Cph CT Toolbox

Last updated 28-06-2013 10:58

Contents

1 Introduction

The Cph CT Toolbox contains a collection of applications for reconstructing CT images and volumes from a set of projections. This document describes the overall use of the toolbox applications with some detailed examples.

2 Installation

Please refer to the general README for details about requirements and installation.

3 CT Reconstruction Instructions

In general there are two ways of specifying the settings for CT reconstruction runs.

1. configuration files
2. command line options

The overall flow of all Cph CT Toolbox applications is to start out with application defaults, update with any configuration file settings and finally override with any options supplied on the command line. Thus command line options can be used on their own or as a way to override configuration file settings.

You can find out about configurable settings by running the application with the -H or –help flag. Further details are included in the individual application sections below.

The help output includes a list of all configurable values and their type along with a short description. Each entry contains one or more option formats typically the long option name prefixed with two dashes, the short name prefixed with a single dash and the configuration file option prefixed by 'cfg:' as in:

```
--log-path / -l / cfg:log_path    <type 'str'>
    Log file path (default: )
```

This simply means that you can specify the log file path using either of three methods:

1. Adding a line like
   `log_path = recon.log`
   in the configuration file(s)
2. Calling the application with the –log-path option like:
   `APPLICATION --log-path=recon.log`
3. Calling the application with the -l option like:
   `APPLICATION -l recon.log`

where APPLICATION is replaced by one of the actual application names like katsevich.py or fdk.py.

The ‘`<type 'str'>`‘ part of the output indicates that the setting requires a string value. Other lines may specify that the option requires an integer or a float. The important thing is that the provided value must be meaningful as that type, so for the bool case we could as well use any of the values true, True or 1 to enable a bool setting and one of the values false, False or 0 to disable such a value. Values of float or integer type have more strict limitations, so they must be purely numerical and floats require a period (’.’) as separator between the integer and fractional part.

Each application has default values where it makes sense. The default values are displayed in the ’help output’ shown when the application is run with -H or –help flag.

The help output includes information about all settings in the form of long and short option flags and any arguments the option takes.

All applications look for configuration files in a number of standard locations. Namely APP.cfg files in /.cphcttoolbox and in the working directory in that order. Where APP is the base name of the actual application so that cufdk.py automatically looks for fdk.cfg and cufdk.cfg configuration files. Each configuration file extends any previous ones so the one in the current directory can be used to override settings specified in the general configurations just like command line options can be used to override default and configuration file setting.

3.1 Center FDK

We provide a single wrapper, centerfdk.py, for all the actual compute backends so that it is simple to switch backend engine.

Configurable settings are available in the output of:

```
centerfdk.py --help
```

```
USAGE: ../fanbeam/centerfdk/centerfdk.py [OPTIONS] ARGS
where OPTIONS may include the following:
--help / -H
```

```
    Show this help
--proj-filter / cfg:proj_filter    <type 'str'>
    Projection filter filepath or one of the builtin filters:
['hamming', 'ram-lak', 'shepp-logan', 'cosine', 'hann', 'skip']
    Default: hamming
--source-distance / cfg:source_distance    <type 'float'>
    Distance in cm from source to isocenter
    Default: 3.0
--detector-width / cfg:detector_width    <type 'float'>
    Detector width in cm (-1 for auto)
    Default: -1
--proj-filter-nyquist-fraction / cfg:proj_filter_nyquist_fraction    <type 'float'>
    FDK projection filter nyquest fraction (used when generating builtin filters)
    Default: 1.0
--detector-column-offset / cfg:detector_column_offset    <type 'float'>
    Center ray alignment offset in pixel columns
    Default: 0.0
--load-gpu-binary-path / cfg:load_gpu_binary_path    <type 'str'>
    Path to load compiled CUDA kernels from
    Default:
--save-gpu-kernels-path / cfg:save_gpu_kernels_path    <type 'str'>
    Path to save runtime optimized CUDA kernels code in
    Default:
--cu-postprocess-output / cfg:cu_postprocess_output    <type 'str'>
    Apply specified cuda postprocessing to output array
    Default:
--temporary-directory / cfg:temporary_directory    <type 'str'>
    Prefix for all temporary file paths
    Default: /tmp
--gpu-projs-only / cfg:gpu_projs_only    <type 'bool'>
    Keep partial results on the GPU - no copy to host
    Default: True
--npy-preprocess-input / cfg:npy_preprocess_input    <type 'str'>
    Apply specified numpy preprocessing to input array
    Default:
--gpu-target-filter-memory / cfg:gpu_target_filter_memory    <type 'int'>
    GPU memory in bytes to aim for with projection filter chunking
    Default: 268435456
--detector-distance / cfg:detector_distance    <type 'float'>
    Distance in cm from isocenter to detector
    Default: 3.0
--working-directory / -w / cfg:working_directory    <type 'str'>
    Prefix for all relative paths
    Default: .
--input-precision / -i / cfg:input_precision    <type 'str'>
    Select input data type (precision)
```

```
    Default: float32
--version / -V
    Show version
--proj-weight / cfg:proj_weight    <type 'str'>
    FDK projection weight, filepath or float.
    Default:
--cuda-device-index / cfg:cuda_device_index    <type 'int'>
    Which CUDA device to use: -1 for auto
    Default: -1
--y-min / cfg:y_min    <type 'float'>
    Field of View minimum y coordinate in cm
    Default: -1.0
--proj-filter-width / cfg:proj_filter_width    <type 'int'>
    FDK projection filter resolution, must be a power of two.
    Default: -1
--x-max / cfg:x_max    <type 'float'>
    Field of View maximum x coordinate in cm
    Default: 1.0
--gpu-target-threads / cfg:gpu_target_threads    <type 'int'>
    Number of CUDA kernel threads to aim for per block
    Default: 256
--angle-start / cfg:angle_start    <type 'float'>
    Which angle to begin from
    Default: 0.0
--checksum / cfg:checksum    <type 'int'>
    Level of checksums to enable
    Default: 0
--y-max / cfg:y_max    <type 'float'>
    Field of View maximum y coordinate in cm
    Default: 1.0
--filter / cfg:filter    <type 'str'>
    Type of projection filter to apply
    Default: none
--x-min / cfg:x_min    <type 'float'>
    Field of View minimum x coordinate in cm
    Default: -1.0
--npy-save-output / cfg:npy_save_output    <type 'str'>
    Output saver to format and write output data to one or more files
    Default:
--log-path / -l / cfg:log_path    <type 'str'>
    Log file path (empty for stdout)
    Default:
--detector-row-offset / cfg:detector_row_offset    <type 'float'>
    Center ray alignment offset in pixel rows
    Default: 0.0
--load-gpu-kernels-path
```

```
    Path to load CUDA kernels code from
--chunk-range / cfg:chunk_range    <type 'str'>
    Select range of chunks to reconstruct
--output-precision / cfg:output_precision    <type 'str'>
    Select output data type (precision)
    Default: float32
--cu-preprocess-input / cfg:cu_preprocess_input    <type 'str'>
    Apply specified cuda preprocessing to input array
    Default:
--proj-filter-scale / cfg:proj_filter_scale    <type 'float'>
    FDK projection filter scale (used when generating builtin filters)
    Default: -1.0
--npy-postprocess-output / cfg:npy_postprocess_output    <type 'str'>
    Apply specified numpy postprocessing to output array
    Default:
--complex-precision / cfg:complex_precision    <type 'str'>
    Select internal complex data type (precision)
    Default: complex64
--timelog / cfg:timelog    <type 'str'>
    Log execution times
    Default: default
    Allowed: default, verbose
--projs-per-turn / cfg:projs_per_turn    <type 'int'>
    Number of projections in a full gantry rotation
    Default: 360
--engine / -E / cfg:engine    <type 'str'>
    Back end calculation engine
    Default: numpy
    Allowed: numpy, cuda
--total-turns / cfg:total_turns    <type 'int'>
    Number of full gantry rotations (-1 for auto)
    Default: -1
--precision / -p / cfg:precision    <type 'str'>
    Select internal data type (precision)
    Default: float32
--y-voxels / cfg:y_voxels    <type 'int'>
    Field of View resolution in y
    Default: 512
--save-filtered-projs-data-path / cfg:save_filtered_projs_data_path    <type 'str'>
    Save filtered binary projection data in given path
    Default:
--log-format / cfg:log_format    <type 'str'>
    Log line format
--volume-weight / cfg:volume_weight    <type 'str'>
    FDK reconstructed volume weight, filepath or float.
    Default:
```

```
--limit-sources / cfg:limit_sources    <type 'str'>
    Limit reconstruction to given source angles
--detector-columns / cfg:detector_columns    <type 'int'>
    Number of pixel columns in projections
    Default: 256
--save-gpu-binary-path / cfg:save_gpu_binary_path    <type 'str'>
    Path to save runtime optimized compiled CUDA kernels in
    Default:
--detector-shape / cfg:detector_shape    <type 'str'>
    Shape of detector
    Default: flat
    Allowed: flat, curved
--log-level / -L / cfg:log_level    <type 'str'>
    Log verbosity
--npy-load-input / cfg:npy_load_input    <type 'str'>
    Input loader to parse and read input data into array
    Default:
--gpu-target-input-memory / cfg:gpu_target_input_memory    <type 'int'>
    GPU memory in bytes to aim for as input for backprojection chunking
    Default: 134217728
--detector-pixel-width / cfg:detector_pixel_width    <type 'float'>
    Detector pixel width in cm (-1 for auto)
    Default: -1
--chunk-size / cfg:chunk_size    <type 'int'>
    Number of z slices in reconstruction chunks
    Default: -1
--x-voxels / cfg:x_voxels    <type 'int'>
    Field of View resolution in x
    Default: 512
--proj-chunk-size / cfg:proj_chunk_size    <type 'int'>
    CUDA FDK number of projections processed at a time.
    Default: 1
--detector-pixel-height / cfg:detector_pixel_height    <type 'float'>
    Detector pixel height in cm (-1 for auto)
    Default: -1
--load-gpu-init-path / cfg:load_gpu_init_path    <type 'str'>
    Path to load optional CUDA init code from
    Default:
```

Please refer to the general FDK notes that apply here as well.

3.2 FDK

We provide a single wrapper, fdk.py, for all the actual compute backends so that it is simple to switch backend engine.

Configurable settings are available in the output of:

```
fdk.py --help

USAGE: ../conebeam/fdk/fdk.py [OPTIONS] ARGS
where OPTIONS may include the following:
--npy-save-output / cfg:npy_save_output    <type 'str'>
    Output saver to format and write output data to one or more files
    Default:
--help / -H
    Show this help
--log-path / -l / cfg:log_path    <type 'str'>
    Log file path (empty for stdout)
    Default:
--detector-row-offset / cfg:detector_row_offset    <type 'float'>
    Center ray alignment offset in pixel rows
    Default: 0.0
--load-gpu-kernels-path
    Path to load CUDA kernels code from
--proj-filter / cfg:proj_filter    <type 'str'>
    Projection filter filepath or one of the builtin filters:
['hamming', 'ram-lak', 'shepp-logan', 'cosine', 'hann', 'skip']
    Default: hamming
--chunk-range / cfg:chunk_range    <type 'str'>
    Select range of chunks to reconstruct
--z-voxels / cfg:z_voxels    <type 'int'>
    Field of View resolution in z
    Default: 512
--output-precision / cfg:output_precision    <type 'str'>
    Select output data type (precision)
    Default: float32
--cu-preprocess-input / cfg:cu_preprocess_input    <type 'str'>
    Apply specified cuda preprocessing to input array
    Default:
--proj-filter-scale / cfg:proj_filter_scale    <type 'float'>
    FDK projection filter scale (used when generating builtin filters)
    Default: -1.0
--npy-postprocess-output / cfg:npy_postprocess_output    <type 'str'>
    Apply specified numpy postprocessing to output array
    Default:
--complex-precision / cfg:complex_precision    <type 'str'>
    Select internal complex data type (precision)
    Default: complex64
--timelog / cfg:timelog    <type 'str'>
    Log execution times
    Default: default
    Allowed: default, verbose
--detector-width / cfg:detector_width    <type 'float'>
```

```
    Detector width in cm (-1 for auto)
    Default: -1
--projs-per-turn / cfg:projs_per_turn    <type 'int'>
    Number of projections in a full gantry rotation
    Default: 360
--proj-filter-nyquist-fraction / cfg:proj_filter_nyquist_fraction    <type 'float'>
    FDK projection filter nyquest fraction (used when generating builtin filters)
    Default: 1.0
--engine / -E / cfg:engine    <type 'str'>
    Back end calculation engine
    Default: numpy
    Allowed: numpy, cuda
--total-turns / cfg:total_turns    <type 'int'>
    Number of full gantry rotations (-1 for auto)
    Default: -1
--detector-column-offset / cfg:detector_column_offset    <type 'float'>
    Center ray alignment offset in pixel columns
    Default: 0.0
--precision / -p / cfg:precision    <type 'str'>
    Select internal data type (precision)
    Default: float32
--save-gpu-kernels-path / cfg:save_gpu_kernels_path    <type 'str'>
    Path to save runtime optimized CUDA kernels code in
    Default:
--cu-postprocess-output / cfg:cu_postprocess_output    <type 'str'>
    Apply specified cuda postprocessing to output array
    Default:
--temporary-directory / cfg:temporary_directory    <type 'str'>
    Prefix for all temporary file paths
    Default: /tmp
--y-voxels / cfg:y_voxels    <type 'int'>
    Field of View resolution in y
    Default: 512
--save-filtered-projs-data-path / cfg:save_filtered_projs_data_path    <type 'str'>
    Save filtered binary projection data in given path
    Default:
--detector-rows / cfg:detector_rows    <type 'int'>
    Number of pixel rows in projections
    Default: 64
--gpu-projs-only / cfg:gpu_projs_only    <type 'bool'>
    Keep partial results on the GPU - no copy to host
    Default: True
--npy-preprocess-input / cfg:npy_preprocess_input    <type 'str'>
    Apply specified numpy preprocessing to input array
    Default:
--log-format / cfg:log_format    <type 'str'>
```

```
    Log line format
--gpu-target-filter-memory / cfg:gpu_target_filter_memory    <type 'int'>
    GPU memory in bytes to aim for with projection filter chunking
    Default: 268435456
--volume-weight / cfg:volume_weight    <type 'str'>
    FDK reconstructed volume weight, filepath or float.
    Default:
--limit-sources / cfg:limit_sources    <type 'str'>
    Limit reconstruction to given source angles
--detector-columns / cfg:detector_columns    <type 'int'>
    Number of pixel columns in projections
    Default: 256
--detector-distance / cfg:detector_distance    <type 'float'>
    Distance in cm from isocenter to detector
    Default: 3.0
--working-directory / -w / cfg:working_directory    <type 'str'>
    Prefix for all relative paths
    Default: .
--input-precision / -i / cfg:input_precision    <type 'str'>
    Select input data type (precision)
    Default: float32
--detector-shape / cfg:detector_shape    <type 'str'>
    Shape of detector
    Default: flat
    Allowed: flat, curved
--log-level / -L / cfg:log_level    <type 'str'>
    Log verbosity
--npy-load-input / cfg:npy_load_input    <type 'str'>
    Input loader to parse and read input data into array
    Default:
--gpu-target-input-memory / cfg:gpu_target_input_memory    <type 'int'>
    GPU memory in bytes to aim for as input for backprojection chunking
    Default: 134217728
--z-max / cfg:z_max    <type 'float'>
    Field of View maximum z coordinate in cm
    Default: 1.0
--version / -V
    Show version
--save-gpu-binary-path / cfg:save_gpu_binary_path    <type 'str'>
    Path to save runtime optimized compiled CUDA kernels in
    Default:
--proj-weight / cfg:proj_weight    <type 'str'>
    FDK projection weight, filepath or float.
    Default:
--cuda-device-index / cfg:cuda_device_index    <type 'int'>
    Which CUDA device to use: -1 for auto
```

```
    Default: -1
--y-min / cfg:y_min    <type 'float'>
    Field of View minimum y coordinate in cm
    Default: -1.0
--detector-pixel-width / cfg:detector_pixel_width    <type 'float'>
    Detector pixel width in cm (-1 for auto)
    Default: -1
--load-gpu-binary-path / cfg:load_gpu_binary_path    <type 'str'>
    Path to load compiled CUDA kernels from
    Default:
--load-gpu-init-path / cfg:load_gpu_init_path    <type 'str'>
    Path to load optional CUDA init code from
    Default:
--z-min / cfg:z_min    <type 'float'>
    Field of View minimum z coordinate in cm
    Default: -1.0
--detector-height / cfg:detector_height    <type 'float'>
    Detector height in cm (-1 for auto)
    Default: -1
--proj-filter-width / cfg:proj_filter_width    <type 'int'>
    FDK projection filter resolution, must be a power of two.
    Default: -1
--x-max / cfg:x_max    <type 'float'>
    Field of View maximum x coordinate in cm
    Default: 1.0
--chunk-size / cfg:chunk_size    <type 'int'>
    Number of z slices in reconstruction chunks
    Default: -1
--x-voxels / cfg:x_voxels    <type 'int'>
    Field of View resolution in x
    Default: 512
--gpu-target-threads / cfg:gpu_target_threads    <type 'int'>
    Number of CUDA kernel threads to aim for per block
    Default: 256
--angle-start / cfg:angle_start    <type 'float'>
    Which angle to begin from
    Default: 0.0
--proj-chunk-size / cfg:proj_chunk_size    <type 'int'>
    CUDA FDK number of projections processed at a time.
    Default: 1
--checksum / cfg:checksum    <type 'int'>
    Level of checksums to enable
    Default: 0
--y-max / cfg:y_max    <type 'float'>
    Field of View maximum y coordinate in cm
    Default: 1.0
```

```
--detector-pixel-height / cfg:detector_pixel_height    <type 'float'>
    Detector pixel height in cm (-1 for auto)
    Default: -1
--source-distance / cfg:source_distance    <type 'float'>
    Distance in cm from source to isocenter
    Default: 3.0
--x-min / cfg:x_min    <type 'float'>
    Field of View minimum x coordinate in cm
    Default: -1.0
```

Generally the default of no projection prefiltering produces highly blurred reconstruction output. It is recommended to enable a filter but the choice of filter depends on the actual application. Please refer to filtered back projection literature for details.

3.3 Katsevich

We provide a single wrapper, katsevich.py, for all the actual compute backends so that it is simple to switch backend engine.

Configurable settings are available in the output of:

```
katsevich.py --help
```

```
USAGE: ../conebeam/katsevich/katsevich.py [OPTIONS] ARGS
where OPTIONS may include the following:
--npy-save-output / cfg:npy_save_output    <type 'str'>
    Output saver to format and write output data to one or more files
    Default:
--help / -H
    Show this help
--log-path / -l / cfg:log_path    <type 'str'>
    Log file path (empty for stdout)
    Default:
--detector-row-offset / cfg:detector_row_offset    <type 'float'>
    Center ray alignment offset in pixel rows
    Default: 0.0
--load-gpu-kernels-path
    Path to load CUDA kernels code from
--chunk-range / cfg:chunk_range    <type 'str'>
    Select range of chunks to reconstruct
--z-voxels / cfg:z_voxels    <type 'int'>
    Field of View resolution in z
    Default: 512
--output-precision / cfg:output_precision    <type 'str'>
    Select output data type (precision)
    Default: float32
--cu-preprocess-input / cfg:cu_preprocess_input    <type 'str'>
```

```
    Apply specified cuda preprocessing to input array
    Default:
--npy-postprocess-output / cfg:npy_postprocess_output    <type 'str'>
    Apply specified numpy postprocessing to output array
    Default:
--complex-precision / cfg:complex_precision    <type 'str'>
    Select internal complex data type (precision)
    Default: complex64
--timelog / cfg:timelog    <type 'str'>
    Log execution times
    Default: default
    Allowed: default, verbose
--detector-width / cfg:detector_width    <type 'float'>
    Detector width in cm (-1 for auto)
    Default: -1
--projs-per-turn / cfg:projs_per_turn    <type 'int'>
    Number of projections in a full gantry rotation
    Default: 360
--engine / -E / cfg:engine    <type 'str'>
    Back end calculation engine
    Default: numpy
    Allowed: numpy, cuda
--total-turns / cfg:total_turns    <type 'int'>
    Number of full gantry rotations (-1 for auto)
    Default: -1
--detector-column-offset / cfg:detector_column_offset    <type 'float'>
    Center ray alignment offset in pixel columns
    Default: 0.0
--precision / -p / cfg:precision    <type 'str'>
    Select internal data type (precision)
    Default: float32
--save-gpu-kernels-path / cfg:save_gpu_kernels_path    <type 'str'>
    Path to save runtime optimized CUDA kernels code in
    Default:
--cu-postprocess-output / cfg:cu_postprocess_output    <type 'str'>
    Apply specified cuda postprocessing to output array
    Default:
--temporary-directory / cfg:temporary_directory    <type 'str'>
    Prefix for all temporary file paths
    Default: /tmp
--y-voxels / cfg:y_voxels    <type 'int'>
    Field of View resolution in y
    Default: 512
--save-filtered-projs-data-path / cfg:save_filtered_projs_data_path    <type 'str'>
    Save filtered binary projection data in given path
    Default:
```

```
--detector-rows / cfg:detector_rows    <type 'int'>
    Number of pixel rows in projections
    Default: 64
--gpu-projs-only / cfg:gpu_projs_only    <type 'bool'>
    Keep partial results on the GPU - no copy to host
    Default: True
--npy-preprocess-input / cfg:npy_preprocess_input    <type 'str'>
    Apply specified numpy preprocessing to input array
    Default:
--log-format / cfg:log_format    <type 'str'>
    Log line format
--gpu-target-filter-memory / cfg:gpu_target_filter_memory    <type 'int'>
    GPU memory in bytes to aim for with projection filter chunking
    Default: 268435456
--detector-rebin-rows / cfg:detector_rebin_rows    <type 'int'>
    Number of rows in projection rebinning
    Default: 64
--limit-sources / cfg:limit_sources    <type 'str'>
    Limit reconstruction to given source angles
--progress-per-turn / -P / cfg:progress_per_turn    <type 'float'>
    Geometrical helix pitch in cm, i.e. conveyor progress per turn
    Default: 1.0
--detector-columns / cfg:detector_columns    <type 'int'>
    Number of pixel columns in projections
    Default: 256
--detector-distance / cfg:detector_distance    <type 'float'>
    Distance in cm from isocenter to detector
    Default: 3.0
--working-directory / -w / cfg:working_directory    <type 'str'>
    Prefix for all relative paths
    Default: .
--input-precision / -i / cfg:input_precision    <type 'str'>
    Select input data type (precision)
    Default: float32
--detector-shape / cfg:detector_shape    <type 'str'>
    Shape of detector
    Default: flat
    Allowed: flat, curved
--log-level / -L / cfg:log_level    <type 'str'>
    Log verbosity
--npy-load-input / cfg:npy_load_input    <type 'str'>
    Input loader to parse and read input data into array
    Default:
--gpu-target-input-memory / cfg:gpu_target_input_memory    <type 'int'>
    GPU memory in bytes to aim for as input for backprojection chunking
    Default: 134217728
```

```
--z-max / cfg:z_max    <type 'float'>
    Field of View maximum z coordinate in cm
    Default: 1.0
--version / -V
    Show version
--save-gpu-binary-path / cfg:save_gpu_binary_path    <type 'str'>
    Path to save runtime optimized compiled CUDA kernels in
    Default:
--cuda-device-index / cfg:cuda_device_index    <type 'int'>
    Which CUDA device to use: -1 for auto
    Default: -1
--y-min / cfg:y_min    <type 'float'>
    Field of View minimum y coordinate in cm
    Default: -1.0
--detector-pixel-width / cfg:detector_pixel_width    <type 'float'>
    Detector pixel width in cm (-1 for auto)
    Default: -1
--load-gpu-binary-path / cfg:load_gpu_binary_path    <type 'str'>
    Path to load compiled CUDA kernels from
    Default:
--load-gpu-init-path / cfg:load_gpu_init_path    <type 'str'>
    Path to load optional CUDA init code from
    Default:
--z-min / cfg:z_min    <type 'float'>
    Field of View minimum z coordinate in cm
    Default: -1.0
--detector-height / cfg:detector_height    <type 'float'>
    Detector height in cm (-1 for auto)
    Default: -1
--x-max / cfg:x_max    <type 'float'>
    Field of View maximum x coordinate in cm
    Default: 1.0
--chunk-size / cfg:chunk_size    <type 'int'>
    Number of z slices in reconstruction chunks
    Default: -1
--x-voxels / cfg:x_voxels    <type 'int'>
    Field of View resolution in x
    Default: 512
--gpu-target-threads / cfg:gpu_target_threads    <type 'int'>
    Number of CUDA kernel threads to aim for per block
    Default: 256
--angle-start / cfg:angle_start    <type 'float'>
    Which angle to begin from
    Default: 0.0
--checksum / cfg:checksum    <type 'int'>
    Level of checksums to enable
```

```
    Default: 0
--y-max / cfg:y_max    <type 'float'>
    Field of View maximum y coordinate in cm
    Default: 1.0
--detector-pixel-height / cfg:detector_pixel_height    <type 'float'>
    Detector pixel height in cm (-1 for auto)
    Default: -1
--source-distance / cfg:source_distance    <type 'float'>
    Distance in cm from source to isocenter
    Default: 3.0
--x-min / cfg:x_min    <type 'float'>
    Field of View minimum x coordinate in cm
    Default: -1.0
```

Please note that the total_turns configuration value is the number of core scan rotations. That is, the number 'without' the extra overscan rotation. Thus if you specify 4 turns it means 5 actual turns if the overscan is counted, too. If the total_turns setting is left unset the value will be automatically calculated from the z range and progress per turn again adding one extra rotation of overscan.

4 Reconstruction Examples

Concrete examples are often easier to grasp, so this section explains a few common use examples for each of the applications. Some of the examples use command line options, but every setting could be specified in the configuration files instead. Vice versa for the examples using configuration files. All the examples are available for download from the official project site:

https://code.google.com/p/cphcttoolbox/downloads/list

Please note that the examples below expect the toolbox applications to be either installed or available in the general search path. If you want to run them directly from a checkout or unpack you may need to set the PATH environment.

4.1 Center FDK

4.1.1 Tiny Numpy Engine Example

Download and unpack e.g. the circular 1x32 example from our project page somewhere for out-of-the-box use:

https://cphcttoolbox.googlecode.com/files/circular-shepp-logan-1x32.zip

Then execute the Center FDK application with the numpy engine from the directory where you unpacked the example.

```
centerfdk.py --engine=numpy \
    circular/shepp-logan-1x32/shepp-logan-1x32-float32-32x32x1-auto.cfg
```

This will run the default numpy CenterFDK reconstruction on a tiny sample and produce output like

```
2012-10-02 16:03:32,464 INFO Initializing loadscene, flux2proj,
saveslices, saveslices numpy plugin(s)
2012-10-02 16:03:32,465 INFO Starting curved FDK reconstruction
...
2012-10-02 16:03:33,555 INFO Complete time used 1.092s
```

and save result images in circular/shepp-logan-1x32/output-1x32-32x32x1-auto/ In this case the configuration creates just a single z slice and it should finish in reasonable time (minutes) even on limited machines.

4.1.2 Tiny CUDA Engine Example

To try out the CUDA version instead use the same downloaded example and simply change the engine flag:

```
centerfdk.py --engine=cuda \
    circular/shepp-logan-1x32/shepp-logan-1x32-float32-32x32x1-auto.cfg
```

If you have a working CUDA and pycuda installation it should yield the same reconstructed results in the output directory but at a much faster pace.

4.1.3 Medium CUDA Engine Example

To try out the CUDA version on a bigger example instead download and unpack the example from:

https://cphcttoolbox.googlecode.com/files/circular-shepp-logan-1x128.zip

Then run it with

```
centerfdk.py --engine=cuda \
    circular/shepp-logan-1x128/shepp-logan-1x128-float32-128x128x1-auto.cfg
```

If you have a working CUDA and pycuda installation it should yield the same reconstructed results in the output directory but at a much faster pace.

4.2 FDK

4.2.1 Tiny Numpy Engine Example

Download and unpack e.g. the circular 8x32 example from our project page somewhere for out-of-the-box use:

https://cphcttoolbox.googlecode.com/files/circular-shepp-logan-8x32.zip

Then execute the FDK application with the numpy engine from the directory where you unpacked the example.

```
fdk.py --engine=numpy \
    circular/shepp-logan-8x32/shepp-logan-8x32-float32-32x32x32-auto-single.cfg
```

This will run the default numpy FDK reconstruction on a tiny sample and produce output like

```
2012-09-21 15:27:29,211 INFO Initializing loadscene, saveslices,
saveslices, savestacked numpy plugin(s)
2012-09-21 15:27:29,212 INFO Starting curved FDK reconstruction
...
2012-09-21 15:27:31,359 INFO Complete time used 2.150s
```

and save result images in circular/shepp-logan-8x32/output-8x32-32x32x32-auto-single/ In this case the configuration creates just a single z chunk of two slices to finish in reasonable time (minutes) even on limited machines.

It should only take slightly longer to reconstruct the entire volume with the other configuration file:

```
fdk.py --engine=numpy \
    circular/shepp-logan-8x32/shepp-logan-8x32-float32-32x32x32-auto.cfg
```

4.2.2 Tiny CUDA Engine Example

To try out the CUDA version instead use the same downloaded example and simply change the engine flag:

```
fdk.py --engine=cuda \
    circular/shepp-logan-8x32/shepp-logan-8x32-float32-32x32x32-auto-single.cfg
```

If you have a working CUDA and pycuda installation it should yield the same reconstructed results in the output directory but at a much faster pace.

Again you can quickly reconstruct the entire volume as well:

```
fdk.py --engine=cuda \
    circular/shepp-logan-8x32/shepp-logan-8x32-float32-32x32x32-auto.cfg
```

4.2.3 Medium CUDA Engine Example

To try out the CUDA version on a bigger example instead download and unpack the example from:

https://cphcttoolbox.googlecode.com/files/circular-shepp-logan-32x128.zip

Then run it with

```
fdk.py --engine=cuda \
    circular/shepp-logan-32x128/shepp-logan-32x128-float32-128x128x128-auto.cfg
```

If you have a working CUDA and pycuda installation it should yield the same reconstructed results in the output directory but at a much faster pace.

4.2.4 Large CUDA Engine Example

To try out the CUDA version on a large example instead download and unpack the example from:

https://cphcttoolbox.googlecode.com/files/circular-shepp-logan-128x512.zip

Then run it with

```
fdk.py --engine=cuda \
    circular/shepp-logan-128x512/shepp-logan-128x512-float32-512x512x512-auto.cfg
```

If you have a working CUDA and pycuda installation it should yield the same reconstructed results in the output directory but at a much faster pace.

4.3 Katsevich

4.3.1 Tiny Numpy Engine Example

Download and unpack e.g. the tiny example from our project page somewhere for out-of-the-box use:

https://cphcttoolbox.googlecode.com/files/spiral-shepp-logan-8x32.zip

Then execute the Katsevich application with the numpy engine from the directory where you unpacked the example.

```
katsevich.py --engine=numpy \
    spiral/shepp-logan-8x32/shepp-logan-8x32-float32-32x32x32-auto-0.5-rebin-16-single.cfg
```

This will run the default numpy Katsevich reconstruction on a tiny sample and produce output like

```
2012-09-06 10:04:34,286 INFO Initializing loadscene, clip, saveslices,
saveslices, savestacked numpy plugin(s)
2012-09-06 10:04:34,288 INFO Starting curved Katsevich reconstruction
...
2012-09-06 10:06:37,888 INFO Complete time used 123.606s
```

and save result images in spiral/shepp-logan-8x32/output-8x32-32x32x32-auto-0.500-rebin-16-single/ In this case the configuration creates just a single z chunk of two slices to finish in reasonable time (minutes) even on limited machines.

4.3.2 Tiny CUDA Engine Example

To try out the CUDA version instead use the same downloaded example and simply change the engine flag:

```
katsevich.py --engine=cuda \
    spiral/shepp-logan-8x32/shepp-logan-8x32-float32-32x32x32-auto-0.5-rebin-16-single.cfg
```

If you have a working CUDA and pycuda installation it should yield the same reconstructed results in the output directory but at a much faster pace.

4.3.3 Medium CUDA Engine Example

To try out the CUDA version on a bigger example instead download and unpack the example from:

https://cphcttoolbox.googlecode.com/files/spiral-shepp-logan-32x128.zip

Then run it with

```
katsevich.py --engine=cuda \
    spiral/shepp-logan-32x128/shepp-logan-32x128-float32-128x128x128-auto-0.5-rebin-64.cfg
```

If you have a working CUDA and pycuda installation it should yield the same reconstructed results in the output directory but at a much faster pace.

4.3.4 Large CUDA Engine Example

To try out the CUDA version on a large example instead download and unpack the example from:

https://cphcttoolbox.googlecode.com/files/spiral-shepp-logan-128x512.zip

Then run it with

```
katsevich.py --engine=cuda \
    spiral/shepp-logan-128x512/shepp-logan-128x512-float32-512x512x512-auto-0.5-rebin-256.cf
```

If you have a working CUDA and pycuda installation it should yield the same reconstructed results in the output directory but at a much faster pace.

4.3.5 Tiny Numpy Engine Example Revisited

Lets take another look at the numpy reconstruction example above to get an idea about the configuration possibilities. We use the sample Shepp-Logan projections in uint16 format this time for a curved 8x32 pixel detector. With the raw uint16 intensities we have to convert the input data to proper attenuation projections with the flux2proj plugin as part of the process. This time we reconstruct the volume with a resolution of 64 voxels in each direction. The conveyor belt moves the object 0.5 cm at a constant speed for each rotation and the distance between the X-ray source and the iso-center is 3.0 cm just like the distance from the iso-center to the detector. We choose to reconstruct at a 64-bit precision to get high quality at not much added cost for most CPUs. Finally we decide to handle chunks of 8 z-slices at a time and save images of these chunks concatenated in a file and save the corresponding meta data for later reference.

```
katsevich.py --engine=numpy \
    --npy-load-input=loadscene#spiral/shepp-logan-8x32/input-8x32-auto-uint16/\
raw-scene_1801_8_32_0.500_1.000-3.000_2.000_2.000_2.000_uint16.csv \
    --npy-preprocess-input=flux2proj#0#65535 \
    --detector-rows=8 --detector-columns=32 --x-min=-1 --x-max=1 --y-min=-1 \
    --y-max=1 --z-min=-1 --z-max=1 --source-distance=3.0 \
```

```
    --detector-distance=3.0 --progress-per-turn=0.5 --projs-per-turn=360 \
    --total-turns=4 --detector-shape=curved --input-precision=uint16 \
    --precision=float64 --x-voxels=64 --y-voxels=64 --z-voxels=64 \
    --chunk-size=1 --chunk-range=11:12 --detector-rebin-rows=16 \
    --npy_postprocess_output=clip#0#2 \
    --npy_save_output=savestacked#z#spiral/shepp-logan-8x32/\
output-8x32-64x64x64-auto-0.5-rebin-16-single/z_slices-%d-%d.png
```

If everything goes as planned the execution will slowly proceed printing progress details including timing values

```
2012-09-21 11:49:35,320 INFO Initializing loadscene, clip, savestacked
numpy plugin(s)
2012-09-21 11:49:35,321 INFO Starting curved Katsevich reconstruction
...
2012-09-21 11:52:38,641 INFO Complete time used 183.321s
```

and save result images in spiral/shepp-logan-8x32/output-8x32-64x64x64-auto-0.5-rebin-16-single/ which you can inspect in your favorite image viewer. The resolution is not that high so the images will be of modest quality even though we used high precision for the calculations. Please note that we do not currently support float64 precision in the cuda engine, so replacing numpy with cuda in the example above will fail.

4.3.6 Medium CUDA Engine Example Revisited

Lets rerun the medium sized reconstruction example but with different settings. We use sample Shepp-Logan projections in float32 format for a curved 32x128 pixel detector. The region of interest is set to the range -0.5, cm in all three dimensions and we reconstruct that volume with a resolution of 512 voxels in each direction. The conveyor belt moves the object 0.5 cm at a constant speed for each rotation and the distance between the X-ray source and the iso-center is 3.0 cm just like the distance from the iso-center to the detector. We choose to reconstruct at a 32-bit precision to get the best trade off between performance and quality for most GPUs. Finally we decide to handle chunks of 16 z-slices at a time and save images of these chunks concatenated in a file.

```
katsevich.py --engine cuda \
    --npy_load_input=loadscene#spiral/shepp-logan-32x128/\
input-32x128-auto-float32/\
raw-scene_1801_32_128_0.500_1.000-3.000_2.000_2.000_2.000_float32.csv \
    --input-precision=float32 --detector-rows=32 --detector-columns=128 \
    --x-min=-0.5 --x-max=0.5 --y-min=-0.5 --y-max=0.5 --z-min=-0.5 --z-max=0.5 \
    --source-distance=3.0 --detector-distance=3.0 \
    --progress-per-turn=0.5 --precision=float32 --x-voxels=512 \
    --y-voxels=512 --z-voxels=512 --chunk-size=16 --detector-shape=curved \
    --detector-rebin-rows=64 --npy_postprocess_output=clip#0#2 \
```

```
    --npy_save_output=savestacked#z#spiral/shepp-logan-32x128/\
output-32x128-512x512x512-auto-0.5-rebin-64-single/z_slices-%d-%d.png#16
```

If everything goes as planned the execution will quickly print progress details including timing values

```
2012-09-21 12:03:01,963 INFO Init GPU -1
2012-09-21 12:03:02,140 INFO using GPU 0: GeForce GTX 460
...
2012-09-21 12:03:43,546 INFO Complete time used 41.583s
```

and save result images in spiral/shepp-logan-32x128/output-32x128-512x512x512-auto-0.5-rebin-64-single/ which you can inspect in your favorite image viewer. The lower calculation precision typically does not hurt the quality of the output much.

5 Plugins

The toolbox provides extreme flexibility using the included or user-implemented plugins. Input and output data is loaded with the load and save plugins and all kinds of pre- and postprocessing is possible before and after reconstruction. We provide basic plugins to cover the most common cases of input and output but it is not hard to create your own plugins if you want to work on data in another format. All applications include numpy plugin support and the applications with optimized back end engines additionally allow engine specific plugins so that it is possible to process the data directly while it is still inside the engines e.g. on the GPU.

Plugins are enabled with the plugin hooks configuration or command line options. The options are generally on the form ENGINE-HOOK-NAME where ENGINE is the back end engine short name (e.g npy or cu) and HOOK-NAME is a two word description of the target and action. An example option is

```
--npy-load-input / cfg:npy_load_input
```

which loads input using numpy. This is the general load mechanism for all engines and the data is automatically transferred to the engine buffers for the optimized engines before use.

Plugins typically take one or more positional or named arguments. To pass arguments to a plugin simply provide the plugin name followed by `#` separated argument values:

```
--npy-postprocess-output=PLUGIN#ARG0#ARG1#ARG2
```

A simple example would be the clip output plugin that limits all values to the range [MINVAL:MAXVAL] for given MINVAL and MAXVAL values. With MINVAL 0 and MAXVAL 100 that would be specified with the command line option:

```
--npy-postprocess-output=clip#0#100
```

Equivalently the named argument form can be used:

```
--npy-postprocess-output=clip#clip_min=0#clip_max=100
```

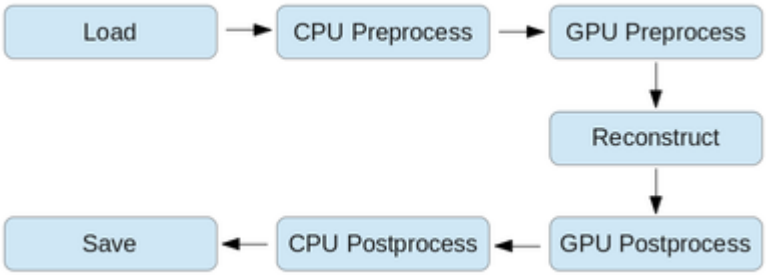
This is most useful if the plugin takes multiple arguments where some of them have default values. Then it may be simpler to only provide the arguments where the defaults should be overridden.

Plugins can be chained endlessly to provide completely free processing. Just use a `:` between plugin arguments to pass the output of one plugin to the next one like so:

```
--npy-postprocess-output=clip#0#100:normalize#2#42
```

This will result in the output first being clipped to the range [0:100] and then normalized to the range [2:42]. The plugin combination and values here are completely arbitrary, so it is only to show the chaining.

The engine-specific plugin hooks are called in the same way but with the npy prefix replaced by the engine shortname. E.g. the applications using a CUDA engine will provide not only a npy-preprocess-input hook but also a cu-preprocess-input hook that can be used to modify the input data after the usual numpy input preprocessing is completed. For the optimized engines data is typically moved to high performance hardware like GPUs before the main calculations, so those engine plugins work on the data directly on the device, whereas the numpy plugins work on the data before it is moved to the device and after it is moved back from there. The general plugin order is outlined below with the Numpy CPU plugins and CUDA GPU plugin hooks. For the non-GPU versions the GPU pre-/post-preprocessing is simply skipped.



Toolbox applications generally use chunking to limit memory use so all the plugins are called as shown above but repeated for each chunk.

Numpy plugins work on standard ndarrays [1] and CUDA plugins work on the very similar GPUArray [2] abstraction.

[1] http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html
[2] http://documen.tician.de/pycuda/array.html#the-gpuarray-array-class

5.1 Load Plugins

All input data should be loaded through a load plugin. We provide the `loadscene` numpy plugin that takes a scene file with meta data and individual intensity or attenuation files and loads the data into the applications. It is also possible to load the actual input values from a single binary file instead if that is preferred. The other important information in scene files is the scan angles. The same angles are used even if a separate binary projections file is used. The loadscene plugin is flexible enough to handle most common cases including the provided examples, but you can write your own if you want to support another data format.

5.1.1 Scene File

We use a common comma-separated list of fields format where each line contains the path to a projection and a projection angle in degrees. The projection files can be in raw byte format or a common image format like PNG. The scene parser tries to auto detect the proper format handler based on the file extension. In the case of raw format it is typically necessary to provide the byte interpretation as an option to the application execution. We support the most widely used formats like uint16, uint32, float32 and float64. Intensity files with detector readings are supported in combination with the flux2proj plugin which translates measured intensities/flux to actual X-ray attenuation values before the reconstruction. This step includes scaling with the provided zero and air normalization values. Please refer to the corresponding preprocess section for details.

5.1.2 Single Projections File

It is possible to load projections from a single raw binary data file instead of from individual projection files. In that case the scene file is only used for the angle meta data and the loaded projections are mapped on a line by line basis to the scene settings. We generally use row-major matrices, so the projections should be written row-by-row.

5.2 Preprocess Input

It is possible to preprocess the input data before the actual filtering and back projection steps. It is common for real scanners to produce X-ray intensity/flux measurements rather than the attenuation values needed for reconstruction. Thus it is necessary to translate the flux values before the reconstruction commences. We provide a flux2proj plugin for that purpose.

5.2.1 Intensity to Attenuation

The flux2proj plugin translates measured intensities/flux to actual X-ray attenuation values before the reconstruction. This step includes scaling with the

provided zero and air normalization values. The zero normalization value is the measured intensity with the X-ray source turned off and the air normalization is the intensity measured for with the X-ray on but nothing but air in the scan region. Both values can be provided as a constant value or as a path to a binary file with values for all detector pixels. For idealised uint16 intensity/flux measurements the zero normalization could be 0 and air normalization 65535 and that would be specified like:

```
--npy-preprocess-input=flux2proj#0#65535
```

It is highly unlikely that real scanners provide such clean and simple values, so it is more likely that files with individual calibration measurements are required:

```
--npy-preprocess-input=flux2proj#zero.bin#air.bin
```

where zero.bin is a single position scan with the X-ray source off and air.bin is a single scan without anything in the scan region. Both files should be binary encoded with the same data type as the scan input, i.e. uint16 in this case.

5.3 Postprocess Output

It is possible to postprocess the output data after the actual filtering and back projection steps. It is quite common to limit or scale the output range for real use to produce human friendly images. We provide a few plugins including clip and normalize for that purpose, but you may add your own if you want to modify the output in another way. We can revisit the clip and normalize example to show how:

```
--npy-postprocess-output=clip#0#100:normalize#2#42
```

5.3.1 Value Clipping

The clip plugin limits the value range by cutting off all values outside a provided minimum and maximum value. All values outside the provided range are set to the boundary values. This is useful to remove e.g. outliers and noise outside the circular FoV. Some filters may introduce negative values in the reconstructed result and those values can be truncated to zero with something like:

```
--npy-postprocess-output=clip#0#10000
```

5.4 Save Output

After reconstructing a volume it is possible to save the volume in one or more file formats ranging from a raw binary dump of the voxels to individual or stacked volume slice images. We provide the `savevolume`, `saveslices` and `savestacked` output plugins for that purpose. They do not change the output so they can be enabled individually or in combination with basic plugin chaining.

5.4.1 Raw Volume

Use the `savevolume` plugin to dump the entire reconstructed volume to a single binary file. Then it can be loaded into e.g. ImageJ [3] or be manually further processed with numpy/python scripts:

```
--npy-save-output=savevolume#PATH/TO/VOLUME.bin
```

5.4.2 Slice Images

For more human friendly output it may be convenient to use the `saveslices` or `savestacked` plugins. They are both called with a volume slice dimension (x, y or z) and a file path pattern.

```
--npy-save-output=saveslice#DIM#SLICEPATHPATTERN
```

```
--npy-save-output=savestacked#DIM#STACKPATHPATTERN[#STACKSIZE]
```

Actual examples could be:

```
--npy-save-output=saveslice#z#img/z-slice-%d.png
```

to save individual z slices in the img subdirectory with filenames z-slice-0.png and so on.

```
--npy-save-output=savestacked#z#z-stack_%d-%d.png
```

to save stacked z slices in the working directory with filenames z-stack_0-15.png and so on.

The file patterns use the standard python string expansion patterns like `%d` to insert the slice index. Please refer to the String Formatting Operations section in the Python library reference for details.

The optional STACKSIZE argument to savestacked can be used to create N/STACKSIZE files of STACKSIZE stacked images each instead of a single file with the entire stack of N slices.

6 Tools

A number of commonly used tools are included in the toolbox in the tools (sub)directories. Some of the tools mimic the plugins so that the in-line processing with plugins can be pulled out into a stand-alone preprocessing. This is useful to save time if the reconstruction is repeated on the same data multiple times or for inspecting the preprocessed results during debug. Other tools allow resampling and cropping of the input data to compensate for any scanner intricacies. Each tool includes short option help available when run with the –help flag.

[3] http://rsb.info.nih.gov/ij/

6.1 Intensity to Attenuation

Let's revisit the example of converting measured intensities/flux to actual X-ray attenuation values as described in the flux2proj plugin section. If the conversion is wanted once and for all it is possible to run it through the genflux2proj.py tool and save the result as direct input for future reconstructions. For idealised uint16 intensity/flux measurements the zero normalization could be 0 and air normalization 65535 and that would be specified like:

```
genflux2proj.py --npy-load-input=loadscene#SCENE_ARGS \
    --detector-rows=DETECTOR_ROWS --detector-columns=DETECTOR_COLUMNS \
    --zero-norm=0 --air-norm=65535 --flux2proj-save-path=SAVE_PATH
```

where the first three arguments are the same as would be used for the corresponding reconstruction with the conversion as plugin.

Again the zero and air norms can be single scalars or files with individual values.

6.2 Fan slices

Instead of downloading the 1x32 projections in the centerfdk example you could use the downloaded circular 8x32 example mentioned in the ordinary FDK examples and create the single row version of the input data with the genfanslices.py script from the conebeam/tools directory.

```
DTYPE="float32"
INPUT="circular/shepp-logan-8x32/input-8x32-auto-$DTYPE"
OUTPUT="circular/shepp-logan-1x32/input-1x32-auto-$DTYPE"
genfanslices.py --npy-load-input=loadscene#$INPUT/\
raw-scene_360_8_32_0.000_1.000-3.000_2.000_2.000_2.000_$DTYPE.csv \
    --save-projs-image-path=$OUTPUT/projection.%d.raw \
    --save-projs-scene-path=$OUTPUT/\
raw-scene_360_1_32_0.000_1.000-3.000_2.000_2.000_2.000_$DTYPE.csv \
    --precision=float32 --input-precision=$DTYPE --output-precision=$DTYPE \
    --detector-shape=curved --detector-distance=3 --source-distance=3 \
    --detector-columns=32 --detector-rows=8 \
    --chunk-projs=360 --use-relative-scene-paths=true
```

You would still need to fetch the reconstruction configuration or manually create it e.g. from the 8x32 FDK version, though. Similarly the 1x128 projections can be generated from the 32x128 example:

```
DTYPE="float32"
INPUT="circular/shepp-logan-32x128/input-32x128-auto-$DTYPE"
OUTPUT="circular/shepp-logan-1x128/input-1x128-auto-$DTYPE"
genfanslices.py --npy-load-input=loadscene#$INPUT/\
raw-scene_360_32_128_0.000_1.000-3.000_2.000_2.000_2.000_$DTYPE.csv \
```

```
    --save-projs-image-path=$OUTPUT/projection.%d.raw \
    --save-projs-scene-path=$OUTPUT/\
raw-scene_360_1_128_0.000_1.000-3.000_2.000_2.000_2.000_$DTYPE.csv \
    --precision=float32 --input-precision=$DTYPE --output-precision=$DTYPE \
    --detector-shape=curved --detector-distance=3 --source-distance=3 \
    --detector-columns=128 --detector-rows=32 \
    --chunk-projs=360 --use-relative-scene-paths=true
```

and the 1x512 projections can be generated from the 128x512 example:

```
DTYPE="float32"
INPUT="circular/shepp-logan-128x512/input-128x512-auto-$DTYPE"
OUTPUT="circular/shepp-logan-1x512/input-1x512-auto-$DTYPE"
genfanslices.py --npy-load-input=loadscene#$INPUT/\
raw-scene_360_128_512_0.000_1.000-3.000_2.000_2.000_2.000_$DTYPE.csv \
    --save-projs-image-path=$OUTPUT/projection.%d.raw \
    --save-projs-scene-path=$OUTPUT/\
raw-scene_360_1_512_0.000_1.000-3.000_2.000_2.000_2.000_$DTYPE.csv \
    --precision=float32 --input-precision=$DTYPE --output-precision=$DTYPE \
    --detector-shape=curved --detector-distance=3 --source-distance=3 \
    --detector-columns=512 --detector-rows=128 \
    --chunk-projs=360 --use-relative-scene-paths=true
```

6.3 Sinograms

Sometimes it is easier to inspect sinograms than individual projections. We provide a tool to generate sinograms from both fan and cone beam projections. Let's proceed with the 1x128 projections from the centerfdk example and gather all the projections in a single sinogram image with the gensinograms.py script from the conebeam/tools directory.

```
DTYPE="float32"
INPUT="circular/shepp-logan-1x128/input-1x128-auto-$DTYPE"
OUTPUT="circular/shepp-logan-1x128/input-1x128-auto-$DTYPE"
gensinograms.py --npy-load-input=loadscene#$INPUT/\
raw-scene_360_1_128_0.000_1.000-3.000_2.000_2.000_2.000_$DTYPE.csv \
    --save-projs-image-path=$OUTPUT/sinograms.%d.png \
    --precision=float32 --input-precision=$DTYPE --output-precision=$DTYPE \
    --detector-shape=curved --detector-distance=3 --source-distance=3 \
    --detector-columns=128 --detector-rows=1 \
    --chunk-projs=360 --use-relative-scene-paths=true
```

6.4 Arbitrary Pre/postprocessing Chains

Arbitrarily complex preprocessing chains can be effectuated with the general preprocess tool that mimics the reconstructions preparation steps and enables saving just before the reconstruction would commence.

```
preprocess.py --npy-load-input=loadscene#SCENE_ARGS \
    --detector-rows=DETECTOR_ROWS --detector-columns=DETECTOR_COLUMNS \
    --npy-preprocess-input=clip#10#65535:flux2proj#0#65535 \
    --save-projs-image-path=SAVE_PATH
```

where all but the save option are the same as would be used for the corresponding reconstruction with the relevant plugins.

Thus if we revisit the manual katsevich example above we could do just the preprocessing with:

```
preprocess.py --engine=numpy \
    --npy-load-input=loadscene#spiral/shepp-logan-8x32/input-8x32-auto-uint16/\
raw-scene_1801_8_32_0.500_1.000-3.000_2.000_2.000_2.000_uint16.csv \
    --npy-preprocess-input=flux2proj#0#65535 \
    --detector-rows=8 --detector-columns=32 --source-distance=3.0 \
    --detector-distance=3.0  --projs-per-turn=360 \
    --total-turns=4 --detector-shape=curved --input-precision=uint16 \
    --precision=float64 \
    --save-projs-image-path=spiral/shepp-logan-8x32/input-8x32-auto-float32/\
preprocessed.%d.png
```

7 Input/Output

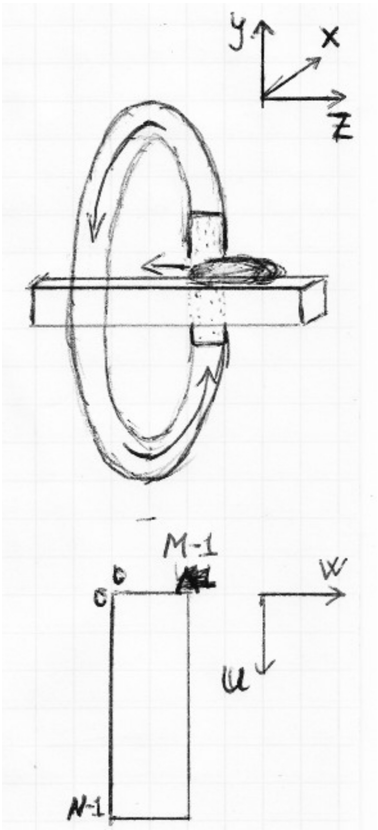
The load and save plugins are generally quite flexible in terms of file formats. It is often just a matter of passing the wanted file extension. Thus the saveslices plugin automatically writes e.g. PNG and JPEG images if the output path is set to end in .png or .jpg respectively. The same applies for the scene files with the loadscene plugin.

8 Scanner Setup

Some variables are not obvious without further scanner definitions like axis directions and so on. This section tries to outline how the the variables are interpreted so that it is possible to use the toolbox with custom scanner configurations.

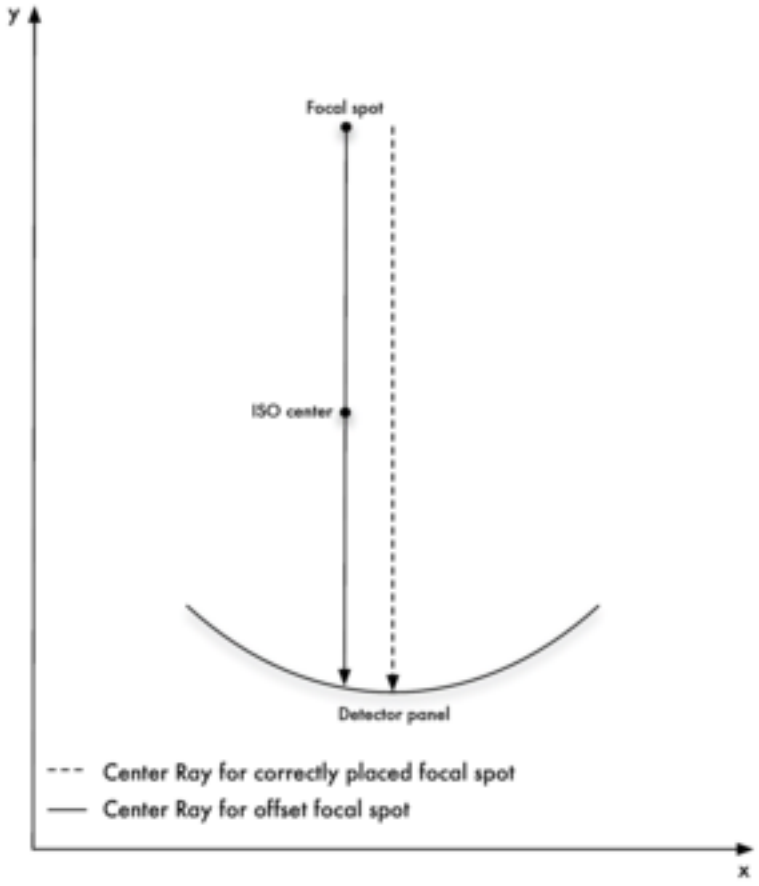
The scanner is positioned with the coordinate system system in a way where the scan object is moved on the z-axis but in the opposite direction so that scan chunks/slices are assigned growing z-values. For the step-and-shoot model with no move during scan this still makes sense because the movement direction is the direction that long objects would be moved for multiple partial scans.

The detector coordinates are assigned so that the row that first sees the scan object have the highest coordinates. Looking from the source to the detector with that row at the top the columns have coordinates that grow to the right. Please refer to figure below for full axis details.



8.1 Detector Offset

The detector offset is used to compensate for situations where the detector is misaligned so that the center ray does not actually hit the center of the detector as it should. Using the figure from the Scanner Setup section a positive row offset means that the detector is really located above the expected position (i.e. it sees the object earlier than it should). Using the same figure a positive column offset means that the detector is located to the right of the expected position. Negative offsets similarly mean that the detector is actually below and left of the expected position. Obviously the directions described here are reversed if one instead looks at the center ray offset in relation to a fixed detector. All offsets are measured in pixels rather than absolute values like centimeters. That is, an offset of 0 means that the detector is perfectly aligned with the source. A row offset value of 1.5 means that the detector is one and a half pixel height above the expected position. NOTE: The column offset is only accurate for flat detector panels and may cause volume distortion for curved detector panels as the focal spot is no longer the center of the curve, as seen on the figure below.



9 Disclaimer of Warranty

The CT Toolbox is developed by computer scientist and although we strive to achieve correct reconstruction results it was never intended or certified for clinical use. Please use it at your own risk and with the warranty notes from the license in mind!

10 Troubleshooting

Basic installation is described in the main README file, but typical application execution problems are covered a bit more here.

10.1 ImportError cphct

- Do you get a Traceback with cphct ImportError when running Cph CT Toolbox applications?

  Did you remember to set the PYTHONPATH environment to the path where you unpacked the Cph CT toolbox? You need to either set that environment or install the Cph CT toolbox system wide as described in the README.

10.2 Alloc error

- If you get a memory allocation error when running the tools.

  You probably implicitly try to use more memory than your system has. Please try using a smaller chunk size.

10.3 Bus error

- If you get a bus error when running the tools.

  The disk containing your temporary directory is probably full, try setting the temporary directory to a disk with sufficient free space.

10.4 ImportError numpy or pycuda

- If you get an ImportError for the numpy or pycuda module when running the tools.

  You probably don't have numpy or pycuda properly installed. Please refer to the references for install instructions.

10.5 AttributeError for numpy or pycuda

- If you get an AttributeError for the numpy or pycuda module when running the tools.

  You probably don't have a recent enough numpy or pycuda version to support the required helper functions. Please refer to our requirements and to the references for install/upgrade instructions.

11 Questions and Feedback

Feel free to contact us [4] with your questions and feedback.

[4] The contact information is available in the README and on the project page online