# RTK-handler Documentation v0.1

Danny Lessio - danny.lessio@gmail.com

April 4, 2016

## Contents

## Part I

# What this software does

## 1 Short description

Given a .csv file containing the informations about the projections (projections angles, proj_offset_x/y, `Io` source intensity) and an .mha projection stack, RTK-handler helps with the creation of the RTK .xml geometry, with the normalization of the .mha stack and with the reconstruction using the rtkfdk procedure.

## 2 Detailed description

During the acquisition phase, the detector (for each projection) has absorbed the `I` intensity wich follows the Lambert-Beer law:

$$\mathtt{I} = \mathtt{Io} \cdot e^{-(\mu 1, \mu 2, ..., \mu n)} \tag{1}$$

Where:

- `I` = Attenuated intesity.
- `Io` = Source intensity.
- $e$ = Euler number.
- $(\mu 1, \mu 2, ..., \mu n)$ = Attenuation value.

With simple algebric operation we can get the attenuation value if we have the `Io` source intensity:

$$(\mu 1, \mu 2, ..., \mu n) = -\log_e I/Io \tag{2}$$

This process is also known as Normalization and the `Io` source intensity can be:

1. Manually saved for each projection:

   This must be done for each projection during the forward projection phase ( for example in a .csv file ). This is the best choice because each projection will not have an approximated intensity values and also it is always possible.

2. Automatically approximated:

   If the object size is smaller than the detector, some pixels of it will receive x-rays with no attenuation. The source intensity can be calculated by choosing a projection's pixel with the highest intensity value or with other techniques like taking the average of all the non attenuated pixels. This is not always possible.

The RTK library reads the stack of projections with a class called ProjectionsReader and from its documentation:

*"The universal projections reader of RTK (raw data converted to attenuation). Currently handles .his (Elekta Synergy), .hnd (Varian OBI), .tiff (Digisens), .edf (ESRF), .XRad. For all other ITK file formats, it is assumed that the attenuation is directly passed and there is no processing."*

So, RTK makes an automatic Normalization only for a specific set of projections types. Before reading the .mha stack with the ProjectionsReader class it must be normalized manually obtaining the `Io` intensity directly from a .csv file.

RTK-handler automates the Normalization process according to (2) and creates the .xml geometry according to this:
`http://www.openrtk.org/Doxygen/geometry.pdf`

It also provides an interface to the rtkfdk procedure in order to get a reconstruction from:

- Normalized stack: Simply reconstructs with rtkfdk.

  OR

- Non normalized .tiff set: It performs an auto-normalization and than reconstructs with rtkfdk.

The binary rtkfdk is simply executed by Python from the */path_of_RTK-bin/RTK-bin/bin* folder by choosing the .mha projections stack and the .xml geometry file directly by the folder structure generated by RTK-handler and asking the rest of the parameters directly from the user.

# Part II
# Input Requirements

## 3 .csv file

### 3.1 structure

The .csv file is required by the RTK-handler application. It must have the following structure:

| Projection Name | Angle | Projection offset x | Projection offset y | `Io` intensity |
|---|---|---|---|---|

The rows must not contain any title, ex: the position (0,1) must contain the first projection angle.

- Projection Name:

  All this column can be blank, RTK-handler fills it with an incremental name during the creation of the .xml geometry.

- Angle:

  The angle of each projection.

- Projection offset x/y:

  The offset from the detector origin (wich is assumed to be in the center) to the image origin (if not specified an .mha slice have the origin situated on the top left corner).

- Io intensity:

  The `Io` source intensity readed during the acquisition of each projection.

## 4 .mha projection stack

### 4.1 Structure

The .mha metaimage format is useful to handle n dimensional images. The internal structure is like:

< HEADER >
< RAWDATA >

The < HEADER > part contains metaobject that indicate how to read the < RAWDATA > part, and adds informations about dimensions.

### 4.2 Header

This Metaobjects are essential:

- NDims

  The number of dimensions, if there's more than one projection (surely there's more than one), this value will be 3.

- DimSize

  The dimension of the image in pixel.
  For example if "NDims = 1024, 512, 60" we have 60 pojections each with 1024x512 pixels (this is also the detector size in pixel).

- ElementByteOrderMSB

  If is set to "True", the raw data is MSB otherwise is LSB.

- ElementSize

  Single pixel size of the detector on u and v directions (in millimeters). Example: "ElementSize = 0.096, 0.096, 0.0"

- ElementType

  The type of data that each pixel holds.

- ElementDataFile

  For an .mha image this is set to "Local" and indicates that the raw data begins at the beginning of the next line.

For more informations about .mha Metaobjects visit:
`http://www.itk.org/Wiki/ITK/MetaIO/Documentation`

## 5 Additional informations

Those information must be known before the RTK-handler usage.

### 5.1 About acquisition infrastructure

- Source to Isocenter Distance
- Source to Detector Distance

### 5.2 About detector

- Single pixel size on direction u (in millimeters)
- Single pixel size on direction v (in millimeters)

### 5.3 About reconstructed object

- Voxel size on x, y, z directions (in millimeters)
- Reconstructed object size on x, y, z directions

# Part III
# Dependencies

## 6 Python 3 and Lua 5.1

### 6.1 Assert Python >= 3

Python 3 or greather must be the default Python interpreter on the system. To check this simply type:

```
python --version
```

### 6.2 Assert Lua 5.1

An installation of Lua 5.1 ( exactly this version ) must be installed in the system, but is not necessary that this is the default one. To check the default installation type:

```
lua -v
```

## 7 ITK and RTK

### 7.1 Auto compilation for Linux machines

This script fully automates the compilation process of RTK for a Linux machine. Maybe it can also work on Mac OS but it hasn't been tested yet: `https://github.com/dannylessio/auto-build-RTK`

### 7.2 Default compilation

Follow the 0-1-2-3 installation steps that can be found here:
`http://wiki.openrtk.org/index.php/Main_Page#Step_0_-_Getting_ITK`

## 8 SimpleRTK

### 8.1 Compilation

Open the terminal inside the RTK-bin folder, than run:

```
ccmake ../RTK
```

Turn ON both BUILD_SIMPLERTK and WRAP_PYTHON options, if one of those option is not visible press [t] to have a full view of all available options. After that press [c] to configure until the phrase "Press [g] to generate and exit" appears at the bottom of the terminal. When this shows up press [g]. Without changing directory execute:

```
make -j <NUMBER OF CORES>
```

If the compilation process is terminated with no errors, install the .egg package inside the Python3 environment:

```
cd SimpleRTK-build/Wrapping/PythonPackage
sudo python setup.py install
```

## 8.2 Troubleshooting

If the compilation with make is terminated with a "pkg_resources" error, replace default ez_setup.py inside:
RTK-bin/SimpleRTK-build/Wrapping/PythonPackage/ez_setup.py

With this:
`https://bitbucket.org/pypa/setuptools/raw/bootstrap/ez_setup.py`

Than remake RTK.

# 9 SimpleITK

## 9.1 Installation from precompiled source

Before start with an heavy compilation process it is possible that there's a matching distribution available on PyPi (for Windows this may work) :

```
pip install SimpleITK
```

If there are some problems, proceed with the compilation from source.

## 9.2 Compilation

This compilation requires high (more than 4Gb) amount of memory, so make sure to have a swap partition enabled.

First, clone the lastest source available from github:

```
git clone https://github.com/SimpleITK/SimpleITK
```

Without entering it, make a folder called `SimpleITK-bin`

```
mkdir SimpleITK-bin
```

Enter inside SimpleITK-bin and run ccmake:

```
cd SimpleITK-bin
ccmake ../SimpleITK
```

Now, press [t] to have a full view of all the available options and fill the relevant options like in this example:

```
CMAKE_BUILD_TYPE         Release
ITK_DIR                  /UniFE/ITK-bin
SITK_LUA_EXECUTABLE      /usr/bin/lua5.1
SWIG_EXECUTABLE          /usr/bin/swig
WRAP_PYTHON              ON
```

After that, press [c] to configure and if there are no errors, press [g] to generate and exit. Now execute (this requires a long time):

```
make -j <NUMBER OF CORES>
```

If the compilation process is terminated with no errors, install the .egg package inside the Python3 environment:

```
cd Wrapping/Python/Packaging
sudo python setup.py install
```

If the path is correct, RTK-handler exits with a "configuration success" message and the configuration is done.

# Part IV
# RTK-handler

All the following section must be executed in order.

## 10 Installation and configuration

### 10.1 Installation

Simply install it by using the pip package manager:

```
pip install RTK-handler
```

### 10.2 Configuration

As a user without root privilege open the terminal and execute:

```
RTK-handler -p
```

Now RTK-handler asks to insert the full RTK-bin path like this:

```
/UniFE/RTK-bin
```

## 11 Create the folder structure

Create a folder that holds all your projections file:

1. The non-normalized .mha stack
2. The csv file

First open the terminal and enter to this folder, than execute:

```
RTK-handler -s
```

This will create the required folders structure and leaves you with this structure:

```
│ csv
├── geometry
├── projections
│   ├── non_normalized
│   │   ├── mha
│   │   └── set_of_tiff
│   └── normalized
│       ├── mha
│       └── set_of_tiff
├── reconstructions
│   └── from
│       ├── mha_normalized_proj
│       └── tiff_without_norm_proj
├── your_file.csv
└── your_stack.mha
```

After that move your_file.csv inside the csv folder and your_stack.mha inside projections/non_normalized/mha folder.

## 12 Generate the .xml geometry

From the top folder execute:

```
RTK-handler -g
```

And follow the instructions, the program will ask for:

- Single pixel detector size on u direction
- Single pixel detector size on v direction
- The Source to Isocenter distance
- The Source to Detector distance

Than it search for the .csv file inside the csv folder and creates the geometry.xml file inside the geometry folder. The first column of the .csv file is updated with an incremental name.

## 13 Normalize the .mha stack

From the top folder execute:

```
RTK-handler -n
```

RTK-handler first separate the .mha stack into a set of .tiff images and put them inside projections/non_normalized/set_of_tiff folder. For each saved image it reads the corresponding `Io` value inside the .csv column, it makes the normalization according to (2) and stores the normalized set inside projections/normalized/set_of_tiff folder. At the end it gather this last set inside a single normalized .mha stack.

## 14 Rtkfdk reconstruction

From the top folder execute:

```
RTK-handler -r
```

This begin an rtkfdk recostruction, the procedure asks for 2 options:

1. Reconstruct from normalized .mha

   It stores the result inside:
   reconstructions/from/mha_normalized_proj/

2. Reconstruct from non normalized tiff set

   It stores the result inside:
   reconstructions/from/tiff_without_norm_proj/

After that the procedure will ask those informations:

- Spacing on x,y,z directions (voxel size on x,y,z)
- Dimension of the reconstructed object on x,y,z directions

Now the RTK-handler execute the RTK-bin/bin/rtkfdk and stores the .mha reconstructed stack inside the folders previously described.

## 15 Clean the folder structure

This command is useful in order to clean some disk space, it removes all the content inside:

```
projections/non_normalized/set_of_tiff
projections/normalized
```

But leaves all the relevant file: .csv file, geometry.xml, .mha non normalized projections and the reconstructed stack. It also deletes the previously created folder structure with "RTK-handler -s".