CCPi Core Imaging Library Documentation

Release 1.0

CCPi

Jul 11, 2019
1 Introduction 3

2 Pre-Processing Module 5
   2.1 Beam Hardening Correction (CarouselFit): ................................. 5

3 Reconstruction Module 17
   3.1 Installation ................................................................. 17
   3.2 CCPi Iterative Reconstruction Algorithms ..................................... 17

4 Quantification Module 21
   4.1 Installing software .......................................................... 21
   4.2 Algorithms ..................................................................... 21

5 Segmentation Module 29
   5.1 Simpleflex Segmentation Algorithm .............................................. 29

6 Viewer 39
   6.1 Usage ............................................................................ 40
   6.2 Screenshots ..................................................................... 40

7 Contribution 47

8 Developer’s guide 49
   8.1 Build and Test Automation ......................................................... 49
   8.2 Essential Tools .................................................................. 55
   8.3 Building with Conda .............................................................. 58

9 Changelog 61

Bibliography 63
CCPi Core Imaging Library (CIL) is a library of algorithms that are used in Computed Tomography (CT). CCPi project is funded by EPSRC UK for supporting the CT community. This library provides modules for several steps involved in the CT. The algorithms are contributed by the community and the code is re-engineered by the CCPi funded staff to make them run faster and maintainable.
The CCPi Core Imaging Library (CIL) consists of five main modules. These are part of work flow for the CT based experiments data analysis. The five main modules are:

- Pre-Processing
- Reconstruction
- Segmentation
- Quantification
- Viewer

The pre-processing module provides the methods for modifying the raw data from the instruments before performing the reconstruction, for example: Beam hardening for Lab based instruments (cone beam).

The Reconstruction module provides the methods for reconstructing the volume using the projection data. Currently this module consists of only iterative methods. There are several FBP based methods in other libraries, which is not provided in CIL.

The Segmentation module provides the methods to extract or segment the reconstruction volume with the regions of interest. Currently there is only a topology based segmentation method available.

The Quantification module provides the methods for quantifying the segmented volume.
2.1 Beam Hardening Correction (CarouselFit):

Dr. Ronald Fowler STFC Rutherford Appleton Laboratory

2.1.1 Abstract

This document is a brief user guide to the Python software package CarouselFit. This software takes X-ray image data for a number of known samples, and fits them to a model of the beam hardening process which occurs when a broad spectrum source is used to image a sample. This model can then be used to generate corrections appropriate for a single material to convert the observed attenuation values into the actual attenuation that would be observed for that material with monochromatic X-rays at a given energy. This software is based on the IDL package and ideas described in [DEM13].

2.1.2 Introduction

Beam hardening is well known problem that is described in many works, see [DEM13]. Lab based X-ray Computational Tomography (CT) machines use X-ray tubes as their source of radiation. The voltage to the tube can be varied to control the maximum X-ray energy generated, but the actual output is composed of the broad spectrum Bremsstrahlung component and characteristic peaks. The X-ray attenuation of materials (in general) decreases as the energy increases. This means that as a broad spectrum beam passes through a target the low energy X-rays will be absorbed first leaving a beam that has increasing mean energy. It is this effective hardening of the beam that means that a plot of the log of the attenuation against material thickness is not linear, as would be the case if a monochromatic X-ray source was used. Since CT reconstruction algorithms assume that this linear relationship holds artifacts are generated in the final image due to the beam hardening.

To reduce beam hardening artifacts it is usual to pre-filter the X-ray beam with the built-in filters available on CT machines. While this does reduce the problem, it can not fully solve it since filtering to get close to monochromatic beam would leave such a weak high energy beam as to make imaging impractically slow. Hence a compromise has to be made with sufficient filtering to reduce beam hardening effects while allowing enough X-rays through to image the sample in reasonable time. The figure below compares the 80KeV spectra for an X-ray tube with no added filtering
and with 1mm Cu filtering. The Cu filter reduces the energy range, but still leaves a polychromatic beam. Plots have been normalised, but the Cu beam will have lower intensity than unfiltered one at all points.

![Graph showing intensity vs. X-ray energy](image)

If the energy distribution of the filtered X-ray beam is known, along with the energy dependent attenuation of the sample, then it is possible to apply a correction to the measured attenuation so as to remove beam hardening artifacts. This can be done using the techniques described in [DEM13] which are implemented in this software package.

This software takes as input a number of images of well characterised samples and uses these to fit a simple model of the expected beam hardening (BH) to the observed data. The result is an estimate of the “response function”, R(E), which gives the expected output signal from the detector as a function of X-ray energy for the selected combination of X-ray source, voltage, filters and detector.

Note that due to aging effects of the X-ray source, detector, etc., this function may change over time, so ideally the calibration measurements should be made before and after each CT scan. In addition the model allows for variation in the form of R(E) with the number of the scan line. This can occur due to the way the emitted X-ray spectra is known to depend on the “take-off” angle [DEM13]. Use of pre-filtering the X-ray reduces both beam hardening effects and the variation of these with take-off angle. Low energy X-rays show the greatest variation with take-off angle. Davis recommends using both pre-filtering as well as the software correction described in this guide to best minimise beam hardening artifacts.

Using the fitted response function of the system it is then possible to determine a correction curve that will map from the observed attenuation to the true attenuation that would be seen at a given monochromatic X-ray energy. This correction curve is calculated assuming the sample is composed of a single material type for which the X-ray attenuation coefficient can be determined, using a program such as XCOM [xcom]. This allows for compound materials, as long as the composition is constant. In the case of samples made of more than one compound the correction curve will only be applicable if one material is the dominant absorber and corrections made to that material.
The attenuation correction curve for a material depends on the shape of the energy dependent attenuation within the range of X-ray energies that are present after filtering. If two materials have a near constant ratio of attenuation in this energy range then their correction curves will be similar. However if one has a sharp step in attenuation in this range (a K-edge) while the other does not, then correction curves can be very different. This method is designed for samples of a single material or where one material is the dominant component. The figure below compares the corrections curves for three materials, Ca-hydroxylapatite (brown), aluminium (purple) and tin (blue) with some pre-filtering at 80KeV. The first two have similar correction curves, particularly at low attenuation. The tin curve is different at all attenuations due to a k-edge that causes a jump in its attenuation at about 30KeV.

The next section describes how to download and run the software. The following section describes how to set up the necessary files that give information on the number and type of test images that are used for calibration and the image formats. Details are then given on how to the run the fitting and post-processing image modules of the software. The first of these fits the model to the data while the second applies the correction directly to the CT image data. An alternative to processing all the images is to generates a look-up table or a 4th order polynomial to map observed attenuation values to mono-chromatic attenuation. Such data can be used as input to some reconstruction software, like the standard XTek filtered back projection package and, in future, to the CCPi CGLS iterative code.

2.1.3 Downloading and running the software

Installing the binary

If you have a Python distribution from Continuum(Anaconda https://www.continuum.io/downloads) then you can install the binary package from the CCPi Anaconda channel (https://anaconda.org/ccpi/). To do issue the following at a command prompt:

2.1. Beam Hardening Correction (CarouselFit): 7
conda install -c ccpi -c conda-forge ccpi-preprocessing numpy=1.12

This will provide the executables for running the CarouselFit.

CarouselFit

A set of example data to illustrate use the fitting software can be downloaded from the CIL git repository.

Installing from Source

The software is available from the CCPForge repository. It consists of a Python software package along with a
number of data files that are used to help model the X-ray beams and the material attenuation. As well as a Python
environment the software depends on a number of additional packages being available. An easy way to access most
of the required packages is to download the Anaconda Python environment which is available for Linux, MacOS and
Windows systems from https://www.continuum.io/downloads. The software has been developed using Python version
2.7, though it should also run with Python 3 as well. It is recommended that the user installs this before installing the
CarouselFit software. Alternatively the user may install the required packages in their local Python installation, if they
are not already available. The main Python modules that may need to be added to a local installation are:

- numpy - needed for array operations
- matplotlib - needed for plotting
- scipy - needed for optimization
- tifffile - needed for working with tiff images (install from conda-forge channel by: conda -y -c conda-forge
tifffile)

The CarouselFit software can be checked out to a suitable directory using the command

svn co https://ccpforge.cse.rl.ac.uk/svn/tomo_bhc/branches/release01 carouselFit

This will create a set of three directories under carouselFit:

- src: this contains the Python source code
- doc: this contains documentation of the software
- test: this contains several sub-directories with information on attenuation and X-ray spectra. The source code
must be executed from this directory and any updates to the carousel or crown information should be made in
the carouselData sub-directory.

Running a test case

After downloading the software the installation can be checked by running the software in the test directory and reading
an example script file.

If the software was obtained from SVN this could be done from a command prompt by typing:

cd test
python ../src/runCarouselFit.py
read script.full
quit

Note that it may be useful to look at the graphs plotted by this process before using the quit command, since these
disappear when the program stops.
If the conda installation method was used then it would be necessary to unzip the test data into a folder. The test data is available from git hub as:

https://github.com/vais-ral/CIL-Docs/raw/master/test/bhctestdata.zip

When this has been downloaded and unzipped, get a command prompt in that folder and type the lines:

CarouselFit
read script_demo.txt
quit

This set of commands should run without generating any error messages, such as failure to import modules. If missing modules are reported it will be necessary to add these to the Python system and run the test script again. Check the documentation for your Python system to see how to add modules.

The file script_demo.txt illustrates a simple fitting using some calibration data obtained from QMUL. Lines starting the the # character are comments. The first command in the script loads the files that define the test samples, the imaging voltage and the images themselves. These files, which are explained in more detail later, are:

carouselData/carousel0.data
carouselData/run001.data
carouselData/run001.img

The images that are to used in the calibration are shown by the showimg command. The script then sets the material and the X-ray energy to be corrected to with the setcormat command. Then two fits are done, the first allowing for some variation in the fit with the line number in the image. The second fit does not allow for this variation and assumes that one correction function can be used for all lines. The final fit returns a 4th order polynomial that can be used to correct the reconstruction of the image in software such as that used on Xtek machines. The vales of these coefficients are saved in the file param.log in the same folder that the fitting process was run. For this example data the values should be:

xpoly:
  0  [ 0.00282561 -0.04733048 0.45413055 0.48893604 0. ]

The terms in the square brackets correspond to $X_4, X_3, X_2, X_1, X_0$ in the input used to an XTek reconstruction.

### 2.1.4 Configuration files

The original calibration device described in [DEM13] was called a carousel as it was built from a set of 9 test samples arranged between two circular supports allowing for each of the samples to be imaged individually by the scanner. The samples would cover the full range of lines in the scanner, but not the full range of each row; typically only the centre half of each row would be covered by the sample.

A more recent calibration device has been developed at staff at the Research Centre at Harwell (RCaH) which is known as a crown. This device allows a larger number of samples to be mounted. In this case the sample usually covers all lines and rows of the image.

**Carousel sample definition file**

The materials mounted on the carousel, or crown, must be described in a simple ASCII file which is stored in the test/carouselData directory. An example of the format that was used for the carousel from QMUL is shown below.

```bash
# carousel definition file based on data from QMUL 17/11/14
10
Cu,Ti,Ti,Ti,Al,Al,Al,Al,Al,NOTHING
```

(continues on next page)
This illustrates a case where there are 9 sample materials in the carousel. In this case all the samples are pure metals of known thickness and density. It is important to emphasize that the calibration depends on the sample materials being very well characterised. If a large error exists in either the thickness or purity of a sample this can undermine the accuracy of the fitting process. No exact guidelines have yet been defined on the best set of test materials to use, but obviously samples of the material the forms the dominant absorber in the imaged target would be ideal. However, this is often not practical in many cases, such as bone and teeth studies, where calcium metal is the prime absorber, but samples of the pure metal are subject chemical reactions in air. As long as the energy dependence of the sample attenuation coefficient, $\mu(E)$, is not too different to that of target dominant absorber then the calibration method should work. Some possible problems may occur if the sample has sharp steps in $\mu(E)$ due to band edges that lie in the response range of the system which are not seen in the target material. For example, compare the attenuation of Sn with that of Ca in the range 0 to 75KeV.

The above file uses the simple format:

- a comment line, starting with #, to describe the file
- a single integer giving the number of sample materials plus 1
- a set of comma separated strings giving the names of each sample, with no spaces. the number of names must be the same as the previous number, with the final one named “NOTHING”. In this case the samples are all pure metals and the chemical symbol has been used as the name. However any name be used as long as a corresponding file with the extension .txt exists in the directory test/xcom. This file gives the energy dependent $\mu(E)$ for this sample in steps of 0.5KeV from 0 to the maximum expected energy.
- line4: a set of comma separated values giving the density (in g/cm3) of each sample. A dummy value of 1 is used for the final material.
- line5: a set of comma separated values giving the thickness of each sample in cm. A dummy value of 0. is added on the end.

If a sample type other then the ones already described in test/xcom is used it is necessary to create a file of the attenuation values of that sample. See the Readme file in that directory for details.

The thickness range of the samples should aim to cover the range of attenuations that are expected in the test sample.

Sample image data file

In addition to a description of the samples in the carousel it is also necessary to define the format of the sample images and details of the X-ray source, filters and detector. This is done via another file in the directory test/carouselData which has the default extension .data. One such file must be generated for each calibration case, while the above carousel definition file will only change if the samples are changed.

Again a simple ASCII format is used to define the necessary values. An example is shown below:

```
# data for one QMUL calibration run
80   # voltage
22   # take of angle [not used by default]
W    # target material
19.25 # target density
600  # image res rows
800  # image res lines
carouselData/run001.img    # image file
float32 # data type in image file
2    # number of filters
```
The format has one value per line with a comment to describe the value. Most of these are self-describing, such as the accelerating voltage, the take-off angle, the target material (tungsten, W) and its density, for the X-ray source.

The path to the file containing the sample images must be included in this file. All the images must currently be in a single file. The format used above, float32, assumes a binary format with 9 separate images of $600 \times 800$ 32-bit floating point values. Each value is $\log(I_0/I)$ for that pixel with flat/dark field corrections.

Another supported format is uint16. In this case the sample images values are unsigned 16 bit values of the $I$ value. Again these are all packed in order in a single file. The first image of the file is the (shading corrected) flat field image. The $I_0$ value is taken as the average of this initial image.

A variation on uint16 format, which is slightly more compact, is labelled as uint64_65535. This format is again unsigned 16 bit images, but it assumes that the data has been corrected for flat and dark fields and that it has been normalised to a white level of 65535. This means that the raw binary file no longer needs an initial image giving the white level. This is the format that is generated by the Python script average_mat.py which converts tif image files into this format. See Appendix A for details of using this program.

Usually a set of filters are used to limit the energy range of the X-ray beam. In the case of the QMUL data they normally employ two filters with 0.12cm of Al and 0.1cm of Cu, as shown in the above file. As the fitting process includes varying the exact Cu filter width it is recommended that a zero width Cu filter element is included even if no Cu was used in the actual imaging.

The definition of the detector material is important and tests to date have been made with CsI. However other materials may be used if their attenuation profile is included in the text/xcom directory. Since the width of the detector maybe used as a fitting parameter it is not essential to specify an accurate value, though this will be used in the command showspec, if it is run before a fit has been performed.

2.1.5 The command line interface

Command list

When the Python software is started from Python or a similar environment, a simple command prompt is issued. Typing help will give a list of the available commands.

The commands are:

- **read filename** This command opens the given file and reads commands from it until end of file. Control is then returned to the command line. Comment lines start with #. Do not include blank lines in the command script.

- **load file.def file.data** This reads the definition file for the carousel and the data relating to the actual calibration images. These two files must exist and are described in the previous section. they are normally located in the test/carouselData directory. This is usually the first command to issue since most others need this data to be present.

- **quit** Exit the program.

- **help** Give a list of available commands.

2.1. Beam Hardening Correction (CarouselFit):
• **showcor** [ll l2…] This command will plot the attenuation correction curve for any one or more lines. If no arguments are given it will plot the first, middle and last correction lines. The matplotlib zoom feature can be used to focus on a particular region of the plot. It can only be used after a fit has been performed. The correction is shown in the space of log(I0/I).

• **showimag** This command will plot the images of each sample in one window. It may be useful to check for problems with the samples. It can only be used after data has been loaded.

• **fitatt nlines [linestep]** This command attempts to fit the model to the selected samples (see mask command). The number of lines of data to fit must be given. This may be followed by a “step”, e.g. 10 to use every 10th line. This can be useful when using many lines as fitting all of them can be very slow and the fit may not be improved using more data. The time to fit also increases with the number of variables that have been selected with the “vary” command. Fitting to a few lines can be a good way to see if the model fits and give a better initial guess for a fit to a larger subset of the data.

• **vary** [target|detector|filter|energy|spectra npoly] On its own this command lists the order of polynomial used in fitting the line wise dependence of each of the three main parameters, **target width**, **detector width** and **filter width**. The setting “-1” indicates that the value should be held constant, as set by the initguess command. Using “0” indicates the value will be fitted, but is independent of the line number. Setting to “1” gives a fit allowing a linear variation of the value with line number. For example: vary filter 0 will allow a single fitted value for the filter width. vary detector -1 will keep the detector width constant. vary target 1 allows the target width to vary linearly with the line number. The fit time increases significantly with the order used and values greater than 1 are not recommended. An experimental option is to allow extra terms to be added to the normally linear dependence of the detector response to the photon energy. Note that energy dependence is NOT related to line number in this case. However this polynomial is not constrained to be positive and the fit may fail. Keeping energy variation off (-1) is recommended. The final option called “spectra”, which defaults to 0, i.e. on, when no pre-defined spectra are present, which is the case for the open source release of the package. Setting spectra to 0 causes the calculated spectra to be modelled as a simple non-symmetric Gaussian form with 3 parameters, **peak**, **inverse left width** and **inverse right width**. If pre-computed spectra are available, e.g. from spekCalc, these can be used in preference to the Gaussian by setting vary spectra -1.

• **initguess** [s1 s2 s3 [ s4 s5 s6 s7 ] ] Set the initial guess to be used by fitatt. s1 is the width of the target filter (usually tungsten), s2 is the log width of the detector (usually CsI) and s3 is the width of the fitted filter (usually copper). Commonly used values for the initial guess are 0.01 -6.0 0.01. If using the experimental feature “vary spectra 0” then additional values can be given which are the initial value of the energy term (should be zero) plus the Gaussian centre and widths, e.g. 0.01 -6.0 0.01 0.0 40.0 0.05 0.05. When loading data the Gaussian peak is set to half the maximum X-ray energy. Using this command with no parameters gives the current settings on the values.

• **mask** [n1 n2..] Without arguments this shows the set of masks that control if a given sample will be used in the next fit operation. By default all values are true which means that sample will be used in the fit. Samples are labeled from 1 to n and to mask the m sample that number should be given as an argument to the mask command. A negative value can be used to unmask a previously masked sample.

• **setcormat** material energy This command must be used before a fit operation to define the material and energy to which the correction curve should be determined. For example setcormat Al 40 sets the correction curve to be calculated for Aluminium at 40KeV. At present if the correction material or energy are altered it is necessary to rerun the fit command.

• **transform** This is an experimental command which will be removed in future.

• **showspec** [line] - plot three spectra, the input X-ray spectrum, the filtered spectrum and the response spectrum. Should only be used after a fit has been made. This command needs improving since the “filtered” plot is not meaningful. Also the printed attenuation values are not useful since these are not fitted to. If a line number is given, plots are for that line. The default is line 0.

• **showatt** [nsamp nline] - plot the sample attenuations along a specific line. By default this shows the attenuation for all samples at line 400. Samples are labeled 0 to $n-1$ in this case.
• **debug** - set debugging option, for diagnostic purposes only.

• **showconf** - list some of the settings, such as the filters, detector, source and voltage.

• **setwidth {width}** - without arguments, prints the width, in pixels, used to average over each line to get the mean attenuation. For the QMUL data, where the sample does not cover the whole image, it is important to ensure this does not exceed the true sample width. For the RCaH data, where the image does cover the whole width, a larger value can be used.

• **setfilter {material width}** - without arguments lists the filters defined. Can also be used to change the width of existing filters, though not add new ones. Used for debugging.

• **setoptions {solver=old|new}** - set option. Currently only allows switching between old and new least square solvers in scipy. The old version is more widely available and is the default.

### Using the software

### Getting the image file

As stated previously it is necessary to write the definition file (.def) that describes the carousel or crown and the particular test case that is being treated (.data). The latter file must also point to the data file that contains the sample images in a suitable format. It is assumed that corrections for dark and flat field images have been applied to the images before they are passed to the software. The way in which these are generated will vary with the X-ray CT machine used to gather the images. For the case of an XTek machine an example python script has been written to illustrate one way in which suitable calibration data can be obtained from the images of the test samples. This file is called `average_mat.py` and can be found in the `src` directory from the svn checkout. It can be run from the command line, in a similar manner to the fitting program, though it is not interactive. It takes a number of arguments, such as:

```
python ../src/average_mat.py dir_list.txt images.raw
```

The program requires an input file `dir_list.txt` which contains an ordered list of directories with images of the dark and flat field and projections of the samples. Current practise is to take 11 separate projections of each of the dark field, the flat field and every sample in the crown. Each file is a 16 bit unsigned image in tiff format. Thus the first directory will have 11 tiff images of the dark field, and so on. If there are only three test samples (typically there would be more) then `dir_list.txt` might be:

```
c:/Images/crown01/darkfield
c:/Images/crown01/flatfield
c:/Images/crown01/Al0.1
```

Each directory contains a number of tiff images which are averaged over and then used to calculate the normalised attenuation image for each material sample in the crown. This data is then written as the raw file `images.raw`. This can be copied to the carouselData directory and used as input to a fit run. The first two directories are assumed to be the dark and flat fields while the rest are the materials in the crown, in this case are aluminium at 0.1mm, 0.2mm and 0.5mm. The order of the samples in this list must follow the order that is written in the `.def` file describing the samples. The format of the output produced by this script must be given as `uint64_65535` in the `.data` file.

The `average_mat.py` script also allows selection of a sub-region of the image. This can be useful if just doing a fit to a certain region of the image. Four extra parameters can be given for the minimum and maximum points to output. This reduced image size then needs to be set in the configuration file to describe the data.
Running the fitting process

Once the sample image file has been produced and the two data files describing the test samples and the X-ray conditions written, then the-carouselfit program can be run. This is a command line program and the user can either type the commands at the prompt or read them from a previously prepared file.

A simple partial analysis might consist of the following steps which loads the information about the run and displays the images:

```
load carouselData/carousel0.def carouselData/run001.data
showimg
showatt
```

When the program reads the image data and the sample descriptions from the files, it performs a basic check to see that the observed attenuation is consistent with the stated voltage and thickness. This helps to catch simple unit errors, or mistakes in the sample order. The following commands can then generate a fit to the supplied sample data:

```
setcormat CaHydro 40
vary target 0
vary detector -1
vary filter 0
initguess .01 -6 .01
fitatt 800 20
showspec
showcor
```

These commands then set the material and energy to which we wish to correct the data via the setcormat command, and then alter the default orders of the fit variables. The fitatt command fits the given initial guess across all the lines of the image data (800) but only using every 10th line, which is adequate in most cases. This fit may take around a minute to run. Finally the fitted spectrum and correction curve is plotted. Note that in this case there is no variation of the fit with line number so only a single correction curve will be generated.

The correction curves are stored in the same format as used in the earlier IDL code as separate 8th order polynomial fits to the correction data in a file called polyfit.npz. These curves are the ones shown by the showcor command above. To actually apply the correction to image data requires the use of another Python program, applyTrans.py. In addition to the above 8th order polynomials, 4th order fits are also written to the output file param.log. The 4th order polynomial values are written at the end of the file, one set per line if the solution includes variation with line number. These values can be used in the xtekct file for the parameters X0 to X4, X0 being the rightmost value in param.log. If the variation of the correction with the line number is significant it would be better to correct each project individually as described in the next section.

applyTrans.py

The Python script applyTrans.py can be used to update image files using the correction curves calculated by the above fitting process. It can also calculate a file of type .bht which can be used by XtekCT machines to correct the image data used in CT analysis. In latter case only one correction curve is applied to all the data, in the same way that using the using the 4th order polynomial fit does.

The syntax of the command can be seen using the -h option, which gives:

```
applyTrans.py [-r rows -l lines -p poly.npz -w whiteLevel -x file.bht]
[-d] [file1.ext] [filen.ext]
```
In the above data it is usually necessary to specify the image size in rows and lines. If all the image data is stored in a single file with data type float32, as used for some data from QMUL, then the following command can be used to process it:

```
python ../src/applyTrans.py -r 600 -l 800 images.raw
```

In this case the default file `polyfit.npz` is read to find the correction curves. If 800 curves are present then one will be applied to each line in the image. If only one correction curve is present then this one correction will be used on all image lines. The processed output will be written to `bhc_images.raw`. Note that the `whiteLevel` parameter is not needed in this case as the `.raw` extension is taken to imply float32 data of $\log(I : sub : A')$.

To generate a `.bht` correction file the following command can be used:

```
python ../src/applyTrans.py -b -x xtekct.bht -w 59200
```

In this case only the file `xtekct.bht` is generated. It is necessary to provide an accurate estimate of the white level since any pixels above this are mapped to no attenuation.

References

2.1. Beam Hardening Correction (CarouselFit):
3.1 Installation

CCPi Reconstruction codes are only available as binary distribution. Anaconda distribution of python is required. Download and installation instructions are available at https://www.continuum.io/downloads. Please follow the instructions below for installing CCPi Reconstruction codes from anaconda channel.

```
conda install -c ccpi -c conda-forge ccpi-reconstruction numpy=1.12
```

3.2 CCPi Iterative Reconstruction Algorithms

There are three main iterative reconstructions available in this package. They are

- Conjugate Gradient Least Squares (CGLS)
- Maximum Likelihood Estimation Method (MLEM)
- Simultaneous Iterative Reconstructive Technique (SIRT)

In addition to the above methods there are three more variants of CGLS available in this package. They are

- CGLS with Convolution
- CGLS with Tikhonov regularization
- CGLS with Total Variation Regularisation (TV)

These reconstruction algorithms currently only work on the parallel beam datasets. The next version will have a support for iterative reconstruction algorithms for cone beam datasets.

3.2.1 API

The Python wrapper for the CIL uses numpy arrays as medium to pass data to and from each algorithm.
ccpi.reconstruction.parallelbeam.alg.cgls(normalized_sinogram, projection_angles, →center_of_rotation, resolution, number_iterations, threads, isPixelDataInLogScale)
ccpi.reconstruction.parallelbeam.alg.mlem(normalized_sinogram, projection_angles, →center_of_rotation, resolution, number_iterations, threads, isPixelDataInLogScale)
ccpi.reconstruction.parallelbeam.alg.sirt(normalized_sinogram, projection_angles, →center_of_rotation, resolution, number_iterations, threads, isPixelDataInLogScale)
ccpi.reconstruction.parallelbeam.alg.cgls_conv(normalized_sinogram, projection_angles, →center_of_rotation, resolution, number_iterations, threads, regularize, →isPixelDataInLogScale)
ccpi.reconstruction.parallelbeam.alg.cgls_tikhonov(normalized_sinogram, projection_→angles, center_of_rotation, resolution, number_iterations, threads, regularize, →isPixelDataInLogScale)
ccpi.reconstruction.parallelbeam.alg.cgls_TVreg(normalized_sinogram, projection_→angles, center_of_rotation, resolution, number_iterations, threads, regularize, →isPixelDataInLogScale)

:parameters:
normalized_sinogram: a numpy array with float32 values representing a sinogram. This can be in -log() values, if so then set the isPixelDataInLogScale to True.
projection_angles: a numpy array with float32 values representing angles in degress.
center_of_rotation: a double value specifying the center of rotation.
resolution: an integer values representing the resolution of number of pixels per voxel.
number_iterations: an integer value representing the number of iteration the algorithm need to run.
threads: an integer value representing the number of CPU cores to use for computation.
isPixelDataInLogScale: an boolean value representing whether the sinogram values are in -log() or not.
regularize: an double value representing the regularization factor.

returns: an numpy array with float32 representing an reconstructed volume.

3.2.2 Example

Let us go through an example that will reconstruct the dataset that is available at the savu GitHub repository.

In the following we go through an example. First we must run the imports:

```python
from ccpi.reconstruction.parallelbeam import alg
import numpy
import h5py
```

After imports one should load the dataset. The pre-processing, i.e. load the nexus (hdf5) file, extracting the angles and image data, scaling to 0-1 scalar range are done within the load_data function.

```python
# 1) load a dataset:

# This dataset is freely available at
# https://github.com/DiamondLightSource/Savu/blob/master/test_data/data/24737_fd.nxs
filename = "24737_fd.nxs"
norm, angle_proj = load_data(filename)
```

Now one can pass the data to the 6 different reconstruction algorithm that are available within the CCPI reconstruction package. One should set
1. the center of rotation,
2. the number of iterations,
3. the number of CPU threads that one wants to use,
4. regularisation parameters (only for CGLS_CONV, CGLS_TIKHONOV, CGLS_TVregularization)

```python
# Data can now be passed to the reconstruction algorithms:
# CGLS, MLEM, SIRT, CGLS_CONV, CGLS_TIKHONOV, CGLS_TVregularization

# center of rotation
center_of_rotation = numpy.double(86.2)

# resolution
resolution = 1

# number of iterations
niterations = 15

# number of threads
threads = 3

# CGLS
img_cgls = alg.cgls(norm, angle_proj, center_of_rotation, resolution, niterations, threads, isPixelDataInLogScale)

# MLEM
img_mlem = alg.mlem(norm, angle_proj, center_of_rotation, resolution, niterations, threads, isPixelDataInLogScale)

# SIRT
img_sirt = alg.sirt(norm, angle_proj, center_of_rotation, resolution, niterations, threads, isPixelDataInLogScale)

# CGLS CONV
iteration_values1 = numpy.zeros((niterations,))
img_cgls_conv = alg.cgls_conv(norm, angle_proj, center_of_rotation, resolution, niterations, threads, iteration_values1, isPixelDataInLogScale)

# Regularization parameter
regularization = numpy.double(1e-3)

# CGLS TIKHONOV
iteration_values2 = numpy.zeros((niterations,))
img_cgls_tikhonov = alg.cgls_tikhonov(norm, angle_proj, center_of_rotation, resolution, niterations, threads, regularization, iteration_values2, isPixelDataInLogScale)

# CGLS Total Variation Regularization
iteration_values3 = numpy.zeros((niterations,))
img_cgls_TVreg = alg.cgls_TVreg(norm, angle_proj, center_of_rotation, resolution, niterations, threads, regularization, iteration_values3, isPixelDataInLogScale)
```

One may want to compare the results of the reconstruction algorithms:

3.2. CCPI Iterative Reconstruction Algorithms
# 3) Visualize a slice of the reconstructed images

```python
import matplotlib.pyplot as plt
fs = 10
fig, ax = plt.subplots(1,6,sharey=True)
ax[0].imshow(img_cgls[80])
ax[0].axis('off')  # clear x- and y-axes
ax[0].set_title("CGLS", fontsize = fs)
ax[1].imshow(img_sirt[80])
ax[1].axis('off')  # clear x- and y-axes
ax[1].set_title("SIRT", fontsize = fs)
ax[2].imshow(img_mlem[80])
ax[2].axis('off')  # clear x- and y-axes
ax[2].set_title("MLEM", fontsize = fs)
ax[3].imshow(img_cgls_conv[80])
ax[3].axis('off')  # clear x- and y-axes
ax[3].set_title("CGLS CONV", fontsize = fs)
ax[4].imshow(img_cgls_tikhonov[80])
ax[4].axis('off')  # clear x- and y-axes
ax[4].set_title("Tikhonov", fontsize = fs)
ax[5].imshow(img_cgls_TVreg[80])
ax[5].axis('off')  # clear x- and y-axes
ax[5].set_title("TV Reg", fontsize = fs)
plt.show()
```

## 3.2.3 The whole demo code

The complete demo can be downloaded at Reconstruction CGLS Demo. To run the demo you need to download the sample dataset and install the following packages

- matplotlib
- h5py
4.1 Installing software

CCPi Quantification codes are only available as binary distribution. Anaconda distribution of python is required. Download and installation instruction are available at https://www.continuum.io/downloads. Please follow the instructions below for installing CCPi Quantification codes from anaconda channel.

```
conda install -c ccpi -c conda-forge ccpi-quantification numpy=1.12
```

4.2 Algorithms

4.2.1 Accessible Volume

The algorithm takes in a binary 3D image (Usually a scaffold images) and calculates the accessible volume for a range of sphere sizes. The algorithm also takes in input mask which defines the boundary of the volume from where the accessibility is measured. The algorithm is described in Sheng Yue PhD thesis. (http://dx.doi.org/10.1016/j.jmatprotec.2014.05.006)

API

```python
AccessibleVolume(input_data, mask_data, origin, voxel_size, sphere_diameter_range_min__in_log, sphere_diameter_range_max_in_log, number_of_spheres_in_range, input_image__resolution)
```

- `input_data`: numpy array 3d volume and has to be an 8bit data.
- `mask_data`: numpy array 3d mask volume and it has to be an 8bit data.
- `voxel_size`: numpy array with sizes along 3 dimensions (x_size,y_size,z_size)
- `origin`: numpy array with origin of the volume (x_center, y_center, z_center)
- `sphere_diameter_range_min_in_log`: logarithmic value of minimum sphere diameter range that need to be used in calculating accessible volume.
sphere_diameter_range_max_in_log: logarithmic value of maximum sphere diameter range that need to be used in calculating accessible volume.

number_of_spheres_in_range: the number of spheres diameters that are used in the input sphere diameter range for which accessible volume is calculated.

input_image_resolution: image resolution usually (1.0).

returns: AccessibleVolume numpy array with sphere diameter and corresponding Accessible Volume

Example

To run the example code you need to download the following files Input Data and Mask Data

To run the below code you need to install the following packages:

- tifffile (conda install -c conda-forge tifffile)

```python
from ccpi.quantification.AccessibleVolume import AccessibleVolume
import numpy as np
import math
from tifffile import TiffFile

#input 3d data volume, has to be binary volume.
data = TiffFile('Data128.tif')

#input 3d mask
mask = TiffFile('DataMask128.tif')

#voxel size
voxel_size = np.ones(3,dtype=np.float32)

#voxel origin
origin = np.zeros(3,dtype=np.float32)

#Invoke the Accessible volume algorithm
av = AccessibleVolume(data.asarray(), mask.asarray(), origin, voxel_size,math.log(80.0), math.log(600.0), 11, 9.0 )

print(av)

data.close()
mask.close()
```

Input Volume
Mask Volume
4.2.2 Label Quantification

This algorithm takes in a labelled image and calculates several characteristics for each label. Below are some of the characteristics:

- Volume by voxel counts
- Equivalent sphere diameter by voxel counts
- Bounding box diagonal
- Principal Component Analysis (PCA)
• Ellipsoid fitting by PCA
• Equivalent circle diameter by PCA
• Isosurface by marching cube.
• Surface area
• Surface Volume
• Equivalent sphere diameter from surface volume
• Sphericity
• Normalised surface area to volume ratio (Radius*Sa/Vol)

The details of this algorithm are given in Evaluation of 3D bioactive glass scaffolds dissolution in a perfusion flow system with X-ray microtomography. Yue S, Lee PD, Poolagasundarampillai G, Jones JR. DOI: 10.1016/j.actbio.2011.02.009

API

```
LabelQuantification(input_3d_volume, origin, voxel_size, min_data_value, max_data_value, minimum_feature_size)
```

- `input_3d_volume`: numpy array of 3d volume. data type reflects the function suffix.
- `origin`: numpy array with 3 values.
- `voxel_size`: numpy array representing a voxel size in three dimensions (x,y,z)
- `min_data_value`: minimum value of the input volume
- `max_data_value`: maximum value of the input volume
- `minimum_feature_size`: minimum size of the feature

Returns: an list with feature names and another numpy array with each row representing a label and its corresponding values.

Example

To run the example code you need to download the following data file: Foam Data

To run the below code you need to install the following packages:

- `tiff` (conda install -c conda-forge tiff)

```
# Imports
from ccpi.quantification.LabelQuantification import LabelQuantification
import numpy as np
import math
from tifffile import TiffFile

# Read the input volume which is labelled during segmentation
img = TiffFile('FoamData.tif')
data = img.asarray()

# Voxel size
voxel_size = np.ones(3,dtype=np.float32)
# Origin
origin = np.zeros(3,dtype=np.float32)
```

(continues on next page)
# computes the 3d quantification

```python
lqsNames, lqsValues = LabelQuantification(data, origin, voxel_size, float(np.amin(data)), float(np.amax(data)), 100.0)
lqs.compute()

print(lqsNames)
img.close()
```
4.2. Algorithms
Segmentation Module

Segmentation is the process from which, given a 3 dimensional dataset (volume), surfaces in the volume are found at a specific value. Such surfaces are also called isosurfaces.

5.1 Simpleflex Segmentation Algorithm

The CIL provides the Segmentation Algorithm described by Carr et al. [Carr2003] based on the calculation of the contour tree. The algorithm is developed in C++ and it is fully wrapped in Python.

The algorithm expects to receive a 3D numpy array as input, and outputs a numpy array with the location of the points of the isosurfaces in image coordinates. It must be pointed out that there are at least 2 reference system of the points of an image:

- Image Coordinate: the actual pixel number in each dimension
- World Coordinate: the location in the real world

World coordinates are linear transformation of the image coordinate. For example, the size of the pixel (voxel) may not be uniform in the 3 dimensions resulting in a stretched image in some direction.

The algorithm itself has no clue about the world-coordinates and its output is in image-coordinates. The user is required to apply the appropriate transformations to translate image coordinates to world coordinates. In the example that follows we will make a image-to-world transformation.

5.1.1 Installation

A binary installation is available from the ccpi conda channel. It can be installed both for python 2.7 and 3.5. Install using:

```
conda install -c ccpi -c conda-forge ccpi-segmentation numpy=1.12
```
5.1.2 Usage

The Python wrapper for the CIL uses numpy arrays as medium to pass data to and from each algorithm. 3D image data should be passed to the simpleflex algorithm in form of 3D numpy arrays.

The algorithm outputs numpy arrays. There are 2 types of output of the algorithm:

- a list of isosurfaces: in practice, for each isosurface the coordinates of points belonging to it
- for each isosurface, a mask with the voxels at the isosurface labeled.

To explain how to use it let us go through an example. In the example we will use the viewer that can be installed as follows.

```
conda install -c ccpi -c conda-forge ccpi-viewer numpy=1.12
```

The data we will be using is from VTKData and has been put into a single MetaImage file. This link points to the original VTKData.

First of all, we start with the proper imports:

```
from ccpi.segmentation.SimpleflexSegmentor import SimpleflexSegmentor
import numpy
import vtk
from vtk.util import numpy_support
from ccpi.viewer.CILViewer import CILViewer
```

### The Create a Segmentor and pass data

The algorithm accepts input as 3D numpy arrays. It will detect the dimensions and it will scale the image to an appropriate size (unsigned short or unsigned char).

The algorithm is wrapped in a Object oriented fashion, and therefore it needs to be instatiated and passed the data.

```
# 1. create a segmentor object
segmentor = SimpleflexSegmentor()

# 2. Load some data and pass data into the algorithm
# load data with vtk
filename = "C:\Path\to\VTKData\Data\headsq\headsq.mha"

# read the data as 3D numpy array
data3d , reader = readAs3DNumpyArray(filename)

# accepts input as 3D numpy array
segmentor.setInputData(data3d)
```

The function readAs3DNumpyArray uses vtk to read a MetaImage file and convert it to a numpy array.

```
def readAs3DNumpyArray(filename):
    reader = vtk.vtkMetaImageReader()
    reader.SetFileName(filename)
    reader.Update()
    # transform the VTK data to 3D numpy array
    narray = Converter.vtk2numpy(reader.GetOutput())
    return (narray , reader)
```
Running the segmentation

The only thing to specify to the algorithm is the target isovalue:

```python
# 3. Calculate the Contour Tree
segmentor.calculateContourTree()

# 4. Set the iso-value in percent of the image dynamic range
# one can also pass the actual value
segmentor.setIsoValue(some_value)
segmentor.setIsoValuePercent(35)

# 5. Construct the iso-surfaces
segmentor.constructIsoSurfaces()
```

Retrieve the data

To retrieve the calculated isosurfaces one has to invoke the getSurfaces method which returns the list of isosurfaces a numpy array. To explain how the data are organized in the array, let’s first consider that a surface is made of triangles in space, which in turn are identified by 3 points in space, which are identified by 3 spatial coordinates:

- point = array([coordX, coordY, coordZ, 1])
- surface = array([point_0, point_1, point_2,...])
- listOfIsosurfaces = array([surface_0, surface_1, surface_2,...])

This list is sorted from largest to smallest surface. For instance, the second largest surface will be the second element of the array.

To retrieve the mask from the same isosurface one can use the following code. Notice that during the retrieval we can access the volume of the voxels at the isosurface, the volume of the voxels with value above/below the isovalue. If the spacing is (1,1,1), the volume is the number of voxels.

```python
mask = segmentor.getSurfaceAsMask(0, labelIso=1, labelHigh=2, labelLow=0)
print("Volume of iso \%d" % segmentor.isoVolume)
print("Volume of lower \%d" % segmentor.lowVolume)
print("Volume of higher \%d" % segmentor.highVolume)
    # reshape the mask as the original data
mask = numpy.reshape(mask, numpy.shape(data3d))
```

That is basically it! You can run the following script that will do the segmentation and show something on screen.

```python
# -*- coding: utf-8 -*-
# This work is part of the Core Imaging Library developed by
# Visual Analytics and Imaging System Group of the Science Technology
# Facilities Council, STFC
#
# Copyright 2017 Edoardo Pasca
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
# http://www.apache.org/licenses/LICENSE-2.0
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
```
from ccpi.segmentation.SimpleflexSegmentor import SimpleflexSegmentor
import numpy
import vtk
from ccpi.viewer.CILViewer import CILViewer
from ccpi.viewer.CILViewer2D import CILViewer2D, Converter

def readAs3DNumpyArray(filename):
    reader = vtk.vtkMetaImageReader()
    reader.SetFileName(filename)
    reader.Update()
    # transform the VTK data to 3D numpy array
    nparray = Converter.vtk2numpy(reader.GetOutput())
    return (nparray , reader)

# 1. create a segmentor object
segmentor = SimpleflexSegmentor()

# 2. Pass data into the segmentor
# load data with vtk
filename = "<Path to VTKData>\VTKData\Data\headsq.mha"
# read the data as 3D numpy array
data3d , reader = readAs3DNumpyArray(filename)
# accepts input as 3D numpy array
segmentor.setInputData(data3d)

# 3. Calculate the Contour Tree
segmentor.calculateContourTree()

# 4. Set the iso-value in percent of the image dynamic range
# one can also pass the actual value
#segmentor.setIsoValue(some_value)
segmentor.setIsoValuePercent(35)

# 5. Construct the iso-surfaces
segmentor.constructIsoSurfaces()

# 6. Retrieve the isosurfaces and display
surf_list = segmentor.getSurfaces()

    # retrieve the image mask
    mask = segmentor.getSurfaceAsMask(0, labelIso=1, labelHigh=2, labelLow=0)
print("Volume of iso %d" % segmentor.isoVolume)
print("Volume of lower %d" % segmentor.lowVolume)
print("Volume of higher %d" % segmentor.highVolume)
mask = numpy.reshape(mask, numpy.shape(data3d))

# 7. Display isosurfaces in 3D
# with the retrieved data we construct polydata actors to be displayed
# with VTK. Notice that this part is VTK specific. However, it shows how to
# process the data returned by the algorithm.

# Create the VTK output
# Points coordinates structure
triangle_vertices = vtk.vtkPoints()
#associate the points to triangles
triangle = vtk.vtkTriangle()
# put all triangles in an array
triangles = vtk.vtkCellArray()
isTriangle = 0
nTriangle = 0

surface = 0
# associate each coordinate with a point: 3 coordinates are needed for a point
# in 3D. Additionally we perform a shift from image coordinates (pixel) which
# is the default of the Contour Tree Algorithm to the World Coordinates.

origin = reader.GetOutput().GetOrigin()
spacing = reader.GetOutput().GetSpacing()

# augmented matrix for affine transformations
mScaling = numpy.asarray([spacing[0], 0, 0, 0,
                          0, spacing[1], 0, 0,
                          0, 0, spacing[2], 0,
                          0, 0, 0, 1]).reshape((4,4))

mShift = numpy.asarray([1, 0, 0, origin[0],
                        0, 1, 0, origin[1],
                        0, 0, 1, origin[2],
                        0, 0, 0, 1]).reshape((4,4))

mTransform = numpy.dot(mScaling, mShift)
point_count = 0

for surf in surf_list:
    print("Image-to-world coordinate trasformation ... %d" % surface)
    for point in surf:
        world_coord = numpy.dot(mTransform, point)
        xCoord = world_coord[0]
        yCoord = world_coord[1]
        zCoord = world_coord[2]
        triangle_vertices.InsertNextPoint(xCoord, yCoord, zCoord);

        # The id of the vertex of the triangle (0,1,2) is linked to
        # the id of the points in the list, so in facts we just link id-to-id
        triangle.GetPointIds().SetId(isTriangle, point_count)
        isTriangle += 1
        point_count += 1

    if (isTriangle == 3) :
        isTriangle = 0;
        # insert the current triangle in the triangles array
        triangles.InsertNextCell(triangle);

    surface += 1

# polydata object

(continues on next page)

5.1. Simpleflex Segmentation Algorithm 33
trianglePolyData = vtk.vtkPolyData()
trianglePolyData.SetPoints( triangle_vertices )
trianglePolyData.SetPolys( triangles )

viewer = CILViewer()
viewer.setInput3DData(reader.GetOutput())
viewer.displaySliceActor(42)
viewer.displayPolyData(trianglePolyData)
viewer.startRenderLoop()

# 2D visualization
viewer2D = CILViewer2D()
viewer2D.setInputAsNumpy(mask,
    origin = reader.GetOutput().GetOrigin(),
    spacing = reader.GetOutput().GetSpacing(),
    rescale = False)
viewer2D.startRenderLoop()
5.1. Simpleflex Segmentation Algorithm
A simple Viewer for 3D data built with VTK and Python.

The interactive viewer CILViewer2D provides:

- **Orthoslicer** (slice in x/y/z direction)
- **keyboard interaction**
  - ‘x’ slices on the YZ plane
  - ‘y’ slices on the XZ plane
  - ‘z’ slices on the XY
  - ‘a’ auto window/level to accommodate all values
  - ‘s’ save render to PNG (current_render.png)
- slice up/down: mouse scroll (10 x pressing SHIFT)
- **Window/Level**: ALT + Right Mouse Button + drag. Horizontal motion modifies the window, vertical motion modifies the level.
- Pan: CTRL + Right Mouse Button + drag
- Zoom: SHIFT + Right Mouse Button + drag (up: zoom in, down: zoom out)
- Pick: Left Mouse Click
- **ROI (square)**: This is implemented as a 3D object. Any slices in front of the ROI will hide the box. Any slices located behind the ROI will still show the ROI as they are behind a transparent 3D object. Switching between the orientations should make it clear which region is being selected.

The histogram is displayed based on the current 2D slice which has its boundaries within the 2D extent of the ROI.

Mouse/Keyboard Interactions:

- Create ROI: CTRL + Left Mouse Button
- Resize ROI: Left Mouse Button on one of the hotspots + drag
6.1 Usage

```python
import vtk
import os
from ccpi.viewer.CILViewer2D import CILViewer2D

fn = "./data/head.mha"

reader = vtk.vtkMetaImageReader()
reader.SetFileName(fn)
reader.Update()

v = CILViewer2D()
v.style.debug = True
v.setInput3DData(reader.GetOutput())
v.startRenderLoop()
```

6.2 Screenshots

View along X-axis:
Window/Level Adjust:
Region of Interest (ROI):
Line Profiler:
The core imaging library consists of algorithms contributed by the CCPi community. If you would like to contribute an algorithm or would like to discuss a problem relating CCPi Imaging then please contact ccpi@stfc.ac.uk. Please join our mailing list https://www.jiscmail.ac.uk/cgi-bin/webadmin?A0=CCPI-MEMBERS

If you would like to report any bugs or issues then please raise an issue ticket at https://github.com/vais-ral/CIL-Docs/issues
8.1 Build and Test Automation

These CCPi modules are built automatically from Github repository:

- Pre-Processing - https://github.com/vais-ral/CCPi-PreProcessing
- Reconstruction - https://github.com/vais-ral/CCPi-reconstruction
- Astra toolbox wrapper - https://github.com/vais-ral/CCPi-astra
- Quantification - https://github.com/vais-ral/CCPi-Quantification

With following limitation:

- every commit to master branch is built and binary is uploaded into anaconda channel `ccpi`
- every pull request is built, but no binaries are uploaded
- commit to non-master branch is not built
- the result of built and test process is available in on top of the github page
- version is determined from last git tag and number of commits after this tag, e.g. `0.10.3_103` means 103 commits after last tagged version `0.10.3`.
- if number of commits after tagged version is 0, then this built is assumed production, this can be installed by

```
conda install -c ccpi ccpi-[module name]
```

- if number of commits after tagged version is greater than 0, then this built is labeled as `dev`

```
conda install -c ccpi/label/dev ccpi-[module-name]
```
8.1.1 Recommended setting in continuous integration

Automatic build process can be triggered by various events. The recommended events are push and pull-request. A so called “Webhook” can be configured in Github to send a notification about an event to third party application. Jenkins (or any other CI tool) can be configured to listen to such notification and launch configured action, i.e. build process.

CCPi modules with source codes at https://github.com/vais-ral/ are configured with proprietar installation of Jenkins at https://anvil.softeng-support.ac.uk/jenkins. This service is provided for UK academics community. However, the following is recommended to be set in any Jenkins deployment for every CCPi module. In the following section, replace [module_name] with your selected ccpi module (e.g. Regularisation-Toolkit) and replace [jenkins_url] with url to jenkins instance (e.g. https://anvil.softeng-support.ac.uk/jenkins):

Production built for CCPi-[module_name]

Jenkins is presumed to contain only “Git” and “Github” plugins. No other plugins are required.

To configure build action on push event on master branch received from Github do following in Jenkins:

1. Create new project by: choose folder, New Item, Enter Item Name: CCPi-[module-name], Freestyle project
2. Enter these values:
   a. [x] Github project -> Project URL: https://github.com/vais-ral/CCPi-[module_name]/
   b. [x] Restrict where this project can be run -> Label Expression sl7 (choose this to scientific linux, ubuntu or any linux based machine)
   c. Source code management -> [x] Git ->
      • Repositories, Repository URL: https://github.com/vais-ral/CCPi-[module_name].git
      • Branches to build, branch specifier: */master
      • Additional Behaviours, Check out to specific local branch
d. Build triggers, [x] Github hook trigger for GITScm polling

![Source Code Management](image)

```
module load conda
# commented version = version will be determined from git tag and number
→ commits
# export CIL_VERSION=0.10.3
# uncomment following and specify
# export CCPI_CONDA_TOKEN=[conda token to upload to ccpi channel]
# export CCPI_PRE_BUILD=[if defined "conda build $CCPI_PRE_BUILD" is called
→ before]
# export CCPI_BUILD_ARGS=[optional args to be appended to main build process]
# build and upload
bash -xe $(curl -L https://raw.githubusercontent.com/vais-ral/CCPi-
→ VirtualMachine/master/scripts/jenkins-build.sh)
```
f. (Optionally) specify post build action like email notification etc.

**Note:** Note that repository url ends with `.git` suffix. Otherwise notification from github are ignored.

**Note:** Check out to specific local branch settings ensures that branch is identified e.g. as refs/heads/master. This is used to determine whether and how to upload binaries. master branch are uploaded, non-master branch (pull requests) are built only.

**Note:** `bash <(curl . . . )` calls universal script, see Section below.

**In Github project -> Settings -> Webhooks**

- Add new Webhook
- Payload URL: `[jenkins_url]/github-webhook/`
- Which events would you like to trigger: [x] Just push event.
Development built for pull request on CCPi-[module_name]

To configure build action on pull-request event on any branch received from Github do following in Jenkins:

1. Create new project by: choose folder, New Item, Enter Item Name: CCPi-[module-name], Freestyle project

2. Enter these values:
   a. [x] Github project -> Project URL: https://github.com/vais-ral/CCPi-[module_name]/
   b. [x] Restrict where this project can be run -> Label Expression sl7 (choose this to scientific linux, ubuntu or any linux based machine)
   c. Source code management -> [x] Git ->
      - Repositories, Repository URL: https://github.com/vais-ral/CCPi-[module_name].git
      - Advanced:
Name: origin
- Refspec: +refs/pull/*/:refs/remotes/origin/pr/*
  - Branches to build, branch specifier: **

**Additional Behaviours:**
- Advanced clone behaviours:
- Fetch tags [x]
- Honor refspec on initial clone [x]
- Check out to specific local branch

**Build triggers, [x] Poll SCM**

**Execute shell:**

```bash
module load conda
# commented version = version will be determined from git tag and number commits
# export CIL_VERSION=0.10.3
# uncomment following and specify
# export CCPI_CONDA_TOKEN=[conda token to upload to ccpi channel]
# export CCPI_PRE_BUILD=[if defined "conda build $CCPI_PRE_BUILD" is called before]
# export CCPI_BUILD_ARGS=[optional args to be appended to main build process]
# build and upload
```

In Github project -> Settings -> Webhooks

- Add new Webhook
- Payload URL: [jenkins_url]/git/notifyCommit?url=http://github.com/vais-ral/CCPi-[module_name].git
- Which events would you like to trigger:
  - [x] Let me select individual events
  - [x] Pull request

**Universal built script**


Variants are supported (combination of python version and dependent libraries). It expects that conda recipe is defined in path Wrapper/Python relative to CCPi-[module].

Typical usage:

```bash
# commented version = version will be determined from git tag and number commits
# export CIL_VERSION=0.10.3
# uncomment following and specify
# export CCPI_CONDA_TOKEN=[conda token to upload to ccpi channel]
# export CCPI_PRE_BUILD=[if defined "conda build $CCPI_PRE_BUILD" is called before]
```

(continues on next page)
# export CCPI_BUILD_ARGS=[optional args to be appended to main build process]
# build and upload

These environment variables can be specified:

- **CCPI_PRE_BUILD** - if defined, then “conda build $PREBUILD” is performed before conda build, binaries will be uploaded to anaconda channel together with main build
- **CCPI_POST_BUILD** - if defined, then “conda build $CCPI_POST_BUILD” is performed after conda build, binaries will be uploaded to anaconda channel together with main build
- **CCPI_BUILD_ARGS** - passed to conda build as conda build Wrappers/Python/conda-recipe “$CCPI_BUILD_ARGS”, e.g. CCPI_BUILD_ARGS="-c ccpi -c conda-forge”;
- **CIL_VERSION** - version of this build, it will be used to label it within multiple places during build. If CIL_VERSION is not explicitly defined, then version is determined from git describe –tags
  - Note that version in CIL_VERSION or determined from git tag contains information about last tag and number of commits after it. Thus e.g. last tag is 0.10.4 and current commit is 3 after this tag, then version is 0.10.4_3
  - If the version is release (no number after ‘.’), anaconda upload is production
  - If the version is not release (number of commits after ‘.’) then anaconda upload is labeled as ‘dev’
  - Some commit can be explicitly tagged including ‘.’ char and something after, then it is considered as ‘dev’ version
- **CCPI_CONDA_TOKEN** - token to upload binary builds to anaconda - it detects the branch under which the CCPi is build, master is uploaded to anaconda channel, non-master branch isn’t

Optional Github and Anaconda information badges

In order to have feedback status of the build process on github or any other third party page, you may put the following code, which shows the image of the build status and links to the build status page where more information can be obtained in order to investigate a build failure.

Note: Replace [CCPi-module] by a selected module, e.g. CCPi-Reconstruction. Replace [conda-module] by module name in conda - it may differ, thus e.g ccpi-reconstruction.

CORE (C/C++) built as library  C/C++ API

Wrappers:  Python Avizo Paraview Tomviz

## 8.2 Essential Tools

The development cycle of the CIL requires the usage of a few common tools:

1. git and GitHub  
2. CMake  
3. Anaconda and conda build
These tools are meant to simplify the development cycle, but the developer is required to have some confidence in the use of them. Mostly, one would be modifying existing files, but you never know.

### 8.2.1 Directory structure

Each CIL repository (public or private) is organized in subdirectories as follows:

```
project_name/
  CMake/
  CMakeLists.txt
  License.txt
  Core/
  recipes/
  Wrappers/
    Python/
    Avizo/
    Paraview/
    Tomviz/
```

### 8.2.2 Git and GitHub

Code development is done via git on (public or private) repositories. The master branch should contain only the latest development stable code. Each new feature should be implemented on a separate branch. Merging of new features on the master branch is done via Pull Requests (PR).

We strongly encourage the use of issues, milestones and projects on GitHub to keep track of the development cycle.

**Typical Workflow:**

1. Use your branch for big and small changes
2. Pull from master frequently to reduce the amount of work needed when merging
3. Use pull requests (PR) to add features in the master, even if you merge them yourself. The best way is to branch from master, make the patch and issue a PR
4. Use issues for bugs, proposals and discussions in general. You can assign issues to someone to require their attention.

**Some git commands**

```
git status
git diff filename
git add filename
git commit
git commit -m "message"
git pull origin <branch>
git push origin <branch>
git rebase -i HEAD~4  # rebases interactively the last 3 commits: read the screen
git checkout <branch>  # to change the actual local branch
git checkout -b <branch>  # to create and checkout a new local branch
git branch -d <branch>  # to delete a local branch
```
On conflict read what git says. Modify the files that git says need attention. You will find in those files some lines like these:

```
<<<<<<< HEAD
  <link type="text/css" rel="stylesheet" media="all" href="style.css" />
=======
  <!-- no style -->
>>>>>>> master
```

your branch is between HEAD and =====, master is between ===== and master. Once removed that git commit

Developing on Windows you may get problems with file permissions. Change from 0755 to 0644 with

### 8.2.3 CMake

Building of the CIL modules is done with CMake, and each module will contain the appropriate CMakeList.txt files. Although one could build the Core with CMake alone, we often use conda to build the Core library.

### 8.2.4 Anaconda

Install Anaconda or Miniconda. `conda` is a package manager system and virtual environment manager, possibly more.

**Managing environments**

With conda you can install multiple python versions and use them at the same time without messing around with your system python installation.

The first thing to do is to create an environment. This is achieved by:

```
conda create --name <environment_name> python=3.5
```

Basically this instructs conda to create an environment named `<environment_name>` with python 3.5. You can also specify a list of other packages you want to install in your environment at creation time.

You can delete an environment by:

Refer to the main conda docs for further information.

**Installing packages**

The basic install command is

```
conda install <package-name>=<version>
```

`conda` searches and installs packages from its main source anaconda.com. It is possible to add sources (channels) of packages from anaconda.org which is a community driven repository. This can be forced on conda by

```
conda install <package-name>=<version> -c <channel-name>
```

One may add a number of channels and they should be searched in the order in which you provide them. I find it easier to use the `.condarc` file to specify the channels:

```
channels:
  - ccpi
  - conda-forge
  - defaults
anaconda_upload: false
```
This instructs conda to search with higher priority the ccpi channel, followed by conda-forge and last default. Notice that any package which may reside on a different channel cannot be installed unless the channel is passed to conda.

8.3 Building with Conda

While building with conda, conda creates an environment for the purpose, copies all the relevant data, issues cmake and packages everything. It’s pretty neat but it must be configured. This configuration is called conda recipe.

We will cover the building with conda in 2 steps:

1) building with a conda recipe that exist and works

2) creating a conda recipe

During the development cycle you will be faced with building your software again and again. The suggestion here is to continue to use a conda build as it keeps things organized. Therefore you will be faced more often with case 1), i.e. building with a pre-existing and working conda recipe. When a new package is created, a new conda recipe must be written. This will happen with less frequency, and I will cover it later.

8.3.1 Building with existing conda recipe

In the CIL there are basically 3 kinds of packages:

1. Shared libraries
2. Python wrappers (or other)
3. Pure Python packages

To compile a shared library:

1. start in the main repository directory
2. export CIL_VERSION=someversion
3. conda build recipes/library --numpy 1.12 --python 3.5 (adjust the python version)
4. conda install cil_libraryname=someversion --use-local --force

To compile a Python wrapper to a shared library or a pure Python package:

1. in the Wrappers/Python directory
2. conda build conda-recipe --numpy 1.12 --python 3.5
3. conda install ccpi-pythonpackagename=someversion --use-local --force

When launching the build you may have activated an environment or not. I suggest to activate an environment with most of the needed packages as the conda build will be quicker. It is fundamental to have an environment activated when installing. Notice that there isn’t any dependency check when installing local packages. Notice that you will have to force installation whenever the version of the package doesn’t change.

When builds end prematurely (on errors), conda will not remove the build tree. Every now and again issue aconda build purge to clean your hard drive.
8.3.2 Writing a conda recipe

The conda build requires the presence of the so-called conda recipe. A recipe lives in a directory where there are 2 or 3 files.

```
recipe/
    meta.yaml
    bld.bat
    build.sh
```

The `meta.yaml` file contains informations about the package that will be created, its dependency at run time and at build time. The `bld.bat` and `build.sh` are files invoked by conda during the build process and are dependent on the system: windows or unix.

In the following a `meta.yaml` for one of the ccpi packages. It should be self evident that one describes the package, its dependencies at runtime and build-time.

```
package:
    name: cil_regularizer
    version: {{ environ['CIL_VERSION'] }}

build:
    preserve_egg_dir: False
    script_env:
        - CIL_VERSION

requirements:
    build:
        - boost ==1.64.0
        - boost-cpp ==1.64.0
        - python 3.5 # [py35]
        - python 2.7 # [py27]
        - cmake >=3.1
        - vc 14 # [win and py35]
        - vc 9  # [win and py27]
        - numpy
    run:
        - boost ==1.64.0
        - vc 14 # [win and py35]
        - vc 9  # [win and py27]
        - python 3.5 # [py35]
        - python 2.7 # [py27]
        - numpy

about:
    home: http://www.ccpi.ac.uk
    license: Apache v2.0
    summary: Regularizer package from CCPi
```

In the `build.sh` one specifies how to build the package.

```
#!/usr/bin/env bash
mkdir build
cd build
#configure
```
BUILD_CONFIG=Release
echo `pwd`
cmake .. -G "Ninja" \ 
-DCMAKE_BUILD_TYPE="$BUILD_CONFIG" \ 
-DCMAKE_PREFIX_PATH:PATH="$PREFIX" \ 
-DCMAKE_INSTALL_PREFIX:PATH="$PREFIX" \ 
-DCMAKE_INSTALL_RPATH:PATH="$PREFIX/lib"

# compile & install
ninja install

There are a number of environment variables that are set by conda, like $PREFIX.

Building Core with conda

  1) Clone the git repository git clone https://github.com/vais-ral/CCPi-FISTA_Reconstruction.git
  2) Create a directory for the builds outside of the source directory
  3) conda create --name cil python=3.5 numpy=1.12
  4) module load python/anaconda (optional, depends on the actual machine installation)
  5) source activate cil

8.3.3 Known issues

On linux with encrypted partitions conda build may fail with a

Conda hints that you don’t have permissions, but trying to create the directory is impossible as you are exceeding the maximum path length. The only solution is to build to a different drive using --croot /path/to/unencrypted/drive
0.9.1:

*Added Viewer package*  Other packages mainly code refactoring, with no new functionality added

0.9
Bibliography

