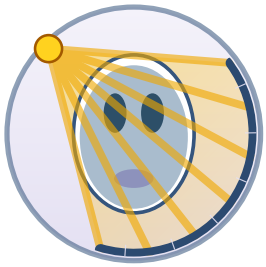
CTSim-7.0.0-User-Manual



Contents

1 Concepts **3**
- 1.1 Conceptual Overview . . . 3
- 1.2 Phantoms . . . 3
  - 1.2.1 Phantom File . . . 3
  - 1.2.2 Phantom Elements . . . 3
  - 1.2.3 Phantom Size . . . 4
- 1.3 Scanner . . . 4
  - 1.3.1 Dimensions . . . 4
  - 1.3.2 Parallel Geometry . . . 5
  - 1.3.3 Divergent Geometries . . . 6
- 1.4 Reconstruction . . . 7
  - 1.4.1 Direct Inverse Fourier . . . 7
  - 1.4.2 Filtered Backprojection . . . 9
- 1.5 Image Comparison . . . 9

2 Installation **10**
- 2.1 Download . . . 10
- 2.2 Installing the Windows package . . . 10
- 2.3 Installing on macOS . . . 10
- 2.4 Installing on Debian and Ubuntu . . . 10
- 2.5 Build From Sources . . . 10
  - 2.5.1 Debian and Ubuntu . . . 11
  - 2.5.2 macOS . . . 11
  - 2.5.3 Windows . . . 12
  - 2.5.4 Libraries . . . 13

3 The Graphical User Interface **13**
- 3.1 Starting CTSim . . . 13
- 3.2 Windows . . . 13
- 3.3 Quick Start . . . 14
- 3.4 File Types . . . 14
  - 3.4.1 Phantom . . . 14
  - 3.4.2 Process - Scale Size, Add, Subtract, Multiply, Divide, Convert to 3-D . . . 14
  - 3.4.3 Projection . . . 15
  - 3.4.4 Plot . . . 15
- 3.5 Global Menu Commands . . . 15
  - 3.5.1 File - Open . . . 15
  - 3.5.2 File - Create Phantom . . . 15
  - 3.5.3 File - Settings . . . 15
  - 3.5.4 File - Save, Save As, Close, Exit . . . 15
  - 3.5.5 Window . . . 15
  - 3.5.6 Help - Manual . . . 15
  - 3.5.7 Help - Tip of the Day . . . 16
  - 3.5.8 Help - About . . . 16
- 3.6 Phantom Windows . . . 16

3.6.1 File - Properties . . . 16
3.6.2 Process - Rasterize . . . 16
3.6.3 Process - Projections . . . 16
3.7 Image Windows . . . 16
3.7.1 File - Properties, Export, Print . . . 16
3.7.2 Edit . . . 16
3.7.3 Process - Filter . . . 16
3.8 Projection Windows . . . 17
3.8.1 File - Properties, Print . . . 17
3.8.2 Edit - Copy . . . 17
3.8.3 Process . . . 17
3.9 Plot Windows . . . 17
3.9.1 File - Properties, Print . . . 17
3.9.2 Edit - Copy . . . 17
3.9.3 View . . . 17
3.10 Text Windows . . . 17
3.11 3-D Windows . . . 18
3.12 Scripts . . . 18
3.12.1 How a script runs . . . 20

4 The Command Line Interface **21**
4.1 Starting ctsim-nogui . . . 21
4.2 Parallel processing . . . 21
4.3 if1 . . . 21
4.4 if2 . . . 21
4.5 ifexport . . . 22
4.6 ifinfo . . . 22
4.7 phm2pj . . . 22
4.8 phm2if . . . 22
4.9 pj2if . . . 22
4.10 pjinfo . . . 22
4.11 pjrec . . . 22
4.12 help and bench . . . 23
4.13 The other functions . . . 23
4.13.1 Helical scans . . . 23
4.14 Odd sizes and counts . . . 23
4.15 File formats . . . 24

5 Algorithms **24**
5.1 Phantom Processing . . . 24
5.2 Background Processing . . . 25
5.2.1 Advantages . . . 25
5.2.2 Disadvantages . . . 26

6 Simple Graphics Package **26**
6.1 Transformation Sequence . . . 26
6.2 Functions . . . 27
6.2.1 Master coordinate functions . . . 27
6.2.2 Normalized coordinate functions . . . 27
6.2.3 Master coordinate to World coordinate transformations . . . 27
6.2.4 Coordinate transformation functions . . . 27
6.2.5 State functions . . . 27
6.3 Coordinate Mapping . . . 27
6.3.1 Dimensions . . . 27

6.3.2 Formulas 27

1 Concepts

1.1 Conceptual Overview

The operation of CTSim begins with the phantom object. A phantom object consists of geometric elements. A scanner is specified and the collection of x-ray data, or projections, is simulated. This projection data can be reconstructed using various user-controlled algorithms producing an image of the phantom object. These reconstructions can be visually and statistically compared to the original phantom object.

In order to use CTSim effectively, some knowledge of how CTSim works and the approach taken is required. CTSim deals with a variety of object, but the two primary objects that we need to be concerned with are the and the .

1.2 Phantoms

CTSim uses geometrical objects to describe the object being scanned. A phantom is composed of one or more phantom elements. These elements are simple geometric shapes, specifically, rectangles, triangles, ellipses, sectors and segments. With these elements, the standard phantoms used in the CT literature can be constructed. In fact, CTSim provides a shortcut to load the published phantoms of Herman[@HERMAN80] and Shepp-Logan[@SHEPP74]. CTSim also reads text files of user-defined phantoms.

The types of phantom elements and their definitions are taken with permission from G.T. Herman's publication[@HERMAN80].

1.2.1 Phantom File

Each line in the text file describes an element of the phantom. Each line contains seven entries, in the following form:

```
element-type cx cy dx dy r a
```

The first entry defines the type of the element, either `rectangle`, `ellipse`, `triangle`, `sector`, or `segment`.

For all phantom elements, `r` is the rotation applied to the object in degrees counterclockwise and `a` is the X-ray attenuation coefficient of the object. Where objects overlap, the attenuations of the overlapped objects are summed.

As opposed to the `r` and `a` fields, the `cx`, `cy`, `dx` and `dy` fields have different meanings depending on the element type.

1.2.2 Phantom Elements

1.2.2.1 ellipse Ellipses use `dx` and `dy` to define the semi-major and semi-minor axis lengths with the center of the ellipse at `(cx,cy)`. Of note, the commonly used phantom described by Shepp and Logan[@SHEPP74] uses only ellipses.

1.2.2.2 rectangle Rectangles use `(cx,cy)` to define the position of the center of the rectangle with respect to the origin. `dx` and `dy` are the half-width and half-height of the rectangle.

1.2.2.3 triangle Triangles are drawn with the center of the base at `(cx,cy)` and a base half-width of `dx` and a height of `dy`. Rotations are then applied about the center of the base.

1.2.2.4 segment Segments are complex. They are the portion of an circle between a chord and the perimeter of the circle. `dy` sets the radius of the circle. Segments start with the center of the chord located at `(0,0)` and the chord horizontal. The half-width of the chord is set by `dx`. The portion of an circle lying below the chord is then added. The imaginary center of this circle is located at `(0,-dy)`. The segment is then rotated by `r` and then translated by `(cx,cy)`.

1.2.2.5 sector Sectors are the like a "pie slice" from a circle. The radius of the circle is set by `dy`. Sectors are defined similarly to segments. In this case, though, a chord is not drawn. Instead, the lines are drawn from the origin of the circle `(0,-dy)` to the points `(-dx,0)` and `(dx,0)`. The perimeter of the circle is then drawn between those two points and lies below the x-axis. The sector is then rotated and translated the same as a segment.

1.2.3 Phantom Size

The overall dimensions of the phantom are increased by 1% above the specified sizes to avoid clipping due to round-off errors from sampling the polygons of the phantom elements. So, if the phantom is defined as a rectangle of size 0.1 by 0.1, the phantom size is 0.101 in each direction.

1.3 Scanner

Understanding the scanning geometry is the most complicated aspect of using CTSim. For real-world CT simulators, this is actually quite simple. The geometry is fixed by the manufacturer during the construction of the scanner and can not be changed. CTSim, being a very flexible simulator, gives tremendous options in setting up the geometry for a scan.

1.3.1 Dimensions

The geometry for a scan starts with the size of the phantom being scanned. This is because CTSim allows for statistical comparisons between the original phantom image and it's reconstructions. Since CT scanners scan a circular area, the first important variable is the diameter of the circle surround the phantom, the *phantom diameter*. Remember, as mentioned above, the phantom dimensions are padded by 1%.

The other important geometry variables for scanning phantoms are the *view diameter*, *scan diameter*, *focal length*, and *center-detector length*. These variables are input into CTSim in terms of ratios rather than absolute values.

1.3.1.1 Phantom Diameter The phantom diameter is automatically calculated by CTSim from the phantom definition. The maximum of the phantom length and height is used to define the square that completely surrounds the phantom. Let p_l be the width and height of this square. The diameter of this boundary box, p_d, is given by the Pythagorean theorem and is

$$p_d = p_l\sqrt{2}$$

CT scanners collect projections around a circle rather than a square. The diameter of this circle is the diameter of the boundary square p_d.

These relationships are diagrammed in the figure showing the scan geometry.

1.3.1.2 View Diameter The *view diameter* is the area that is being processed during scanning of phantoms as well as during rasterization of phantoms. By default, the *view diameter* is set equal to the *phantom diameter*. It may be useful, especially for experimental reasons, to process an area larger (and maybe even smaller) than the phantom. Thus, during rasterization or during projections, CTSim will ask for a *view ratio*, v_r. The *view diameter* is then calculated as

$$v_d = p_d v_r$$

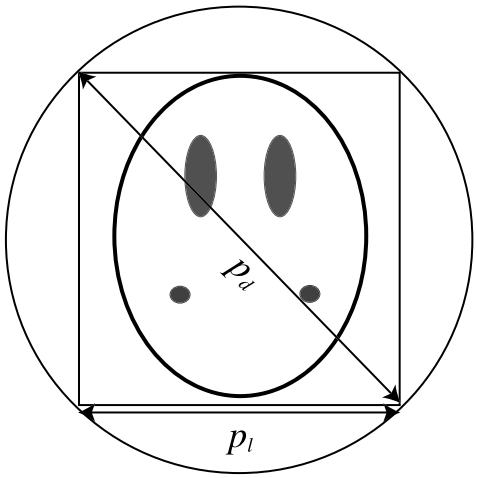


Figure 1: Phantom Geometry

By using a v_r less than 1, CTSim will allow for a *view diameter* less than *phantom diameter*. This will lead to significant artifacts. Physically, this would be impossible and is analogous to inserting an object into the CT scanner that is larger than the scanner itself!

1.3.1.3 Scan Diameter By default, the entire *view diameter* is scanned. For experimental purposes, it may be desirable to scan an area either larger or smaller than the *view diameter*. Thus, the concept of *scan ratio*, s_r, arises. The scan diameter, s_d, is the diameter over which x-rays are collected and is defined as

$$s_d = v_d s_r$$

By default and for all ordinary scanning, the *scan ratio* is to `1`. If the *scan ratio* is less than `1`, you can expect significant artifacts.

1.3.1.4 Focal Length The *focal length*, f, is the distance of the X-ray source to the center of the phantom. The focal length is set as a ratio, f_r, of the view radius. Focal length is calculated as

$$f = (v_d/2) f_r$$

For parallel geometry scanning, the focal length doesn't matter. However, for divergent geometry scanning (equilinear and equiangular), the *focal length ratio* should be set at `2` or more to avoid artifacts. Moreover, a value of less than `1` is physically impossible and it analagous to having the x-ray source inside of the *view diameter*.

1.3.1.5 Center-Detector Length The *center-detector length*, c, is the distance from the center of the phantom to the center of the detector array. The center-detector length is set as a ratio, c_r, of the view radius. The center-detector length is calculated as

$$f = (v_d/2) c_r$$

For parallel geometry scanning, the center-detector length doesn't matter. A value of less than `1` is physically impossible and it analagous to having the detector array inside of the *view diameter*.

1.3.2 Parallel Geometry

The simplest geometry, parallel, was used in first generation scanners. As mentioned above, the focal length is not used in this simple geometry. The detector array is set to be the same size as the *scan diameter*. For

optimal scanning in this geometry, the *scan diameter* should be equal to the *phantom diameter*. This is accomplished by using the default values of `1` for the *view ratio* and the *scan ratio*. If values of less than `1` are used for these two variables, significant distortions will occur.

1.3.3 Divergent Geometries

For both equilinear (second generation) and equiangular (third, fourth, and fifth generation) geometries, the x-ray beams diverge from a single source to a detector array. In the equilinear mode, a single source produces a fan beam which is read by a linear array of detectors. If the detectors occupy an arc of a circle, then the geometry is equiangular. These configurations are shown in the figure showing the divergent geometries.

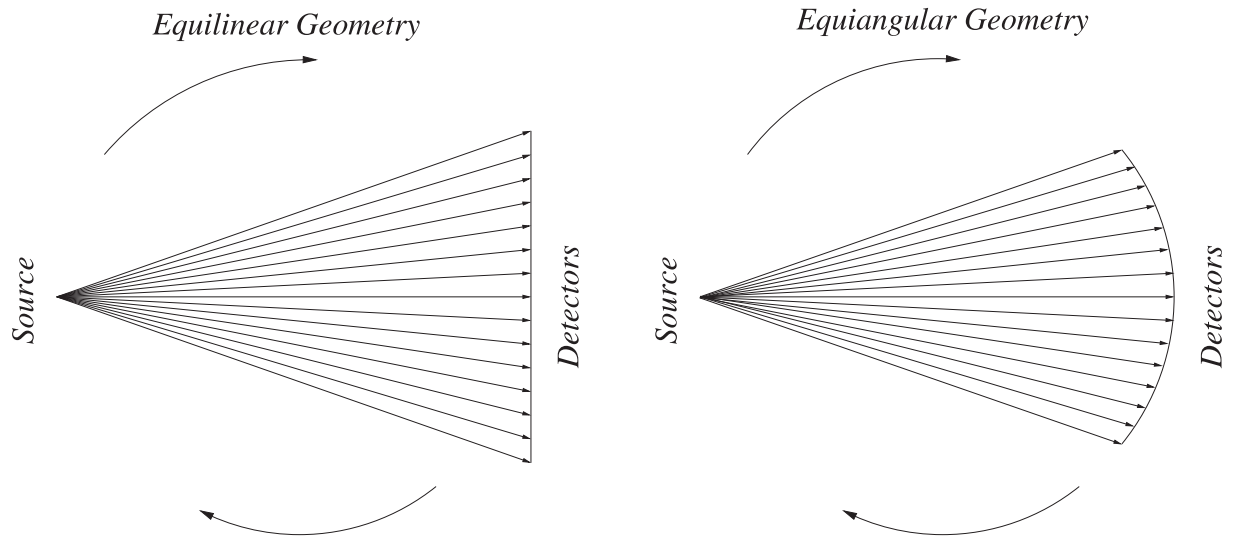


Figure 2: Equilinear and equiangular geometries.

1.3.3.1 Fan Beam Angle For these divergent beam geometries, the *fan beam angle* needs to be calculated. For real-world CT scanners, this is fixed at the time of manufacture. CTSim, however, calculates the *fan beam angle*, α, from the *scan diameter* and the *focal length* as

$$\alpha = 2\sin^{-1}((s_d/2)/f)$$

This is illustrated in the figure showing the fan beam angle.

Empiric testing with CTSim shows that for very large *fan beam angles*, greater than approximately 120°, there are significant artifacts. The primary way to manage the *fan beam angle* is by varying the *focal length* since the *scan diameter* is usually fixed at the size of the phantom.

To illustrate, the *scan diameter* can be defined as

$$s_d = s_r v_r p_d$$

Further, the *focal length* can be defined as

$$f = f_r(v_r p_d/2)$$

Substituting these equations into the equation for the fan beam angle, We have,

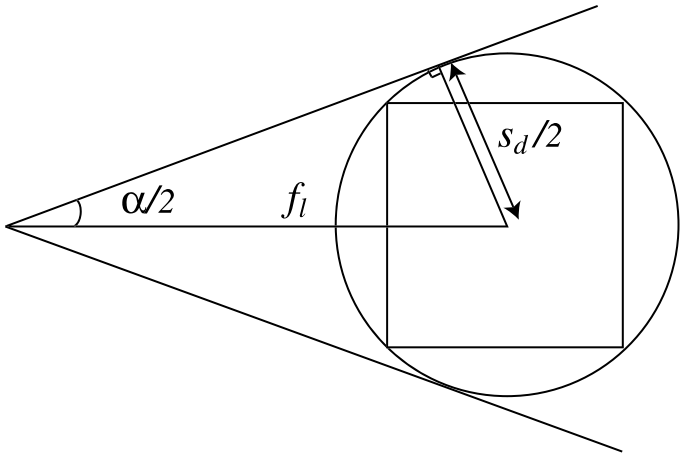


Figure 3: Calculation of α

$$\begin{aligned} \alpha = & \quad 2 \sin^{-1} \frac{s_r v_r p_d/2}{f_r v_r (p_d/2)} \\ = & \quad 2 \sin^{-1}(s_r/f_r) \end{aligned}$$

Since in normal scanning $s_r = 1$, α depends only upon the *focal length ratio* in normal scanning.

1.3.3.2 Detector Array Size In general, you do not need to be concerned with the detector array size – it is automatically calculated by CTSim. For the particularly interested, this section explains how the detector array size is calculated.

For parallel geometry, the detector length is simply the scan diameter.

For divergent beam geometries, the size of the detector array also depends upon the *focal length*: increasing the *focal length* decreases the size of the detector array.

For equiangular geometry, the detectors are equally spaced around a arc covering an angular distance of α as viewed from the source. When viewed from the center of the scanning, the angular distance is

$$\pi + \alpha - 2 \cos^{-1}\left(\frac{s_d/2}{c}\right)$$

The dotted circle in the figure showing equiangular geometry indicates the positions of the detectors in this case.

For equilinear geometry, the detectors are equally spaced along a straight line. The detector length depends upon α and the *focal length*. This length, d_l, is calculated as

$$d_l = 2\,(f + c) \tan(\alpha/2)$$

This geometry is shown in the figure showing equilinear geometry.

1.4 Reconstruction

1.4.1 Direct Inverse Fourier

This method is not currently implemented in CTSim; however, it is planned for a future release. This method does not give results as accurate as filtered backprojection. This is due primarily to interpolation occurring in the frequency domain rather than the spatial domain.

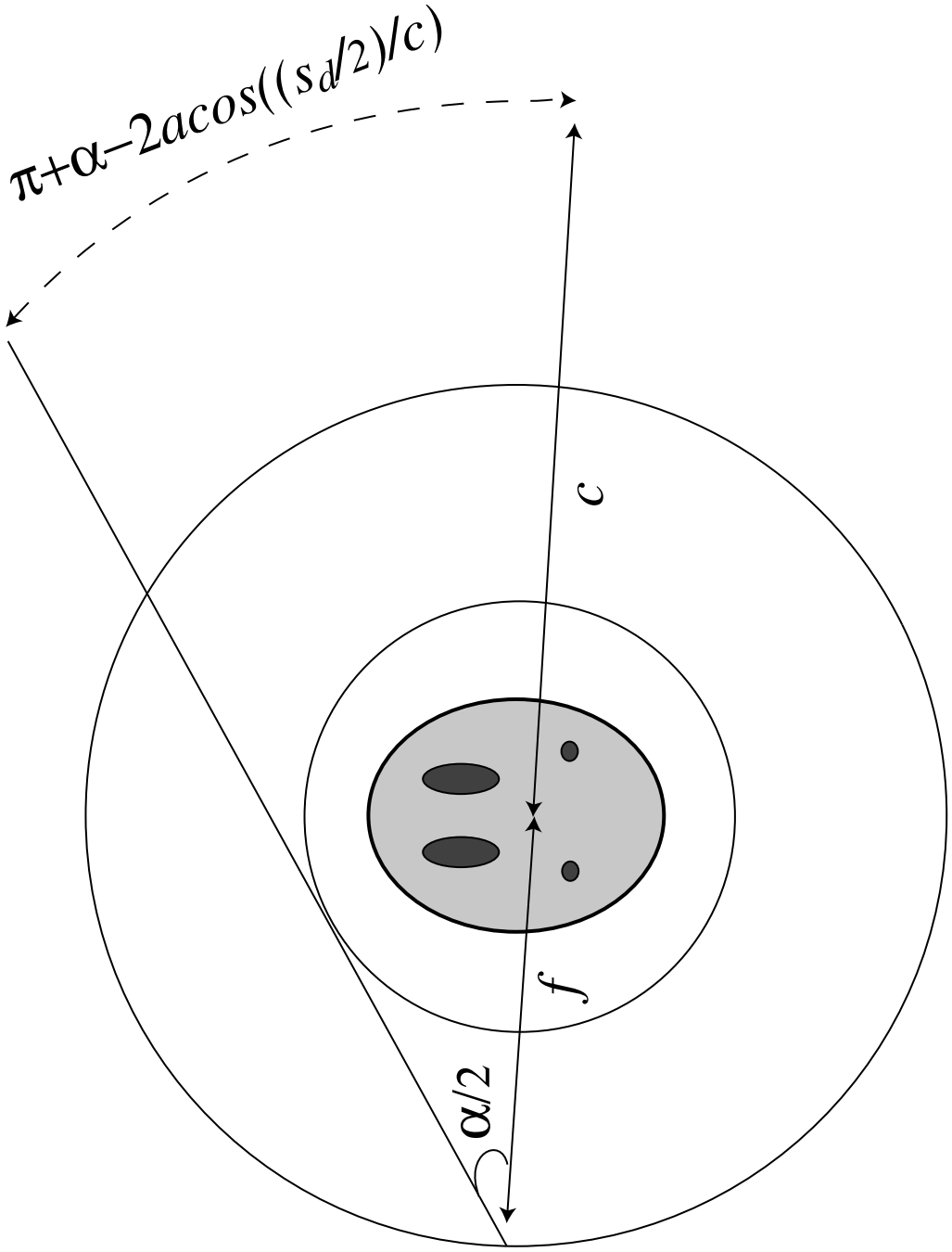


Figure 4: Equiangular geometry

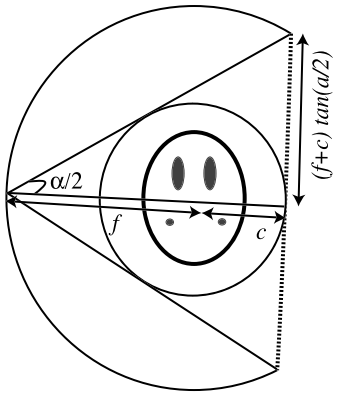


Figure 5: Equilinear geometry

1.4.2 Filtered Backprojection

The technique is comprised of two sequential steps: filtering projections followed by backprojecting the filtered projections. Though these two steps are sequential, each view position can be processed independently.

1.4.2.1 Parallel Computer Processing Since each view can be processed independently, filtered backprojection is amendable to parallel processing. Indeed, this has been used in commercial scanners to speed reconstruction. This parallelism is exploited both in the CTSim graphical shell and in the LAM version of . CTSim can distribute it's workload amongst multiple processors working in parallel.

The graphical shell will automatically take advantage of multiple CPU's when running on a *Symmetric Multiprocessing* computer. Dual-CPU computers are commonly available which provide a near doubling in reconstruction speeds. CTSim, though, has no limits on the number of CPU's that can be used in parallel. The *LAM* version of is designed to work in a cluster of computers. This has been testing with a cluster of 16 computers in a cluster with excellent results.

1.4.2.2 Filter projections The first step in filtered backprojection reconstructions is the filtering of each projection. The projections for a each view have their frequency data multipled by a filter of $|w|$. CTSim permits four different ways to accomplish this filtering.

Two of the methods use convolution of the projection data with the inverse Fourier transform of $|w|$. The other two methods perform an Fourier transform of the projection data and multiply that by the $|w|$ filter and then perform an inverse fourier transform.

Though multiplying by $|w|$ gives the sharpest reconstructions, in practice, superior results are obtained by reducing the higher frequencies. This is performed by mutiplying the $|w|$ filter by another filter that attenuates the higher frequencies. CTSim has multiple filters for this purpose.

1.4.2.3 Backprojection of filtered projections Backprojection is the process of "smearing" the filtered projections over the reconstructing image. Various levels of interpolation can be specified.

1.5 Image Comparison

Images can be compared statistically. Three measurements can be calculated by CTSim. They are taken from the standard measurements used by Herman[@HERMAN80]. They are:

- dThe normalized root mean squared distance measure.
- rThe normalized mean absolute distance measure.
- eThe worst case distance measure over a 2×2 pixel area.

These measurements are defined in the equations for d, r and e. In these equations, p denotes the phantom image, r denotes the reconstruction image, and $\bar{p}$ denotes the average pixel value of p. Each of the images have a size of $m \times n$. In the equation for e $[n/2]$ and $[m/2]$ denote the largest integers less than $n/2$ and $m/2$, respectively.

$$d = \sqrt{\frac{\sum_{i=1}^{n}\sum_{j=1}^{m}(p_{i,j} - r_{i,j})^2}{\sum_{i=1}^{n}\sum_{j=1}^{m}(p_{i,j} - \bar{p})^2}}$$

$$r = \frac{\sum_{i=1}^{n}\sum_{j=1}^{m}|p_{i,j} - r_{i,j}|}{\sum_{i=1}^{n}\sum_{j=1}^{m}|p_{i,j}|}$$

$$e = \max_{\substack{1 \le k \le [n/2] \\ 1 \le l \le [m/2]}}(|P_{k,l} - R_{k,l}|)$$

where

$$P_{k,l} = \tfrac{1}{4}(p_{2k,2l} + p_{2k+1,2l} + p_{2k,2l+l} + p_{2k+1,2l+1})$$
$$R_{k,l} = \tfrac{1}{4}(r_{2k,2l} + r_{2k+1,2l} + r_{2k,2l+1} + r_{2k+1,2l+1})$$

2 Installation

2.1 Download

CTSim is at http://www.ctsim.org, as source and as packages built for Windows, macOS and Debian.

2.2 Installing the Windows package

Run the installer. CTSim is then on the Start menu, and the command-line tools are installed beside it.

CTSim needs Windows 10 or later on a 64-bit machine.

2.3 Installing on macOS

Open the disk image and drag CTSim to Applications. The command-line tools are inside the application, at `/Applications/ctsim.app/Contents/MacOS/ctsim-nogui`.

2.4 Installing on Debian and Ubuntu

```
sudo apt install ./ctsim_*.deb ./ctsim-doc_*.deb
```

`ctsim` holds the application and the command-line tools; `ctsim-doc` holds this manual, which the Help menu reads from `/usr/share/doc/ctsim`.

2.5 Build From Sources

CTSim builds with CMake 3.22 or later and a C++20 compiler. It needs zlib, libpng, FFTW3, HDF5 and DCMTK; the graphical interface needs Qt 6.4 or later. All are ordinary packages on every platform below, and the section on libraries at the end of this chapter says what each is for.

```
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release
cmake --build build -j
cmake --build build --target manual manual-pdf     # optional, see doc/manual/README.md
ctest --test-dir build
sudo cmake --install build
```

`-DCTSIM_GUI=qt` (the default) builds the graphical application; `none` builds only the command-line tools and needs no Qt.

The wxWidgets interface was removed in 7.0.0, after the Qt one reached parity with it.

2.5.1 Debian and Ubuntu

Ubuntu 24.04, 26.04 and Debian 13 (trixie) or later:

```
sudo apt install build-essential cmake pkg-config \
    libfftw3-dev libhdf5-dev libpng-dev zlib1g-dev libedit-dev libgl-dev \
    libdcmtk-dev qt6-base-dev qt6-base-dev-tools qt6-svg-dev
sudo apt install pandoc texlive-latex-recommended lmodern librsvg2-bin   # for the manual
```

To build the Debian packages instead of installing from the tree:

```
sudo apt install debhelper devscripts
dpkg-buildpackage -us -uc -b
sudo apt install ../ctsim_*.deb ../ctsim-doc_*.deb
```

`ctsim-doc` carries the manual under `/usr/share/doc/ctsim`, where the program's Help menu finds it. The same packaging builds unchanged on Ubuntu 24.04 and 26.04 – both were built and their test suites run before release – and nothing in it is specific to a release, so Debian 13 and later should need no change either.

2.5.2 macOS

With Homebrew:

```
xcode-select --install                              # Apple clang, once
brew install cmake pkg-config qt fftw hdf5 libpng dcmtk libedit libomp
brew install pandoc librsvg basictex                # for the manual, optional
cmake -S . -B build -DCMAKE_BUILD_TYPE=Release \
    -DCMAKE_PREFIX_PATH="$(brew --prefix qt);$(brew --prefix libomp)"
cmake --build build -j
ctest --test-dir build
```

Build with Apple's clang, which is what `xcode-select --install` provides and what CMake picks by default. Do not set `CMAKE_CXX_COMPILER` to a Homebrew GCC: Homebrew's Qt and DCMTK are built with clang, and mixing the two compilers gives errors inside the system and Qt headers that have nothing to do with CTSim. OpenMP needs the separate `libomp` package, which is why it is on the `brew install` line and the prefix path.

The build produces `build/ctsim.app`, a bundle: there is no plain `build/ctsim` file to look for. Run it with `open build/ctsim.app`, or `build/ctsim.app/Contents/MacOS/ctsim` from a terminal.

`libomp` matters more than it looks. Apple's clang supports OpenMP but ships neither the runtime nor the headers, so without that package the reconstruction and projection loops run on a single core. CMake finds it through `brew --prefix libomp`; the configure summary reports `OpenMP=TRUE` when it has.

Installing. A Mac application lives in `/Applications`, not under `/usr/local`, and the two halves of CTSim go to different places:

```
cmake --install build --component Runtime --prefix /Applications   # ctsim.app
sudo cmake --install build --component Tools                        # ctsim-nogui, man pages
sudo cmake --install build --component Documentation                # the manual
```

The first needs no `sudo` if `/Applications` is yours. Installing the `Runtime` component also runs Qt's deployment step, which copies the Qt frameworks and plugins the application uses into the bundle so it runs on a Mac without Qt installed.

A disk image. `cmake --install` never makes one; that is `cpack`'s job, and it does its own staging, so it does not depend on having installed first:

```
cpack --config build/CPackConfig.cmake -G DragNDrop
```

produces `build/ctsim-7.0.0-Darwin.dmg` with `ctsim.app` at its root, ready to drag to `/Applications`. The image and cpack's own staging directory are written into the build directory wherever cpack is run from.

The bundle is self-contained: the Qt frameworks, the manual that Help opens (in `Contents/Resources/manual`), `ctsim-nogui` with every command-line tool (in `Contents/MacOS`), and their manual pages (in `Contents/Resources/man`). Build the manual before packaging, or the image will have an application whose Help menu finds nothing:

```
cmake --build build --target manual manual-pdf
cpack --config build/CPackConfig.cmake -G DragNDrop
```

The tools are inside the bundle rather than in `/usr/local/bin`, because a disk image can only ask to be dragged somewhere; nothing in it can install to a second location. To use them from a terminal, link the one program on to your PATH – the image carries a note saying so:

```
sudo ln -s /Applications/ctsim.app/Contents/MacOS/ctsim-nogui \
           /usr/local/bin/ctsim-nogui
export MANPATH="/Applications/ctsim.app/Contents/Resources/man:$MANPATH"
```

Installing from the source tree instead (the three `cmake --install` commands above) puts the tools in `/usr/local/bin` and their pages in `/usr/local/share/man` in the usual way.

Starting it from a terminal. `open ctsim.app` hands the application to macOS to launch, and macOS gives it `/` as its working directory – there is no way for the application to learn where you were. CTSim therefore opens its file dialogs in the directory you last chose one in, rather than at the root of the volume. To have it start where you are instead, pass the directory:

```
open build/ctsim.app --args --cwd "$PWD"
```

or run the program inside the bundle, which inherits the directory as any other program does:

```
build/ctsim.app/Contents/MacOS/ctsim
```

A shell function makes either the normal way to start it:

```
ctsim() { open /Applications/ctsim.app --args --cwd "$PWD" "$@"; }
```

The 3-D view uses OpenGL, which macOS provides; nothing extra is needed.

2.5.3 Windows

The straightforward route is MSYS2 with its UCRT64 environment, which packages everything:

```
pacman -S mingw-w64-ucrt-x86_64-toolchain mingw-w64-ucrt-x86_64-cmake \
    mingw-w64-ucrt-x86_64-ninja mingw-w64-ucrt-x86_64-qt6-base \
    mingw-w64-ucrt-x86_64-qt6-svg mingw-w64-ucrt-x86_64-fftw \
    mingw-w64-ucrt-x86_64-hdf5 mingw-w64-ucrt-x86_64-libpng \
    mingw-w64-ucrt-x86_64-zlib mingw-w64-ucrt-x86_64-dcmtk
cmake -S . -B build -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build build
ctest --test-dir build
```

Then gather the Qt libraries next to the executable for running outside MSYS2:

```
windeployqt build/ctsim.exe
```

Visual Studio also works, with the same libraries from vcpkg (`vcpkg install qtbase qtsvg fftw3 hdf5 libpng zlib dcmtk`) and `-DCMAKE_TOOLCHAIN_FILE=<vcpkg>/scripts/buildsystems/vcpkg.cmake` on the configure line. `libedit` is Unix-only and is left out automatically; the `ctsim-nogui` shell then runs without line editing.

On Windows the settings file is `%USERPROFILE%\.ctsim.ini` and the manual is looked for in a `manual` directory beside the executable.

2.5.4 Libraries

CTSim needs these, and the configure step stops with the name of the package to install if one is missing.

- **FFTW3** (double precision)\ Fourier transforms of projections and images, used by the filtering and by direct Fourier reconstruction.
- **libpng** and **zlib**\ Reading and writing PNG images, and reading a phantom from one.
- **DCMTK**\ Reading and writing DICOM, so that CTSim can import a scan from a scanner and export a reconstruction to one. This was optional until 7.0.0, when the alternative was a DICOM library no distribution packaged; DCMTK is now everywhere, and reading a scanner's own files is too much of what CTSim is for to be left out of a build.
- **HDF5**\ The format CTSim writes by default. `-DCTSIM_WITH_HDF5=OFF` falls back to the historical format, which CTSim reads and writes either way.
- **Qt 6** and **OpenGL**\ The graphical application. Not needed with `-DCTSIM_GUI=none`, which builds the command-line tools alone.

These are genuinely optional:

- **readline** or **libedit**\ Command history and line editing in the `ctsim-nogui` shell. `-DCTSIM_USE_READLINE=OFF` builds without it; the shell still works.
- **OpenMP**\ Rasterization, projection and backprojection run on several cores. `-DCTSIM_USE_OPENMP=OFF` builds a single-threaded program that produces identical results.
- **pandoc**, **pdflatex** and **rsvg-convert**\ Building this manual. Where pandoc is absent, a manual committed to `artifacts/manual` is copied instead, so a package can be built without them.

3 The Graphical User Interface

CTSim is the graphical shell for the CTSim project. This shell uses the library for cross-platform compatibility. The graphical shell is compatible with Microsoft Windows, , and graphical environments.

3.1 Starting CTSim

```
ctsim [files to open...]
```

CTSim can be started on its own or with any number of files to open – images (`.if`), projections (`.pj`), plots (`.plt`) and phantoms (`.phm`). Files can also be dragged from a file manager onto any CTSim window.

3.2 Windows

CTSim 7 is a set of independent windows rather than one large window with smaller ones inside it.

The log window opens first. It is small, and lists what has been opened, created, saved and closed, with the details of each operation – the scan that made a projection set, the parameters of a reconstruction, how long

it took. Closing the log window quits CTSim, after asking about any unsaved windows. Its File menu opens files and creates phantoms; its Window menu lists every open window and brings any of them to the front.

Document windows, one per image, projection set, plot or phantom, are placed by the desktop and have their own menu bars. Each has the shared File, Window and Help menus, an Edit menu where copying applies, and menus for the document's own operations: Process on all of them, View on images and plots. On an image window the filters are under `Process - Filter`, since each of them makes a new image from the one on screen. Properties, Export and Print are on the File menu. The window keeps the proportions of what it shows when it is resized, so enlarging it enlarges the picture rather than adding a margin.

Long operations – rasterizing a phantom, taking projections, reconstructing – run in the background. A Jobs panel appears under the log while anything is running, with a progress bar and a Cancel button for each, and the result opens in its own window when it finishes. Choosing a trace level in the operation's dialog runs it in the foreground instead, drawn a view at a time in a window with Run, Pause, Step and a speed control.

3.3 Quick Start

The fastest way to put CTSim through its basic operation is:

1. `File - Create Phantom...`
   Choose the `Herman` head phantom. A window opens showing it in outline.
2. `Process - Rasterize...`
   On the phantom window. This converts the geometric definition into a raster image, which opens in a new window.
3. `View - Display Scale Auto...`
   On the new image window. This optimizes the intensity scale for display.
4. `Process - Projections...`
   On the phantom window. This simulates the collection of x-ray data. Pick a trace level to watch the scan happen; the projections open as a sinogram in their own window either way.
5. `Process - Filtered Backprojection...`
   On the projection window. This reconstructs an image from the projections, in the background; the reconstruction opens in a new window when it finishes.
6. `Process - Compare Images...`
   On the reconstructed image, choosing the rasterized image from step 2. This measures how well the reconstruction matches the original; the result opens as a text window, and a difference image if asked for.

3.4 File Types

3.4.1 Phantom

Besides loading phantom files from the disk, the Herman[@HERMAN80] and Shepp-Logan[@SHEPP74] phantoms are built-in to CTSim. Phantom files can be read from and written to the disk. Phantom files are stored in a simple ASCII format. A text editor is required to create and edit these files.

3.4.2 Process - Scale Size, Add, Subtract, Multiply, Divide, Convert to 3-D

Image files contain 2-dimensional arrays that store 4-byte floating point values. Images files can be either real or complex-valued. Typically, all images are real-valued except for images that have been processed by Fourier transforms. As you might expect, complex-valued images are twice the size of real-valued images since both a real and imaginary component need to be stored. When complex-valued images are viewed on the screen, only the real component is displayed.

Images files can store any number of text labels. CTSim uses these labels for recording history information regarding the creation and modifications of images.

3.4.3 Projection

Projection files are created from phantom files via the projection process. Numerous options are available for the creation of the these files. The files are stored in a binary format with cross-platform compatibility on little and big-endian architectures.

3.4.4 Plot

Plot files are created by CTSim during analysis of image files. They can be read from and written to the disk. They are stored as ASCII files for easy cross-platform support and editing.

3.5 Global Menu Commands

These are on every window's File, Window and Help menus.

3.5.1 File - Open

Opens image (`.if`), projection (`.pj`), plot (`.plt`) and phantom (`.phm`) files, and PNG images. The dialog starts in the directory you last chose a file in, so a session stays where its work is. Several files can be chosen at once, and recently opened files are listed under `File - Open Recent`. A PNG opens as an image with its pixels as grey levels from 0 to 1 – a colour image by luminance – and its properties show the PNG's own metadata: colour type, bit depth, gamma, resolution and any text the file carries. It has no CTSim file until it is saved.

3.5.2 File - Create Phantom

Creates a phantom either from one of the built-in definitions – the Herman head, the Shepp-Logan head, and a unit pulse – or from a PNG image. A phantom from an image is one *raster* element: the image centred on the origin, one world unit per pixel, with black at attenuation 0 and the brightest value the file can hold at 1. A colour image is converted by luminance, and a transparent area counts as nothing there – attenuation 0 – so a phantom can be drawn on a transparent background. Every PNG form is read: 8 or 16 bits per channel, grey or colour, with or without transparency. The `ctsim-nogui png2phm` command makes the same phantom file from the command line.

3.5.3 File - Settings

Opens the settings dialog (`Ctrl+,`). Text size, for every window; a Light, Dark or System theme; whether generated windows count as unsaved and prompt before closing; whether long operations run in the background; whether a tip is shown at start-up; the plot text size, and whether it scales with the window; and verbose logging. Changes apply at once; Cancel puts them back. Settings are stored in `~/.ctsim.ini` (`%USERPROFILE%\.ctsim.ini` on Windows), or the file `CTSIM_SETTINGS` names.

3.5.4 File - Save, Save As, Close, Exit

Save writes a document to its file, or asks for one; Save As always asks. Files are written in HDF5 unless the legacy format is chosen. Close closes the window, asking first if the document is unsaved; Exit closes every window and quits, asking about each unsaved document – Cancel at any of them leaves everything open.

3.5.5 Window

Lists the log window and every document window; choosing one brings it to the front.

3.5.6 Help - Manual

Opens the manual in the web browser, from `/usr/share/doc/ctsim` or the directory the package installed it to; `Help - Manual (PDF)` opens the PDF instead, and copes with the PDF having been compressed. Set `CTSIM_DOCDIR` to point at a copy elsewhere.

3.5.7 Help - Tip of the Day

Shows a tip, with Next for another. The checkbox controls whether one is shown at start-up, as does the setting.

3.5.8 Help - About

Shows the version and licence.

3.6 Phantom Windows

A phantom window shows the phantom's elements in outline, and a raster element as its image. The window keeps the phantom's proportions.

3.6.1 File - Properties

Lists the elements and their parameters, and the phantom's history. Copy puts the text on the clipboard.

3.6.2 Process - Rasterize

Converts the phantom to an image (`Ctrl+R`): the image size, the number of samples per pixel, and the view ratio. Runs in the background; the image opens in a new window when done, with the phantom's history and the rasterization recorded.

3.6.3 Process - Projections

Simulates the x-ray scan (`Ctrl+J`): detectors and views, samples per ray, geometry (parallel, equilinear or equiangular), rotation angle, focal length and centre-detector ratios, view and scan ratios, and the trace level. With no trace the scan runs in the background; with a trace it is drawn view by view in a window with Run, Pause, Step, Finish Now and a speed control. The projections open as a sinogram in their own window either way.

3.7 Image Windows

An image window shows the image at its intensity scale, fitted to the window, which keeps the image's proportions. The status bar reports the pixel under the pointer and its value; the row and column markers follow the last click and are what the plot and compare commands use.

3.7.1 File - Properties, Export, Print

Properties shows the size and range, the statistics, the history – every operation that made the image, with its parameters, how long it took and when – and, for an image opened from a PNG, the PNG's own metadata. Export writes the image as PNG, PGM or text ({#IDH_DLG_EXPORT}). Print prints it, fitted to the page with a caption.

3.7.2 Edit

Copy puts the image on the clipboard as a picture; Cut copies it and clears the image; Paste replaces the image with the clipboard's picture.

3.7.3 Process - Filter

The filters make a new image from the one on screen, so they are under Process rather than beside it. They divide into three groups.

The point operations – invert, square root, square, log and exp – apply a function to every pixel.

The Fourier operations transform between the image and its frequency representation: FFT and inverse FFT over both dimensions, or over rows or columns alone. An image that has been transformed is complex-valued, and the window shows its real component.

The complex operations – magnitude, phase, real and imaginary – take one component of a complex image as a new real image, and the two shuffles move between natural and Fourier ordering, which is where the zero frequency sits.

3.8 Projection Windows

A projection window shows the sinogram – detectors across, views down – and keeps its proportions. The status bar reports the detector, view and value under the pointer.

3.8.1 File - Properties, Print

Properties shows the scan geometry and the history: the scan that made the projections, its parameters, how long it took and when. Print prints the sinogram.

3.8.2 Edit - Copy

Puts the sinogram on the clipboard as a picture.

3.8.3 Process

One menu, in three sections: the conversions to images (rectangular, polar, FFT polar, and interpolation to parallel geometry); the plots (histogram, T-theta sampling); and the reconstructions.

3.9 Plot Windows

A plot window shows a plot file's curves with EZPlot; the y-axis range can be zoomed, and the range is saved with the plot so it opens the same way.

3.9.1 File - Properties, Print

Properties shows the columns and records, the display range, the plot commands, the descriptions, and the history – for a plot made from an image, the image's own history and then the plotting step; for a comparison of two images, both. Print prints the plot.

3.9.2 Edit - Copy

Puts the plot on the clipboard as a picture.

3.9.3 View

`Display Scale Set` chooses the y-axis minimum and maximum; `Display Scale Auto` chooses them from the data's mean, mode or median and a factor; `Display Full Scale` shows everything. The set range is kept in the plot file and marks it as changed.

3.10 Text Windows

Comparisons and other reports open as text windows. `Edit - Copy` copies the selection, or all of the text if nothing is selected; `Edit - Select All` selects it; `File - Print` prints it.

3.11 3-D Windows

`Process - Convert to 3-D` (`Ctrl+3`) on an image opens it as a surface, coloured by value. Drag to rotate, Shift+drag or the middle button to pan, the wheel to zoom; the arrow keys pan, + and - zoom, Home resets. Sliders under the surface set zoom, turn, tilt and light, with a Reset button; the menu switches wireframe, colour, lighting and smooth shading. The view needs OpenGL and says so if the display has none.

3.12 Scripts

`ctsim --script --workdir directory` says where a script writes what it makes, making the directory if need be. Without it a directory named `ctsim-work-` and the time is made under the system temporary directory and removed when CTSim exits. `--script-save` saves every window the script made into that directory when it ends, each under a name made from its title, and copies the script beside them – so a run can be archived whole and repeated.

The filters are named for what they do rather than for who devised them. `abs_bandlimit` is the plain ramp, published by Ramachandran and Lakshminarayanan and often called the Ram-Lak filter; `abs_sinc` is that ramp windowed by a sinc, published by Shepp and Logan and often called the Shepp-Logan filter. The `abs_` prefix marks the forms multiplied by |w|, which is what a reconstruction needs.

`plot-row` and `plot-column` take `compare` with one title or several separated by commas – `plot-column 68 compare nearest,linear,cubic` – and draw a curve for this image and one for each of them, each in its own colour and named in the legend.

`plot-points` draws a scatter of x,y pairs read from a text file, so that anything a tool can print can be drawn: `plot-points linogram-6.xy`. The file holds one pair to a line or several separated by spaces, each pair written `x,y` or `x y`. `symbol` chooses the mark – 1 a cross, 2 a plus, 3 a box, 4 a circle, 5 an error bar and 6, the default, a point – and `every n` marks only each nth pair. `line yes` joins the points in the order the file gives them; the default, `line no`, leaves them as a scatter. `color` takes a colour number and defaults to black, since single-pixel marks in the curve palette's mid-tones are hard to see on white. `x-title` and `y-title` label the axes. Like every named parameter, each takes a value: `line yes`, never `line` alone. The linogram demonstration, `scripts/demos/linogram.ctsim`, is drawn this way from the output of `linogram n d --xy`.

`plot-scale` fixes a plot window's y axis, as `display-scale` does for an image. A column through a reconstruction means more against a fixed range than against one chosen from whatever that column happens to hold.

`measure distance` reports how far apart two images are – the normalized root mean squared distance, the normalized mean absolute distance and the worst 2x2 neighbourhood distance – and records them. `measure intensity` reports what one image holds: its minimum, maximum, range, mean, standard deviation, median and mode, the same statistics the display scaling is chosen from. Both report to seven places. `expect-compare` reports and records the same numbers and then checks them against bounds; `measure` is for a run whose purpose is the number itself.

A `--script-save` run writes a record of itself into the work directory, named for the script and the time it ran – `recon-filters-260917051608.txt` and the same name with `.json`. It lists every open window, the tool that made it, the parameters that tool received, when it ran and how long it took. A reconstruction lists the scan it came from as well as itself, so a result can be traced back to the settings that produced it. Both name the version, the commit, Qt, the platform and the architecture the run was made with. This is what turns a script from a demonstration into an experiment that can be repeated and checked. With no `--workdir` the work directory goes when CTSim exits, so a run nobody wanted to keep leaves nothing behind.

`geometry` writes each window's actual position and size to the log, and names any pair that overlaps. A window does not always get the size a `move` asked for – a view can have a size of its own, and a minimum can be larger – so this reports what happened rather than what was requested. With a title it reports that window alone.

A script ends by leaving CTSim open with its windows. `wait-at-end` asks instead for the older behaviour, where the last `pause` waits for a key: Escape leaves the windows up, any other key quits.

`ctsim --script` exits non-zero if any command in the script fails, so a demonstration that has stopped working is noticed rather than passing silently. `--script-ignore-errors` exits zero regardless, for a script expected to fail somewhere whose remainder is still worth running.

File - Run Script lists the scripts CTSim can find and runs the one you choose. It looks for `scripts/demos` under the working directory first, so that running from a source tree offers the scripts you are editing; failing that, where the platform installs them; failing that, the working directory itself. The list is built each time the menu opens, so a script you have just written is there without restarting.

CTSim can be driven by a script: a list of commands in a text file, run with

```
ctsim --script demonstration.ctsim
```

Two things are done with this. A **demonstration** shows someone what the program does, with captions and pauses, and is what **Help - Run Demonstration** opens. A **sequence test** runs the same commands headlessly and checks the results, which is how `tests/gui-sequence.sh` proves that a phantom scanned and reconstructed through the interface gives the same file as the same pipeline run with `ctsim-nogui`.

A script is one command per line; `#` begins a comment; a quoted argument may contain spaces. Window titles name windows, so `title` is worth using to give a window a name a later command can rely on.

Each operation waits for the one before it. Rasterizing, scanning and reconstructing run as background jobs, and a script does not continue until the job has finished and its window exists, so the commands can be written in the order the work happens.

```
phantom herman                     # a built-in phantom
title phantom                      # name its window
rasterize nx 256 ny 256 nsample 3  # waits for the job
title original
activate phantom
project ndet 367 nview 320 geometry parallel trace plot
title projections
reconstruct nx 256 ny 256 filter abs_bandlimit
title reconstruction
compare original difference
save reconstruction.if
```

The commands are these.

Documents: `open`, `phantom`, `phantom-png`, `filter-image`, `activate`, `title`, `save`, `close`, `close-all`, `cd`, `quit`.

Operations: `rasterize`, `project`, `reconstruct`, `reconstruct-rebin`, `reconstruct-fourier`, `export`, `filter`, `image`, `scale-size`, `plot-row`, `plot-col`, `plot-fft-row`, `plot-fft-col`, `plot-points`, `histogram`, `cut`, `compare`, `compare-row`, `compare-col`, `convert-3d`, `display-scale`. Each takes the parameters its dialog takes, named: `rasterize nx 256 ny 256 nsample 3`. Anything not given keeps the dialog's own default.

Sequencing: `background` before an operation runs it without waiting, and `wait` collects them all.

Assertions, for tests: `expect-window`, `expect-windows`, `expect-image`, `expect-log`, `expect-compare`. The last one reports all three measures – `NormRMS`, `NormMeanAbs` and `Worst2x2` – whatever bounds are given, so a harness can read the numbers. The bounds themselves are still written `d`, `r` and `e`: `expect-compare a b d 0.1`.

Demonstrations: `say` puts a caption along the bottom of the log window, `clear` removes it, `pause` waits for a number of seconds or for a keypress, `speed` sets the delay per view in traces, `snapshot` writes a window to

a PNG file, and `on-error` decides what a failure does – `stop` (the default), `continue`, or `keep`, which stops the script but leaves the windows up.

Placement: `screen` declares the screen size the coordinates were written for – 1920 by 1080 unless said otherwise – and `move` places a window in those coordinates, scaled to whatever screen the script is running on, so a demonstration written on a large monitor still works on a laptop. `arrange` tiles everything without coordinates. On a Wayland desktop the compositor places windows itself and ignores the program; a demonstration that needs a layout is run with `QT_QPA_PLATFORM=xcb`.

Settings: `setting theme light|dark`, `setting text-size`, `setting plot-text-size`, `setting background-jobs on|off`. These last for the run only; whatever was there before is put back when the script ends.

`export <format> <path>` writes the active image in one of the formats `ifexport` offers – `png`, `png16`, `pgm`, `raw`, `text`. The display range is what maps pixel values on to the format's levels, so set it deliberately before exporting: `display-scale 0 1` for an image whose values are attenuations, since reading a PNG back gives 0 for black and 1 for white. Exporting over the full range of the data instead would scale every attenuation by whatever that range happens to be.

`scripts/demos/intro.ctsim` in the source tree is the demonstration the Help menu runs, and `scripts/demos/raster-vs-geometric.ctsim` compares scanning a phantom's geometry with scanning an image made from it; both are worth reading as examples.

3.12.1 How a script runs

Worth knowing, because it explains what a script can and cannot assume.

A script never blocks the program. The runner executes commands one after another until it reaches one that starts something which finishes later – a background job, a trace window, a pause – and then it stops and hands control back to CTSim. It is started again by whatever signal says the thing has finished. Between those moments the application is fully responsive: windows repaint, menus work, and a key can be pressed. That is what lets a demonstration be watched, and what lets *Escape* stop one.

Waiting is automatic. `rasterize`, `project` and `reconstruct` run in the background, and the next command does not begin until the work has finished *and* the window holding its result exists. There is no need to write a wait after each one; the ordering a person would expect is the ordering they get. `background` before an operation is the way to say the opposite – run it and carry on – and `wait` then collects every operation so started.

The active window is what a command without a title acts on. It is the window most recently opened, created, or named by `activate`, and it follows work the script did not name: a filter, a plot and a comparison each make a window, and that new window is what the next command means. When two windows could be meant, say which with `activate`, and give windows names of your own with `title` rather than relying on the ones CTSim generates.

A failure stops the script and the program exits with status 1, which is how a test reports itself. `on-error continue` logs the failure and carries on, for a script that is surveying rather than asserting. `on-error keep` stops the script but leaves the windows standing, which is what a demonstration wants when something goes wrong in front of an audience. Every failure names the script line, the command and what was wrong, on the terminal as well as in the log window.

A script leaves no trace in the settings. What `setting` changes is put back when the script ends, and the values a dialog would remember are read but never written – so a demonstration cannot leave the next person's dialogs full of its own numbers. The exception is deliberate: `cd` moves the directory the file dialogs open in, so that a demonstration which hands over to a person leaves them where the work was.

4 The Command Line Interface

CTSim's functions – `phm2if`, `pjrec` and the rest – run from the command line as well as from the graphical interface. They are the way to use CTSim for batch processing, shell scripting, and on a machine with no display.

From 7.0.0 there is one program with two builds. `ctsim` offers the functions as well as its interface; `ctsim-nogui` is the same program built without the interface, for a machine with no Qt. Everything in this chapter applies to both, and `ctsim` may be written wherever `ctsim-nogui` appears:

```
ctsim phm2if head.if 256 256 --phantom herman
ctsim-nogui phm2if head.if 256 256 --phantom herman
```

do the same thing and produce the same file. Until 7.0.0 the program without the interface was called `ctsimtext`.

4.1 Starting ctsim-nogui

`ctsim-nogui` can be invoked via three different methods.

1. `ctsim-nogui` can be run without any parameters, or `ctsim --shell`. In that case, `ctsim-nogui` offers a command-line to enter the function-names and their parameters. The output of the command is displayed. Further commands may be given at the prompt. The shell is exited by the `quit` command. It uses the readline or libedit library, where the build found one, for history and line editing.
2. `ctsim-nogui` can also be called to execute a single command. This is especially useful for batch files containing multiple `ctsim-nogui` commands. This is invoked by calling
   `ctsim-nogui function-name parameters....`
3. Using operating systems that support soft or hard linking of files (such as UNIX and Linux), the executable file `ctsim-nogui` can be linked to the function names. This is automatically done by the installation program and the `rpm` manager. Thus, to use `ctsim-nogui` with the function name `pjrec`, the below command can be executed:
   `pjrec parameters...`
   as a shortcut to the equivalent command
   `ctsim-nogui pjrec parameters...`

4.2 Parallel processing

The functions that do the arithmetic – `phm2if`, `phm2pj` and `pjrec` – divide their work across the cores of one machine with OpenMP, where the build found it. `ctsim-nogui --version` reports whether OpenMP is in.

CTSim once distributed work across a cluster with MPI, as a separate `ctsim-nogui-lam` binary. That was removed before 7.0.0; a modern desktop has more cores than the clusters it was written for.

4.3 if1

Performs math functions on a single image. The commands works with both real and complex-valued images.

```
if1 input-filename output-filename [options...]
```

Its options are listed by `--help`, and in its manual page.

4.4 if2

Performs math functions on a two images. The command works with both real and complex-valued images.

```
if2 input-filename1 input-filename2 output-filename [options...]
```

Its options are listed by `--help`, and in its manual page.

4.5 ifexport

Export an image file to a standard graphics file.

```
ifexport input-filename output-filename [options...]
```

Its options are listed by `--help`, and in its manual page.

4.6 ifinfo

Displays information about an image file. By default, history labels and image statistics are displayed.

```
ifinfo input-filename [options...]
```

Its options are listed by `--help`, and in its manual page.

4.7 phm2pj

Simulates collection of X-rays data (projections) around a phantom object.

```
phm2pj projection-filename number-detectors number-views [options...]
```

Its options are listed by `--help`, and in its manual page.

4.8 phm2if

Generates a raster image file based on a phantom.

```
phm2if phantom-filename image-filename x-image-size y-image-size [options...]
```

Its options are listed by `--help`, and in its manual page.

4.9 pj2if

Convert a projection file into an image file where each row of the image file contains the projection data from a single view.

```
pj2if projection-filename image-filename [options...]
```

Its options are listed by `--help`, and in its manual page.

4.10 pjinfo

Displays information about a projection file.

```
pjinfo projection-filename [options...]
```

Its options are listed by `--help`, and in its manual page.

4.11 pjrec

Reconstructs the interior of an object from a projection file.

```
pjrec projection-filename image-filename image-cols image-rows [options...]
```

Its options are listed by `--help`, and in its manual page.

4.12 help and bench

`help` at the prompt or on the command line lists every command, and `help pjrec` is `pjrec --help`. `help` knows about `quit`, `bench` and itself as well as the functions.

The graphical interface offers the same shell in a window, under File - New Shell, with Up and Down for the history and Tab to complete a command name. It keeps the same history file, so a command typed at either is remembered by both.

`bench` runs a standard benchmark – rasterize, project and reconstruct at working sizes – and prints the time each step took. `bench --list` names the benchmarks available and `bench filters` runs a named one. They are files installed beside the demonstrations, under `share/ctsim/benchmarks`, and `scripts/benchmark.sh` runs the same files with repetition, statistics and CSV output.

4.13 The other functions

The conversion of this chapter covered the functions CTSim had in 2002. These came later, and each is described by `--help` and by its manual page:

- `png2phm` – make a raster phantom from a PNG image
- `phm2helix` – take helical projections of a phantom
- `pjHinterp` – interpolate helical projections
- `pjfourier` – reconstruct by direct Fourier inversion
- `dicomimport` – import a DICOM file as an image or as projections
- `ctconvert` – convert files between HDF5 and the legacy format
- `ctdump` – print the metadata, history and user attributes of a file
- `linogram` – print linogram sampling

4.13.1 Helical scans

`phm2helix` scans a phantom that changes while the scan proceeds. Before each view it runs a program of yours, which writes the phantom as it stands at that moment to the file named by `--phmfile`; `phm2helix` reads the file, projects that one view and deletes it. The program is called as

```
<program> <view> <nview> <file>
```

with the number of the view about to be taken, counted from zero, the total number of views, and the phantom file to write. `share/ctsim/examples/dynphm.c` is an example, a page of C whose ellipse moves as the view number rises:

```
cc -o dynphm dynphm.c -lm
ctsim-nogui phm2helix helix.pj 255 1080 ./dynphm --phmfile t.phm \
      --geometry equiangular --rotangle 3
```

`pjHinterp` then interpolates the helical projections to a single plane, by 180 degree linear interpolation, and `pjrec` reconstructs that plane. `--interpview` names the view the plane passes through; without it the plane is the earliest the data allows. Two constraints apply. The interpolation is written for the equiangular geometry only, and parallel and equilinear scans are refused. And it needs half a turn plus the fan angle of data on each side of the plane: with CTSim's 60 degree fan that is 240 degrees either way and 480 in all, so a single turn is never enough. Three turns with the plane at the middle view, `--interpview 540` for the scan above, leaves room to spare.

4.14 Odd sizes and counts

Three things must be odd, and the tools refuse an even value with a line saying why:

- the number of detectors in `phm2pj`, so that a detector lies on the central ray;
- `--nsample` in `phm2if`, so that one of the samples per pixel is the centre of the pixel;
- the size of the image `pjrec` makes, so that a pixel lies on the axis of rotation.

Herman's book uses odd throughout – 243 by 243, 345 detectors – for these reasons, and his SNARK refuses even values too. CTSim used to accept an even number of detectors and quietly give it the spacing of one fewer, with a detector appended that read nothing; refusing is plainer.

4.15 File formats

CTSim writes HDF5 by default, for images, projections and plots alike. A file carries its own history: every function that made it, the parameters that function received, and the time it took. `ctdump` prints that record.

The legacy binary format is still read, and is written by the `--legacy-format` option of the functions that write files.

5 Algorithms

CTSim uses a number of interesting algorithms. This appendix details some of the techniques that CTSim uses.

5.1 Phantom Processing

Key Concepts
`Geometric transformations`
`Matrix algebra`
Phantom objects are processed in two different ways: rasterization and projections. CTSim uses optimized techniques to perform those procedures.

The primary tool used to optimize these processes is *Geometric transformations*. For every primitive phantom element, a standardized configuration is defined. This standard configuration is used to speed the process of collecting projections.

In general, to transform an object into the standard configuration, the following sequence of transformations occur. When this sequence is performed, the coordinates are termed the *normalized phantom element* coordinates.

- Scaling by the inverse of its size, that is usually scaling by `(1/dx,1/dy)`.
- Translating the object to the origin, that is usually translation by `(-cx,-cy)`.
- Rotating the object by `-r`.

These steps can by combined into a single matrix multiplication which involves only 4 multiplications and 6 additions to transform both `x` and `y` coordinates. This matrix is precalculated and stored when the phantom is created. Similarly, the inverse of the matrix is precalculated and store to perform the inverse of this transformation.

As an example of this technique, consider the problem of finding the length of an arbitrary line that may intersect an arbitary ellipse. Define the endpoints of the line by `(x1,y1)` and `(x2,y2)`.

1. First, transform the coordinates into the normalized phantom element coordinates. At this point, the ellipse will have been transformed into a unit circle centered at `(0,0)`.
2. Translate the coordinates by `(-x1,-y2)`. The line now has the endpoint centered at the origin. The ellipse will now have its center at `(-x1,-y1)`.
3. Rotate the coordinates by the negative of angle of the line with respect to the x-axis.

At this point the line will now lie along the positive x-axis with one end at `(0,0)`. The circle will be rotated around the origin as well. At this point, it is fairly trivial to calculate the length of the intersection of the line with the unit circle. For example, if the `y` coordinate for the center of the circle is greater than `1` or less than `-1`, then we know that the unit circle doesn't intersect the line at all and stop further processing. Otherwise, the endpoints of the intersection of the line with the unit circle is a simple calculation.

Those new, intersected endpoints are then inverse transformed by reverse of the above transformation sequence. After the inverse translation, the transformed endpoints will be the endpoints of the line that intersect the actual ellipse prior to any transformations.

Though this sequence of events is somewhat complex, it is quite fast since the multiple transformations can be combined into a single matrix multiplication. Further, this technique is amendable to rapidly calculating the intersection of a line with any of the phantom elements that CTSim supports.

5.2 Background Processing

Key Concepts
`Multithreading`
`Critical sections`
`Message passing`
`Re-entrant code`
The CTSim graphical shell can optionally perform background processing. CTSim uses threads as tools to achieve this functionality. Using multiple threads, termed *multithreading*, allows CTSim to:

- Execute a lengthy calculation in the background while the graphical shell remains available for use.
- Automatically take advantage of multiple central processing units (CPU's) in a symmetric multiprocessing (SMP) computer.

When background processing option is turned on or when CTSim is running on a SMP computer, and CTSim is directed to perform reconstruction, rasterization, or projections, CTSim will spawn a *Background Supervisor* thread. This supervisor thread then creates a *Supervisor Event Handler* (supervisor). The supervisor communicates with the rest of graphical user interface of CTSim by using message passing to avoid issues with re-entrant code.

The supervisor registers itself via message passing with the *Background Manager* which will display the execution progress in a pop-up window. The supervisor also registers itself with the document being processed. This is done so that if the document is closed, the document can send a message to the supervisor directing the supervisor to cancel the calculation.

After registering with CTSim components, the supervisor creates *Worker Threads*. These worker threads are the processes that actually perform the calculations. By default, CTSim will create one worker thread for every CPU in the system. As the workers complete unit blocks, they notify the supervisor. The supervisor then sends progress messages to background manager which displays a gauge of the progress.

As the worker threads directly call the supervisor, it is crucial to lock the class data structures with *Critical Sections*. Critical sections lock areas of code and prevent more than one thread to access a section of code at a time. This is used when maintaining the tables of worker threads in the supervisor.

After the workers have completed their tasks, they notify the supervisor. When all the workers have finished, the supervisor kills the worker threads. The supervisor then collates the work units from the workers and sends a message to CTSim to create a new window to display the finished work.

The supervisor then deregisters itself via messages with the background manager and the document. The background manager removes the progress gauge from its display and resizes its window. Finally, the background supervisor exits and background supervisor thread terminates.

This functionality has been compartmentalized into inheritable C++ classes `BackgroundSupervisor`, `BackgroundWorkerThread`, and `BackgroundProcessingDocument`. These classes serve as base classes for the reconstruction, rasterization, and projection multithreading classes.

5.2.1 Advantages

This structure may seem more complex than is necessary, but it has several advantages:

- Since the background threads do not directly call objects in the graphical user interface thread, problems with re-entrant code in the graphical interface are eliminated.
- A supervisor can parallel process with any number of worker threads to take advantage of potentially large numbers of CPU's in SMP computers.
- Allows for continued user-interaction with CTSim while lengthy calculations are performed in the background.

5.2.2 Disadvantages

The above advantages are not free of cost. The disadvantages include:

- Increased memory usage.
  The workers threads allocate memory to store their intermediate results. When the worker threads finish, the supervisor allocates memory for the final result and collates the results for the workers. This collation results in a doubling of the memory requirements. Of course, after collation the supervisor releases the memory used by the workers.
- Slower execution on single CPU systems.
  Creating multiple threads, sending progress messages to the background manager, and collation of results for worker threads adds overhead compared to simply calculating the result directly in the foreground. On single CPU systems this results in slower processing compared to foreground processing. On dual-CPU and greater SMP systems, though, the advantage of using multiple CPU's in parallel exceeds the overhead of background processing.

6 Simple Graphics Package

Simple Graphics Package was created in 1980 by Kevin Rosenberg and is modelled after the hypothetical graphics library described by Foley and van Dam[@FOLEY82]. It is designed to be platform-independent.

6.1 Transformation Sequence

Master coordinate (MC) level

↓

Apply *Current transformation matrix*

↓

World coordinate (WC) level

↓

Clipping against Window

↓

Convert to Normalized device coordinates (NDC)

↓

Clipping against Viewport

↓

Convert to Physical device coordinates (PDC)

6.2 Functions

6.2.1 Master coordinate functions

6.2.2 Normalized coordinate functions

6.2.3 Master coordinate to World coordinate transformations

These transformation functions operate on the *Current transformation matrix* (CTM).

6.2.4 Coordinate transformation functions

6.2.5 State functions

6.3 Coordinate Mapping

6.3.1 Dimensions

Window (World Coordinates): $X_{wmin}, X_{wmax}, Y_{wmin}, Y_{wmax}$
Viewport (Normalized Device Coordinates): $X_{vmin}, X_{vmax}, Y_{vmin}, Y_{vmax}$
Physical (Physicial Device Coordinates): X_{pmax}, Y_{pmax}

6.3.2 Formulas

To convert from Master coordinates to World coordinates:

Apply *current transformation matrix*

To convert from WC to NDC:

$$X_{ndc} = X_{vmin} + (X_{vmax} - X_{vmin})(X_{wc} - X_{wmin})/(X_{wmax} - X_{wmin})$$
$$Y_{ndc} = Y_{vmin} + (Y_{vmax} - Y_{vmin})(Y_{wc} - Y_{wmin})/(Y_{wmax} - Y_{wmin})$$

To convert from NDC to PDC:

$$X_{pdc} = X_{ndc}X_{pmax}$$
$$Y_{pdc} = X_{ndc}Y_{pmax}$$