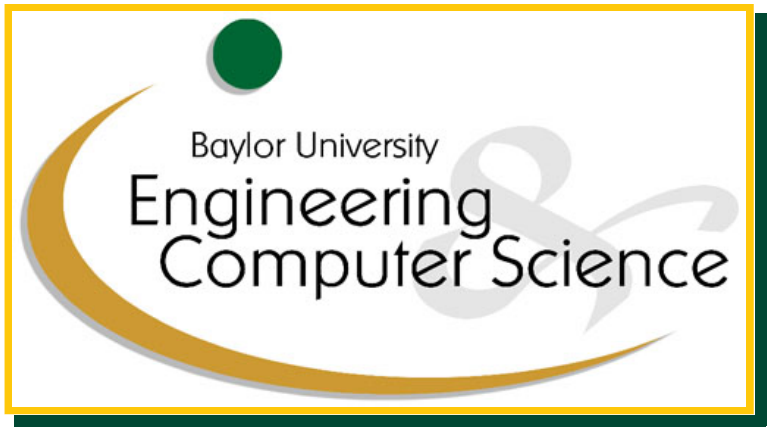
User's Guide: pCT Image Reconstruction Program

Details on Settings, Optional Parameters, and Development Status

Blake Edward Schultze

June 30, 2014



BAYLOR UNIVERSITY

(1) Outline of preprocessing steps (see 'ProgramFlow.pdf' for visual representation)

1. Count histories in each file and total
2. Iterate through all data, one file at a time, and:
   (a) Read data from file
   (b) Identify and remove protons missing reconstruction volume
   (c) Bin protons traversing reconstruction volume according to $[\theta, t, v]$
   (d) Perform SC/MSC/SM with traversing protons with low/low/high WEPL, respectively (*optional*)
3. Perform statistics on MSC/SM data and determine hull (*optional*)
4. Calculate mean $\mu = \sum x_i$ WEPL and relative ut/uv angle angle for each bin
5. Iterate through all remaining data, one chunk at a time, and:
   (a) Calculate total squared discrepancy for each bin, i.e. $\sum_i (\mu - x_i)^2$ for WEPL and relative ut/uv angle measurements x_i
6. Calculate standard deviation $\sigma = \sqrt{\frac{1}{N-1}\sum_i (\mu - x_i)^2}$ for WEPL and relative ut/uv angle
7. Iterate through all remaining data, one chunk at a time, and:
   (a) Identify and remove histories if $\mu - x_i > N\sigma$ ($N = 3$ now) for WEPL or relative ut/uv angle measurement x_i
8. Recalculate mean WEPL μ for each bin to produce sinogram
9. Perform FBP (*optional*)

(2) General notes on preprocessing code and program

Variables with uppercase names correspond to constants (both defined and precalculated), parameters, or options, all of which are defined in the header file. The intention is to make it clear which variables refer to immutable runtime data which can only be modified in the header file before program execution; none of these variables are defined, set, or modified outside the header file. Defining them here makes these global variables which can be accessed anywhere on the host/GPU without having to pass them to function/kernel calls.

Since these constants, parameters, and options are not intended to be modified, they are either defined as preprocessor macros ("#define" [name] [value]) or constant value variables ("const" [name] = value). Most of the options that control conditional program execution correspond to a true of false condition and can easily and clearly be implemented using boolean variables ("const bool [name] = [true/false]") and if/else

statements. However, variables corresponding to options that can take on 3 or more values are defined as an instance of an enumerated type, where the enumerated type and its member correspond to the option and its possible values, respectively ("enum [type_name] *option1*, *option2*,..."). A "const" variable of this enumerated type is then defined and assigned one of these optional values ("const [type_name] [option_name] = *option*") to specify the choice of option. There are several advantages to using enumerated types for controlling conditional program flow.

However, for options with 3 or more values, it is convenient, easier to read, and generally more efficient to use a switch statement instead of a series of if/else if/else statements. However, unlike if/else if/else statements, switch statements can only accept constant integral ("int", "char","bool") or enumerated types, which does not include strings. Booleans can only take on two values and using an integer or character to specify an option makes the function of the switch statement and its various cases opaque, thus requiring additional documentation to describe to which option each integer/variable case corresponds. On the other hand, members of an enumerated type can be given names that correspond directly with the option and the resulting switch statement is easy to read and understand. In addition, an integer/character can have numerous values, almost certainly exceeding the number of optional values required. Thus, the switch statement must include a "default" case which executes if the value of the option does not correspond with any given case. On the other hand, an instance (i.e. variable) of an enumerated type can only be assigned a value corresponding to one of its members, otherwise a compilation error will occur. Hence, as long as the switch statement has a case defined for each member of the enumerated type, no "default" case need be included since the variable cannot take on any other value.

Variables are named with the intention of making it clear what data they refer to, which often makes them long. Similarly, functions are named such that their purpose/action is clear and the names of their input parameters are intended to make it clear what data will be passed to them in function calls, often closely resembling the actual names of these variables in memory (as defined in the header file). In addition, input parameters corresponding to data that is only read and not modified by the function are typically given the "const" qualifier in the function definition parameter list to emphasize this fact.

Although explicitly defining a parameter as constant within the scope of a function has no explicit effect, it does make it clear what data is or is not modified by a function and prevents accidental data modifications. Recall that a variable refers to a portion of memory where data is stored and its name is simply an identifier used to refer to its memory address. When an input parameter is declared "const", it prevents this identifier from being used to modify the data it refers to. If a function attempts to use the identifier to modify the data, compilation will fail and the user will be notified (typically something like "expression must be a modifiable lvalue"). However, since the "const" qualifier only restricts the usage of the identifier and does not protect the actual, if there is another identifier addressing the same data (i.e. an alias) that is not marked "const", this identifier can still be used to modify the data. Thus, aliasing is avoided throughout the program even if modifications to a variable's data over the course of preprocessing means it could be more aptly named. Instead, variable names are assigned which appropriately reflect the data throughout their lifetime.

(3) Specifying location of input data and where output is to be written.

First, specify the directory where the input data is stored ("input_directory") and the directory where output data should be written ("output_directory"). Note that since the various data sets are typically stored in separate folders in the same directory, the program is designed as such, so do not include the folder where the data is located in the directory name, the folder name should be written to the variable "input_folder". Similarly, output for various data sets are typically in separate folders in a single directory, so do not include the output folder name in the output directory name, write the folder name into the variable "output_folder". The filenames for the data files should then be written to the variable "input_base_name" and the file extension to "file_extension" (e.g. .bin, .dat, .txt, etc). Note that some compiler recognize the backslash character "\" as an escape character, e.g. "\n" is new line, "\t" is a tab, etc. To tell the compiler not to interpret this literally, you must use "\\".

The filename format is expected to be "[input_base_name]_xxx.[file_extension]", where "xxx" is a 3 digit number specifying the gantry angle from which the corresponding data was acquired. This completes the input/output path specifications. Since data has been written in various formats in the past/present, you must specify if the data is written in data format Version 0, Version 1, or the format prior to Version 0 by setting the corresponding boolean to true and the others to false (VERSION_0, VERSION_1, and VERSION_OLD, respectively). Similarly, since there have been inconsistent units used in the past (i.e. mm/cm), two booleans have been added to provide compatibility by setting the "SSD_IN_MM" and "DATA_IN_MM" booleans appropriately. In the past, many of the data sets also came with a config file (scan.cfg) which specified the u coordinates of the tracker planes, so if you are using any such data set, set the CONFIG_FILE boolean to true. Otherwise, you must set it false.

If the program cannot find a file, whether it be data or a config file, an error message is printed to the console window and program execution is halted. To allow one to see the error message, an user input request is initiated. Once you are finished reading the error message, hit any key to exit the program.

(4) Hull-Detection (i.e. determining object and its boundary for use in MLP)

There are currently 4 methods of hull-detection: FBP, Space/Silhouette Carving (SC), Modified Space/Silhouette Carving (MSC), and Space/Silhouette Modeling (SM). These can be turned on/off using the variables FBP_ON, SC_ON, MSC_ON, and SM_ON, respectively. These can typically all be turned on at the same time so their results can be compared, but this is not always possible; if the data sets are large, since each of these algorithms consume GPU memory (particularly SM), it is possible to exhaust the available GPU memory. In most cases this does not raise an execution halting exception, but subsequent processing generates erroneous results. This can be seen from the text output to the console window which notifies the user of how many protons passed through the reconstruction volume and, thus, were not cut from the data set: if memory is exhausted, the program will indicate that 0 histories pass reconstruction volume intersection detection for the remainder of gantry angles not yet processed.

(5) Writing Data to File

Initially functions for writing data/images to disk were only added as needed and these were written for specific purposes (viewing/debugging/etc), types of data (bool/int/float), containers (arrays/vectors), and formats (single/separate files); clearly a function written to accept a floating point image stored in an array and write each slice to a separate file could not be used to write an integer vector containing bin counts to a single file. Eventually nearly every combination of data type, container, and number of files to be written had been needed at some point so functions were added to support any possible scenario (12 functions). Aside from the differences for writing to a single file or multiple files and the parameter corresponding to the data/image in each function's definition, their implementation was identical since accessing an element of an array and a vector are both done using "operator[]". Therefore, these 12 functions have been replaced by 2 disk writing functions, one for arrays and one for vectors.

Similar to the specification of the path to input data, these functions have input parameters specifying the name of the file, the name of the folder to write it into, and the directory where this folder is located. This leaves the user free to name their files however they like and to write them to wherever they want, with each of these three parameters passed as character arrays as in "filename". However, the similarity between these parameters and those for the reading of data is due to the assumption that, like the input data, all output data will be written to the same directory and for each input data set, there will be a separate folder in this directory for its corresponding results. If this is the case, define the output directory and folder variables in the header file (like was done for the input path information) and pass these as parameters along with a file name of your choice so the function calls need not be rewritten each time the data set is changed. These functions could have been simplified by incorporating the output directory and folder variables into the function itself, but this would have forced conformity to the aforementioned data storage scheme, so to allow for such differences and make it possible to write a particular data set somewhere else, these were left as parameters so they can be written explicitly if desired (i.e. "directory"/"folder").

Sequential data elements are comma separated and 3 parameters dictate line and file endings: (1) the number of data elements per row (columns/line endings), (2) the number of rows, and (3) either the number of files with this structure or the number of times this row/column structure is repeated below each other in a single file (depending on if writing to one or multiple files). This format was chosen primarily for the purpose of writing images and the binned data to file, either for writing them to one file with each slice placed below one another or for writing each slice to a separate file. When writing an image or binned data to disk, there is an obvious choice for the number of rows, columns, and slices/files, but for other data this is not the case. For this data, it is unlikely that an obvious choice of rows, columns, and slices/files will have a product equal to the total number of data elements in the array/vector, so it is possible to attempt to access an element past the end of an array/vector. Attempting to do so either ideally generates a run time error but it can also result in erroneous data being written to disk, neither of which are acceptable. Fortunately, a vector's size is an accessible property and this situation can easily be prevented. However, a C array generated using the malloc/calloc functions is not an object in the traditional

sense, it is just a block of memory and its variable name is simply a pointer to its first element, and there are no member functions like size() we can use. This absence of properties is precisely the reason this approach to data storage is used for nearly all data in this program; unlike most objects, which consume additional memory to maintain internal property and status variables such as size, explicitly allocated/released data is lightweight and efficient. Therefore, we must add an additional parameter to the function for the user to specify the number of elements in the data set. The user can then use these two functions to write any data to disk and choosing whatever combination of rows, columns, and number of files is convenient for later data analysis from file.

This implementation provides substantial flexibility and appears to be sufficient at the moment, but improvements and simplifications will continue to be implemented when they present themselves or if it becomes apparent that they are needed. Note that although these functions can be added anywhere in the program, if the data desired resides on the GPU, it must first be transferred to the host using:

"cudaMemcpy(device_name, host_name, data_size, cudaMemcpyHostToDevice)"

Please keep me informed if there are any issues with this consolidation of function.

(6) Host/GPU computation and structure information

A proton history contains (12) u/t/v coordinate measurements, a WEPL measurement, and the gantry angle at which these measurements were acquired, so for most data sets and on most computer systems, the storage capacity of the GPU is not large enough to store the entire data set at one time. Thus, we can only process a subset of the data set at one time and then repeat until the entire data set has been processed. The number of proton histories in these subsets is controlled with the constant MAX_GPU_HISTORIES and its upper limit depends on the amount of GPU memory; eventually, the program will query the system to determine the GPU's specifications and determine this value automatically. However, because the program is still in development and it is unclear exactly how much memory is needed for basic operation and how much additional memory is required per proton history, MAX_GPU_HISTORIES must be set manually.

At the moment, the program begins by determining the number of proton histories in each file and then on each iteration, it determines how many files it can process at once before exceeding MAX_GPU_HISTORIES. To provide forward compatibility and prevent the need for structural redesign in the future, the preprocessing routines are capable of processing any number of proton histories at once. This not only allows us to reach maximum efficiency by processing as many proton histories at once as possible, but it also allows us to begin preprocessing as data becomes available during acquisition, before completing the scan. In other words, the software does not limit the number of histories that can be processed simultaneously, the only restriction on this number is the GPU capacity. However, for convenience, if a file cannot be read in entirety without exceeding MAX_GPU_HISTORIES, the file is not included in the current iteration. Since this restriction is not enforced by any of the processing routines, it is isolated in the "while" loop in the main function (and to a lesser degree the function used to read the data) so it can easily be removed in the future if necessary.

During early development, it was found that 4-5 million proton histories could be processed simultaneously (though this may have dropped by now and obviously depends on each file's size), corresponding to around 5-20 files depending on the number of histories in each file. On the other hand, it was also found that increasing the number of histories processed simultaneously had negligible effect on execution time. Although initially counterintuitive, this is due to the fact that computation accounts for a very small fraction of execution; almost the entire execution time is associated with data transfers to/from the GPU. Thus, it is primarily the total size of the particular set of data to be transferred that dictates the transfer time and it makes very little difference whether it is transferred in one piece or broken into smaller pieces. Of course, there is some overhead cost associated with data transfers, but this accounts for an even smaller portion of execution time than computation does. Therefore, since there are some minor complications with partially reading files and it is convenient for development verification to read a single file (i.e. gantry angle) at a time, the current implementation does not allow partial reading of files.

Although multiple files can be read simultaneously if these are written using the data format prior to Version 0, provided that the total number of histories does not exceed MAX_GPU_HISTORIES, this is currently not possible with Version 0 or Version 1. This will eventually be possible, but since the data format is likely to change, this is not a priority. Even with the old data format, reading multiple files simultaneously is not recommended, this feature was only included only for forward compatibility. The hull-detection algorithms consume memory as well (particularly SM), and if multiple hull-detection algorithms are to be used simultaneously so their results can be compared, exceeding 1-2 files at a time can potentially exhaust the available GPU memory (refer to hull-detection section on indications of memory exhaustion). As their is no significant benefit to processing more than 1 file at a time, MAX_GPU_HISTORIES is typically set to a value slightly larger than the maximum number of histories per file. As long as the number of histories per file (gantry angle) is relatively constant, this will enforce single file processing. This approach is particularly useful during development for identifying problems with data and/or computation, especially the more insidious issues with unstable numerics that only occur in certain situations or under particular circumstances, as these can generate computational errors that either go unnoticed or are difficult to locate. For example, trig computations near vertical angles, where $\cos(\theta) \to 0 \implies \tan(\theta) \to \infty$, causes computational errors that become quite large the closer to vertical the angle is. This particular issue was encountered several times during development, both in simulated data sets and with preprocessing computations; the simulated data associated with vertical projection angles was invalid due to numerical errors calculating proton paths and the preprocessing program inaccurately calculated the points of intersection of the proton path with the reconstruction volume for high slope paths, with errors increasing in magnitude as angles approach vertical. These numerical instabilities occurred infrequently and may have gone unnoticed if each gantry angle was not processed individually. In fact, the original pCT reconstruction program contained numerical instabilities (e.g. the same intersection inaccuracy), but because its effects were rarely encountered, the propagation of the error was not dramatic enough to be immediately noticeable (contact me if interested in details).