kHz): Typical ultrasonic testing uses 50–500 kHz.
- Time Step Count: How many steps the simulation runs.
- Energy (J): The energy of the wave emitted by the TX.
- Amplitude: The max amplitude the wave can reach (mainly for visualization)

The following conditions can also be specified:

- Confining Pressure (MPa)
- Tensile Strenght of the material (MPa)
- Failure Angle (degree)
- Cohesion (MPa)

Some of this properties can be calculated using other simulations on CTS (for example the triaxial module).

Apply Calibration (Optional)

If you've used the calibration system, click “Apply Calibration” to autofill Young’s modulus and Poisson’s ratio based on your material’s density or Vp/Vs ratio.

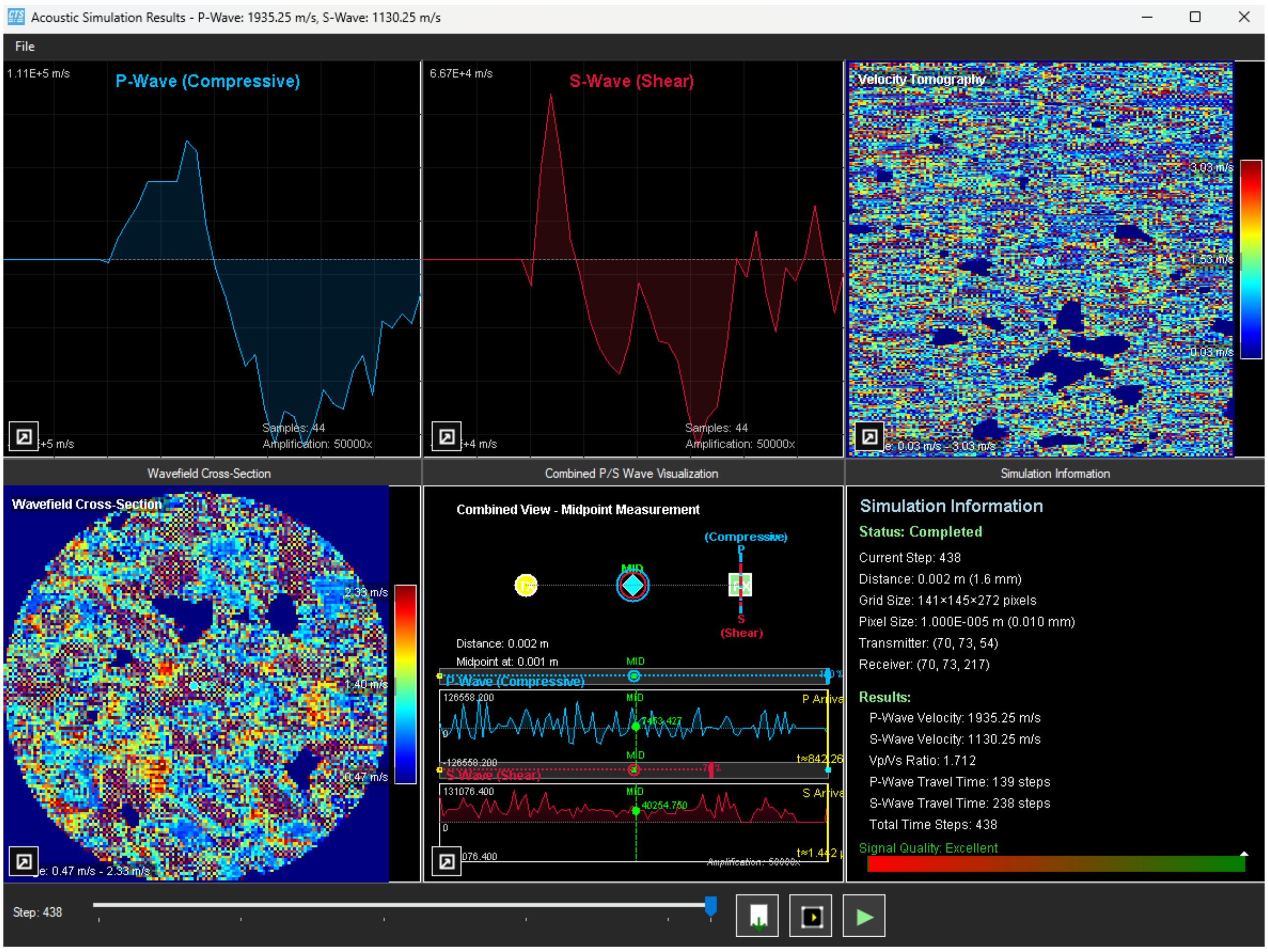
The simulation can be cached to a local folder, in order to allow a full FPS animation export afterwards or the analysis along all the steps. Enabling the cache will raise the simulation time due to the frequent CPU/GPU transfers. To maximize speed, disable the visualizer and caching. If a simulation starts with caching enabled the software will try to identify previous saved caches and asks if the user wants to start over or continue the simulation. A simulation cache can take many GB of space as every voxel update is stored to disk in order to reconstruct every step.

9.4 Running the Simulation

Click Start Simulation. The following occurs:

- The interface shows a progress bar and current time step.
- A 3D viewer updates in real time with the evolving wavefield.
- The simulation halts automatically after reaching the receiver and recording wave arrival.

Depending on dataset size and system specs, simulations can last from seconds (GPU) to minutes (CPU). An interactive visualizer will show the waveform in realtime (if enabled) and will also show a velocity tomography along the simulation path and a cross section wavefield tomography (Figure 10). The same viewer allows for animation exporting (if caching is enabled).



Acoustic Simulation Results - P-Wave: 1935.25 m/s, S-Wave: 1130.25 m/s

Simulation Information
Status: Completed
Current Step: 438
Distance: 0.002 m (1.6 mm)
Grid Size: 141×145×272 pixels
Pixel Size: 1.000E-005 m (0.010 mm)
Transmitter: (70, 73, 54)
Receiver: (70, 73, 217)

Results:
P-Wave Velocity: 1935.25 m/s
S-Wave Velocity: 1130.25 m/s
Vp/Vs Ratio: 1.712
P-Wave Travel Time: 139 steps
S-Wave Travel Time: 238 steps
Total Time Steps: 438
Signal Quality: Excellent

Figure 10 – Acoustic Velocity Simulation visualization window.

9.5 Viewing the Results

Once the simulation finishes, three result tabs become available:

A. **Waveform Analysis**

- Shows P-wave and S-wave arrival as time-series plots (Figure 11).
- Highlights the dead time (time between P and S wave).
- Allows zooming and toggling waveform types.

B. **3D Tomography**

- Visualizes wavefronts inside the CT volume.
- Supports slice views (X/Y/Z), zoom, pan, and rotation.
- Adjust opacity, color scale, and slice index.

C. **Detailed Results**

Shows calculated:

- P-wave velocity (Vp) in m/s
- S-wave velocity (Vs)
- Vp/Vs ratio
- Travel time of each wave in milliseconds
- Distance between TX and RX
- Material statistics, including average density

Click Recalculate if you update simulation settings and want to recompute values.

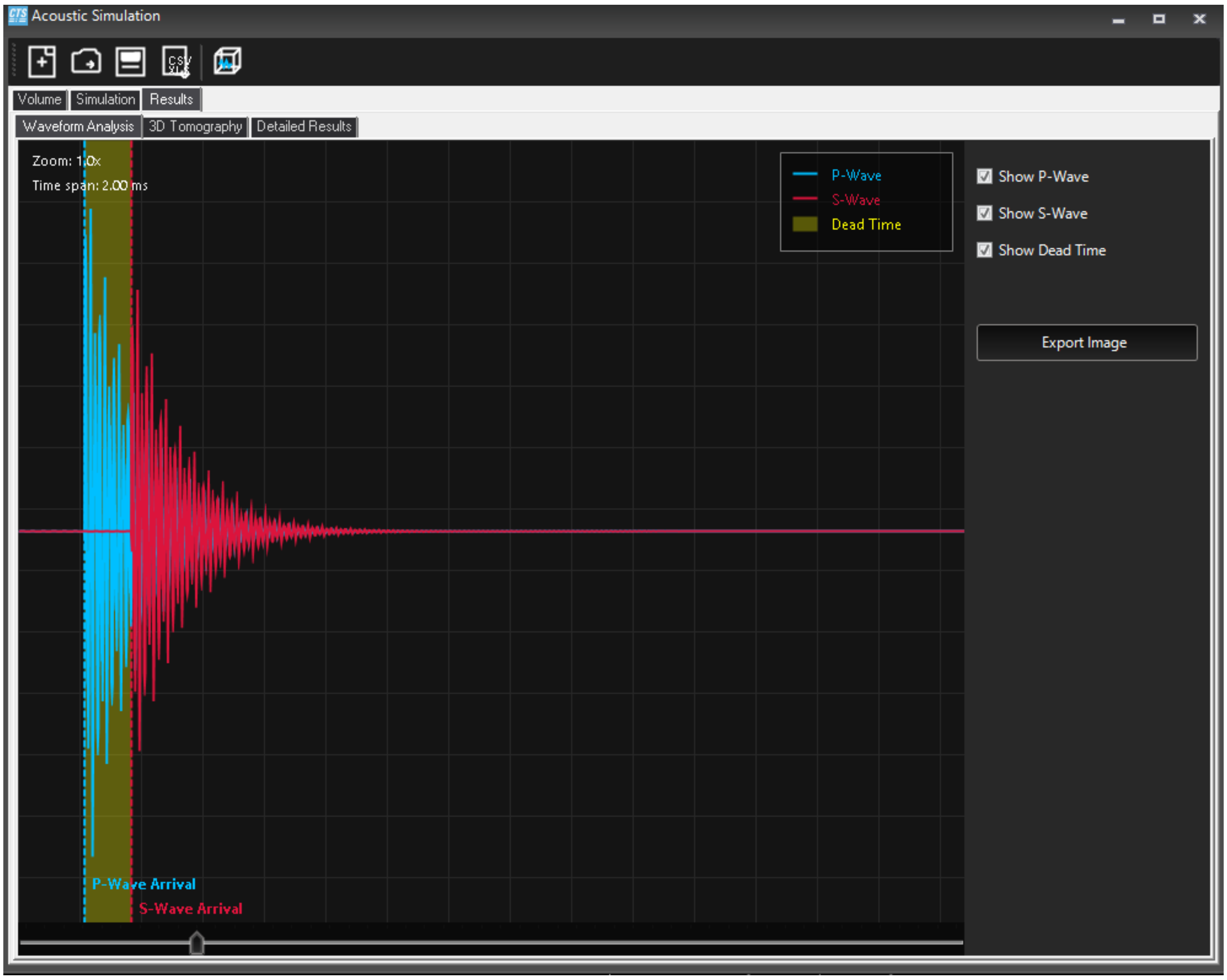


Figure 11 – Time series visualization

9.6 Exporting and Saving

At the bottom of each results tab:

- Export Waveform: Save the amplitude vs time plot as PNG or CSV.
- Export Tomography: Save current 2D/3D views.
- Export Details: Export numeric results as CSV or Excel.
- Export Composite: Save a composite figure showing waveform, tomography, and key results.

You can also export the AcousticVolume mesh (from the main menu) to use in external modeling software.

9.7 Tips and Recommendations

High-accuracy results	Use GPU + Elastic + Fine voxel size
Faster test run	CPU + small subvolume + low resolution
Surface excitation	Place TX near outer surface
Measuring anisotropy	Run multiple TX-RX paths in different directions
Suspected fractures	Enable Brittle flag, observe stress peaks
Comparing materials	Run calibration and apply same TX/RX setup

Common Issues

- No arrival detected at RX? → Check TX and RX are in the same material and close enough.
- Unrealistic Vp/Vs ratio? → Apply calibration or adjust material stiffness.

The Acoustic Simulator in CTS allows researchers to:

- Simulate ultrasonic wave propagation in real CT data
- Quantify velocities, wave arrival times, and material behavior
- Visually analyze wave travel and stress fields
- Calibrate materials to real-world experiments

No other open-source platform currently offers this integrated, CT-driven acoustic simulation pipeline. CTS combines real imaging data with real physics for a new generation of material characterization.

Chapter 10: Triaxial Simulation Framework

The CTS triaxial module simulates axial loading of a CT-scanned core under constant confining pressure. The core volume is represented by a volumetric mesh, and material points follow **elastic–plastic–brittle** constitutive behavior. Initially each element deforms elastically ($\sigma = E\varepsilon$) up to a yield stress. Beyond yielding, a hardening (plastic) law applies additional strain with diminishing stiffness. Ultimately a brittle failure limit causes stress to drop toward a residual value. The Mohr–Coulomb criterion governs shear failure: using the principal stresses $\sigma_1 \geq \sigma_3$, failure occurs when

$$\sigma_1 - \sigma_3 \geq 2c\, cos\, \phi + (\sigma_1 + \sigma_3)\, sin\, \phi,$$

where c is cohesion and ϕ the friction angle. In practice CTS checks this condition and, if violated, marks the element as failed and assigns a residual strength based on $\frac{1-sin\,\phi}{1+sin\,\phi}$relationships. Tensile cutoff is also applied via $\sigma_3 < -\frac{c}{tan\,\phi}$ (Jaeger et al., 2007)[14]. Graphically, Mohr's circle (normal vs. shear stress) is used to visualize stress state relative to the linear failure envelope.

Elastic and plastic deformation generate both shear and volumetric strains. The *volumetric strain* in the triaxial test (axial strain ε_a plus lateral strains) is calculated as

$$\varepsilon_{\text{vol}} = \varepsilon_a + 2(-\nu\varepsilon_a) = \varepsilon_a(1 - 2\nu),$$

where ν is Poisson's ratio. This bulk change is tracked at each load step, enabling calculation of pore-pressure evolution and volume changes.

CTS implements **poroelasticity** via Biot's effective stress principle. Fluid-saturated rock is modeled so that pore pressure p reduces the effective confining stress. The Biot coefficient α (typically 0.5–1.0) weights the pore pressure:

$$\sigma' = \sigma - \alpha p.$$

In saturated cores α may be assumed unity; otherwise, it is set by rock compressibility (Biot, 1941)[28]. In CTS $\alpha = 1 - K/K_s$ is computed using the grain (solid) bulk modulus K_s. For example, using $K_s \approx 36 - 45$ GPa for quartz-rich rock, $\alpha = 1 - K/K_s$ (clamped 0.5–1.0). The code computes pore pressure change from volumetric strain via Skempton's coefficient

[28] Biot, M. A. (1941). General theory of three-dimensional consolidation. Journal of Applied Physics, 12(2), 155–164.

$$B = \frac{1}{1 + \frac{\phi K_f}{K}}, \quad \phi = \text{porosity}$$

and a drainage factor. In an undrained simulation, the pore pressure increment

$\varDelta p = -BK_{\text{bulk}}\varepsilon_{\text{vol}}$ (positive for dilation). Effective confining pressure is then

$\sigma_c^{'} = \sigma_c - \alpha p$, which enters the yield check and stress update.

Real-time tracking: At each load step, CTS updates each element's stress state and computes global quantities. The mean stress and deviator (differential stress) are recorded to assemble stress–strain response and p–q diagrams. Fractional failure is tallied (ratio of failed elements). Volumetric strain is tracked via the formula above, enabling calculation of volumetric changes and pore pressure via Skempton's law. Energy is accumulated separately as elastic (½$\sigma \cdot \varepsilon$) and plastic work.

Compute architecture: The engine uses data-parallel loops over mesh elements. On CPU, multi-threading (Parallel.For) splits the per-element stress integration across cores. Optionally, a compatible GPU can be used: the code checks for supported OpenCL/CUDA devices and uploads element data (densities, stiffness) to GPU buffers for massively parallel strain updates. Whether on CPU or GPU, the key steps are identical: (1) compute each element's yield and brittle strains from its Young's modulus and strengths; (2) compute pore pressure and effective stress; (3) apply axial strain (and lateral strains via υ to get trial stresses; (4) add plastic hardening if beyond yield, then enforce brittle decay beyond peak; (5) check Mohr–Coulomb failure, applying residual strength if needed; (6) update stresses and failure flags. This produces a strain-localized map of damage as "failed" elements.

Material parameters: The simulation requires input values of key parameters:

- *Young's modulus* E: sets the initial stiffness ($\text{stress} = E\varepsilon$) and elastic energy; calibrated by the initial slope of stress–strain.
- *Poisson's ratio* ν: controls lateral vs. axial coupling, and enters the volumetric strain formula. High ν leads to greater volume change in triaxial loading.
- *Cohesion* c and *friction angle* ϕ: define the Mohr–Coulomb envelope. Larger c and ϕ raise the failure envelope (higher shear strength).

- *Biot's coefficient* α: determines how pore pressure reduces effective stress. $\alpha \approx 1$for high-porosity rock, approaching 0 for very stiff grains.
- *Drained Bulk Modulus* K: enters poroelasticity (Skempton's coefficient) and sets magnitude of pore pressure per volumetric strain.

Each parameter influences the simulation E and ν control elastic response; c, ϕ control plastic failure; α, K couple to pore pressure; and *brittle strength* sets ultimate failure limit. Heterogeneity (e.g. density variations from CT) is captured by scaling these parameters per element.

Calibration pipeline: CTS provides tools to derive these parameters from laboratory data. Typically, one runs a simulated triaxial test and fits the response: (1) the initial linear portion of the stress–strain curve gives E via linear regression. (2) Poisson's ratio can be derived from lateral vs. axial strains or estimated by type. (3) Yield strength is identified by the 0.2% offset method or stress plateau. (4) Peak strength sets the brittle limit. (5) Finally, Mohr–Coulomb parameters c, ϕ are obtained by fitting the failure envelope to multiple tests at different confinements: solving $(\sigma_1 - \sigma_3) = 2c \cos\phi + (\sigma_1 + \sigma_3)\sin\phi$. For a single test, one can back-calculate $\phi = \arcsin\left(\frac{\tau_f}{p_f}\right)$ and c from σ_1, σ_3 at failure.

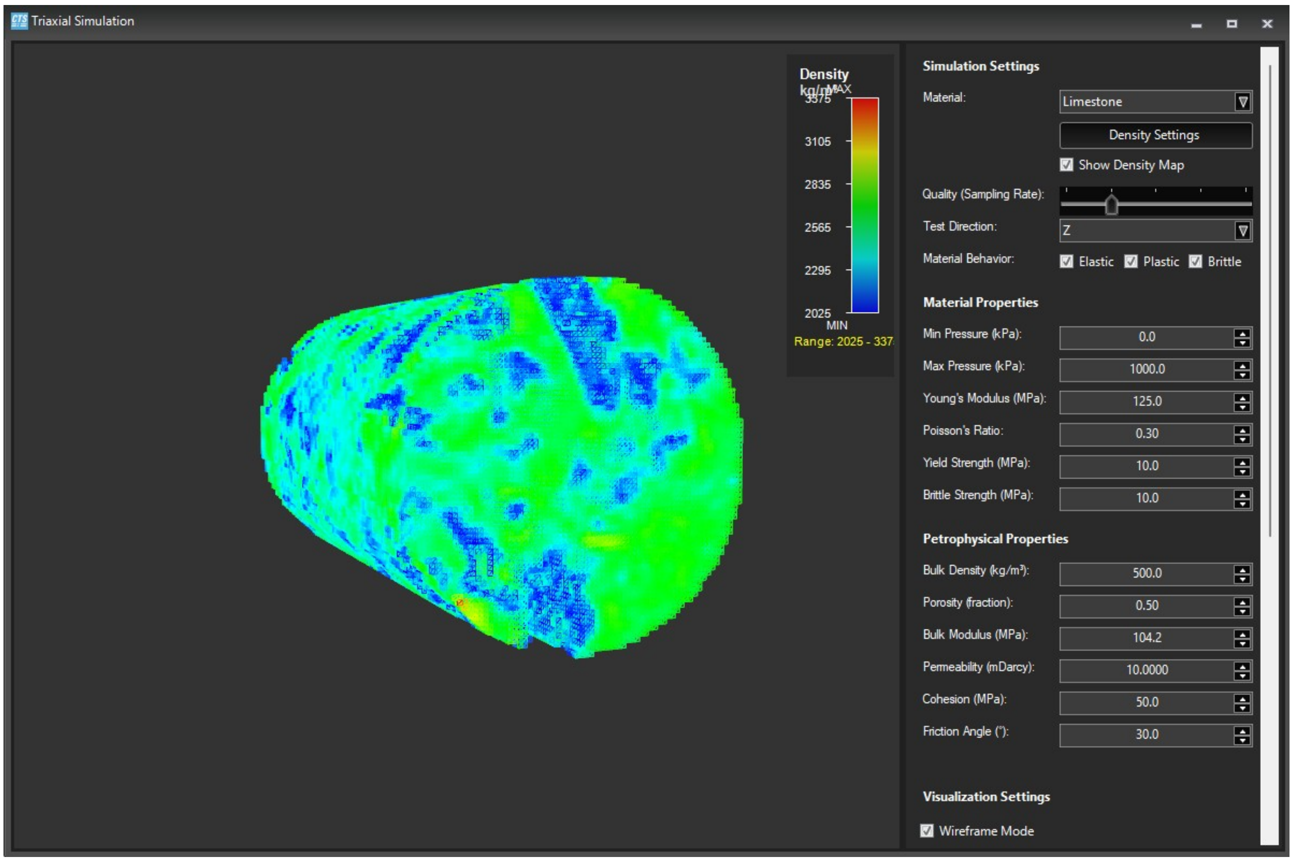
These calibrated parameters are loaded into the simulator, ensuring that CTS reproduces laboratory triaxial behavior.

Visual analysis: CTS includes real-time visualization of stress states. A Mohr's circle diagram plots σ_1 vs. σ_3 or shear vs. normal stress for the current state, with the Coulomb line for reference. Failure "maps" or masks are generated by marking failed elements in slices of the core volume, revealing shear bands and fracture networks. As the simulation progresses, one can trace fracture propagation through the core by animating these failure masks. Color maps of volumetric strain or effective pressure can also be overlaid on the 3D mesh to highlight compaction/dilation zones.

Data export: After simulation, CTS can export results in several formats. The deformed 3D mesh (vertices and element indices) can be saved or converted to a voxel volume, preserving the stress and failure state for each element. Scalar fields (e.g. element stress, strain or permeability) can be written as volume datasets for external analysis. The stress–strain history, pore-pressure, volume change, and failure fraction are exported numerically (CSV/Excel) for plotting and further processing. Fracture "slices" (binary masks of failed elements in cross-sections) may be saved as images or volumes to overlay on the CT scan. Together, these outputs allow detailed post-mortem analysis of the simulated test.

Chapter 11: Using the Triaxial Simulator

To begin, launch the CTS Triaxial Simulator from the main CTS application (e.g. via **Modules → Triaxial Test**). Load the CT-derived volume or imported mesh of the core sample. The GUI displays the 3D core and material properties panel. **Defining material parameters** (Figure 12)**:** You can either enter the calibrated mechanical parameters manually (Young's modulus E, Poisson ratio v, cohesion c, friction angle ϕ, tensile/brittle strength, porosity, etc.) or press *Calibrate* to apply values obtained from a previous calibration routine. If fluid effects are modeled, also specify Biot's coefficient (or porosity/grain modulus to compute it) and fluid bulk modulus. The parameters should match the intended test conditions (e.g. dry vs. saturated).



Simulation Settings

Material:	Limestone
	✓ Show Density Map
Quality (Sampling Rate):	
Test Direction:	Z
Material Behavior:	✓ Elastic ✓ Plastic ✓ Brittle

Material Properties

Min Pressure (kPa):	0.0
Max Pressure (kPa):	1000.0
Young's Modulus (MPa):	125.0
Poisson's Ratio:	0.30
Yield Strength (MPa):	10.0
Brittle Strength (MPa):	10.0

Petrophysical Properties

Bulk Density (kg/m³):	500.0
Porosity (fraction):	0.50
Bulk Modulus (MPa):	104.2
Permeability (mDarcy):	10.0000
Cohesion (MPa):	50.0
Friction Angle (°):	30.0

Visualization Settings

✓ Wireframe Mode

Figure 12: Triaxial simulator setup after density input and generate mesh button pressed.

Running the simulation: Click *Start*. The simulation proceeds in small strain increments, updating element stresses each step. **Real-time feedback:** A live stress–strain plot appears, showing axial stress vs. axial strain. Concurrently, plots of volumetric strain vs. axial strain and pore pressure (if any) update. The 3D view shows the deformed core: failed elements (where Mohr–Coulomb is met) are highlighted in a contrasting color or with an overlay. A *fracture propagation* animation shows shear bands growing from failure points. The GUI also displays cumulative failure percentage and energy metrics (elastic and plastic work).

Visual overlays and diagnostics: Hovering or selecting regions can display local values of σ_1, σ_3 and stresses. An overlay can draw the **failure plane** within the sample volume corresponding to the optimal shear surface. A color map of volumetric strain (or effective stress) can be toggled on the cross-section. In the *Diagrams* panel, the Mohr–Coulomb envelope is shown along with the current Mohr's circle. A p–q (mean-deviator) plot is also provided: here $p = (\sigma_1 + \sigma_2 + \sigma_3)/3$ vs. $q = (\sigma_1 - \sigma_3)/2$ are plotted for each step, illustrating the stress path relative to the yield surface.

Interpreting results: As strain increases, observe how the stress–strain curve peaks and softens, and at what axial strain significant failure occurs. A plateau or drop indicates strain localization (shear band). The Mohr's circle plot shows if the current stress approaches the envelope. A **linear elastic segment** indicates no yielding; departure from linearity indicates onset of plasticity. Delayed or attenuated volumetric strain curves signal pore-pressure buildup under low permeability (drained vs. undrained behavior).

Exporting data and models: At any point or when the test ends, click *Export Results*. This saves the stress–strain curves, volumetric strain, pore pressure, and failure fraction to CSV or Excel. Use *Export to Volume* to output the deformed mesh as a 3D volume dataset. This can include an intensity field for stress or a binary mask of failed elements. You can also snapshot images of slices showing the fracture network. All exports can be used for external analysis or reports.

Advanced integration: The CTS framework allows coupling with other modules after a triaxial run. For example, export the fractured volume to the acoustic simulator: the GPU-accelerated acoustic module uses the same material moduli and failure pattern to simulate wave propagation, revealing changes in elastic wavefields due to damage. Similarly, a pore-network module can import the updated porosity and fracture mask to compute permeability or fluid flow post-damage (Feature to be completed). By chaining the triaxial simulator with acoustic and flow simulations, researchers can study how mechanical damage impacts transport properties in real time.

Chapter 12: NMR Simulation and Calibration

The CTS NMR module models transverse relaxation (T_2) in porous media based on established NMR physics. In a simple system the transverse magnetization decay exponentially according to $\boldsymbol{M(t)} = \boldsymbol{M_0} \cdot \boldsymbol{exp(-t/T_2)}$. Physically, T_2 is shortened by interactions with pore surfaces: in a water-wet pore $1/T_2 \approx \rho \cdot (S/V)$, so that larger pores yield longer T_2 (since $S/V \propto \frac{1}{radius}$). This surface-relaxation effect complements the Bloch phenomenology and is consistent with the Bloembergen–Purcell–Pound (BPP) theory of relaxation, which links molecular tumbling to relaxation rates (typically $T_1 \geq T_2$) (Bloembergen et al., 1948)[29].

The simulation assigns each voxel a relaxation component based on its material and pore geometry. Material NMR properties include a **mean $\boldsymbol{T_2}$** (relaxation time) and distribution width (via a log-normal sigma) along with **hydrogen density**, **tortuosity factor**, **relaxation strength**, and **porosity-effect** parameters. The code generates a set of discrete T_2 values (log-spaced between $MinT_2$ and $MaxT_2$) for each material. For each voxel, the amplitude of a given T_2 component is computed from the voxel's hydrogen content ($density \times voxel\, porosity$) multiplied by the material's T_2 distribution (a log-normal around the mean T_2) and by pore-geometry effects (smaller pores favor shorter T_2). An *effective tortuosity* (increasing for low connectivity or small pores) further modifies each component's T_2 (shortening it for high tortuosity). In aggregate, this maps voxel porosity, pore size, and density into a simulated T_2 spectrum.

The GPU-accelerated engine then computes the net NMR signal over time in parallel. Each selected time point *t* is processed by summing contributions from all relaxation components:

$M(t) = \sum_k A_k \cdot exp\left(-\frac{t}{T_{2k}}\right)$ where A_k and T_{2k} are the amplitude and relaxation time of component k. This exponential-decay summation is executed on the GPU (DirectCompute shader) for all voxels and time steps simultaneously, yielding a fast simulation of the magnetization decay curve (Bloembergen et al., 1948)[17].

[29] Bloembergen, N., Purcell, E. M., & Pound, R. V. (1948). Relaxation effects in nuclear magnetic resonance absorption. Physical Review, 73(7), 679–712.

After the run, the software outputs a $\boldsymbol{T_2}$ **spectrum** and summary metrics (Figure 13). The T_2 spectrum is given as a distribution of amplitudes vs. T_2 time (i.e. a histogram or spectrum of relaxation components). From this spectrum the system computes:

- **Effective porosity** (the fraction of voxels contributing free fluid signal) – essentially the pore volume fraction with hydrogen signal.
- **Log-mean** $\boldsymbol{T_2}$ (the geometric mean of the T_2 distribution), a standard summary in NMR logging.
- **Total volume** (model volume or sample volume) and **total pore volume** (derived from porosity).

These outputs appear in the results panel along with fitted component lists. All numerical results (decay curve, spectrum, porosity, average T_2, etc.) can be **exported** to text/CSV files. The export includes time vs. magnetization (decay curve) and T_2 vs. amplitude (spectrum) data, as well as a log of simulation parameters and summary (showing total porosity, average T_2, tortuosity, component peaks, and any applied calibration statistics).

Interactive plots allow inspection of results: a decay curve (magnetization vs. time) and a T_2-distribution chart (amplitude vs. T_2) are generated on the fly. The interface supports linear or logarithmic axes and can overlay individual component fits. An overview chart can display component contributions or the "waterfall" of multiple runs. These **visualization modules** let the user verify the simulated signal shape and T_2 spectrum.

Finally, users can calibrate the simulation to real measurements. Calibration points (pairs of simulated and reference T_2/amplitude values) are collected in the **Calibration Manager**. The software fits a transform (typically linear on log–log scale) to map simulated T_2 and amplitudes to lab/field values. When calibration is applied, all simulated components are adjusted by these functions. Calibration metadata (sample conditions, author, etc.) are stored with each calibration set for traceability.

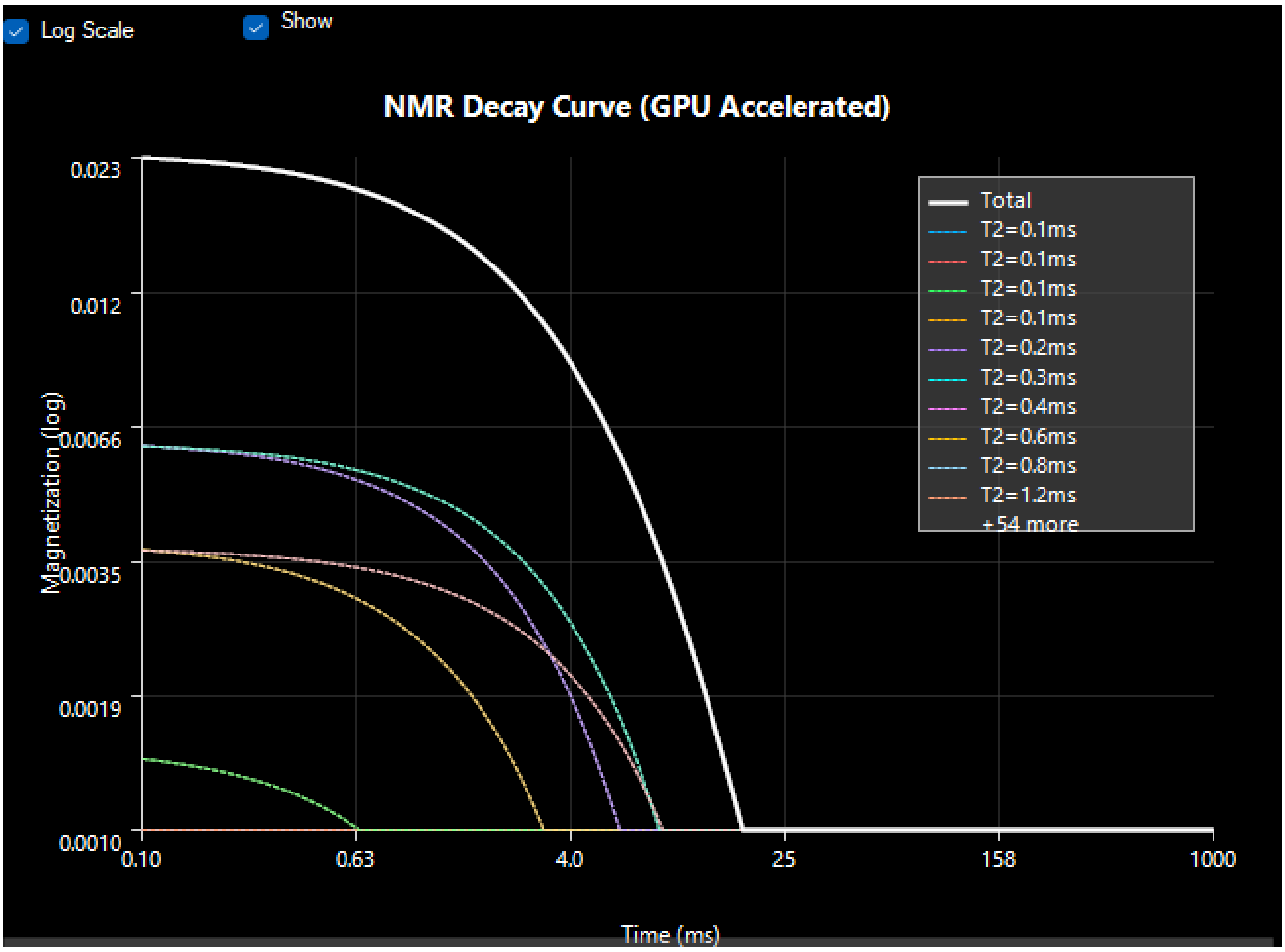


Figure 13 – Example of decay results of an CTS NMR module simulation run (not calibrated).

Update and Bug Reporting System

CTS includes an **auto-updater** that checks for new releases on GitHub. On startup (or when manually invoked via "Check for Updates…"), the program connects securely to the GitHub API (using HTTPS) to fetch the latest release tag and assets. If a newer version is available, the update dialog displays the current and new version numbers and shows the release notes. The user is offered a choice of **Standard** or **Full** update: the standard updater executable (a small patch) or the full installer package. A checkbox or menu option toggles between them, depending on availability.

Once the user confirms, the updater downloads the chosen package over SSL (progress is shown in a progress bar with percent and bytes). Upon completion, the application prompts that an update is ready and will restart. The new updater executable is then launched and the main application exits. The updater performs the installation steps:

- **Elevation and Compatibility:** If the application is installed in a protected location (e.g. Program Files), the updater detects this and relaunches itself with administrator privileges. This ensures it can overwrite files.
- **Extraction:** The downloaded updater package contains the new program files (often appended as a ZIP). The updater extracts these files to a temporary folder.
- **Shutdown:** It then terminates any running instances of CTS (identifying processes by executable name).
- **Installation:** All updated files are copied into the application directory, overwriting old versions.
- **Cleanup:** Temporary files and the update folder are deleted. A small cleanup batch script is scheduled to remove the updater executable itself after the program closes.

Throughout this process, a status form (UpdateProgressForm) shows current actions ("Stopping application…", "Installing update…", etc.) and progress. When done, the application is restarted automatically.

Administrators can also **manually trigger** an update check at any time via a menu item or button. This brings up the same update dialog and process. If the update check fails (for example due to no internet connection), the user is notified by an error dialog and may retry later. As a fallback, users can always download the latest installer from the CTS GitHub releases page when available.

Alongside updating, CTS provides a **bug reporting** form to streamline issue tracking. When the user submits a bug report, the form collects diagnostic information including:

- Application version and build date
- Operating system and hardware environment
- A short summary and detailed description of the problem (entered by the user)
- Any relevant application logs or screenshots (if provided)
- User contact information (optional)

In **online mode** (when internet is available), the report may be sent directly to the CTS issue tracker on GitHub or support server. Typically, the software will open a prefilled GitHub issue or use an API call to create a new issue, attaching logs and details. In **offline mode** (no internet), the form instead packages the report into an email draft or a local file. For example, it might open the default mail client with a message to the support address containing the error details and log excerpt.

If sending fails (network error, or if GitHub rejects the request), the user is informed and given options: they can save the report to disk (a text file or structured log) and manually send it later. The software ensures that critical information (error message, stack trace, context) is preserved in these fallbacks.

CTS is an actively developed software, and new features are continually being designed and integrated. Future updates will not only address bugs and compatibility improvements but also expand the program’s functionality with additional modules. For example, a visual node editor is already planned as part of the development but has not yet been completed. Users can expect upcoming releases to gradually incorporate such enhancements, offering more powerful tools and customization options.