# RTM User-Guide

This guide is meant as an informal introduction on how to run/configure/use the Photon Tracking Model. Keep in mind, it’s new and I’m sure it has bugs, and its definitely missing some consistency checks. … So don’t try to break it just yet by supplying various arguments that defy logic (e.g., asking it to compute transmission at a depth that is greater than the snowpack depth, or telling it to get data from files that don’t exist).

Also, unfortunately, this is not a fully packaged software that you can install from the command line, and as such, it makes use of inline sys.path.append() functions to link all of the relevant modules to eachother. So, in many of the .py files, you will need to go in an edit all the sys.path.append() functions to point to files on your computer before anything works. Further, I cannot know exactly how this will work on a windows computer.

Hopefully this guide helps get you started, and if you find bugs, or find additional information that would be useful, please let me know, or add it to the guide!

To sum up, the photon tracking model can be broken down into three steps once one has the processed microCT data in hand.

1. Process stacks of 2D microCT images into sample volumes
2. Run Geometric optics photon tracking through volumes to get the following optical properties saved to a sample-specific file
   1. Extinction Coefficient
   2. Fraction of path traveled within Ice
   3. Phase Function
3. Run Slab model with optical properties set by files generated in step 2.
   1. Slab model can be configured with multiple layers and with a lower boundary and with known contaminates (maybe).

General code map:

* Main
  + SlabModel.py -> main slab model object and code
  + RTcode.py -> Some functions to perform Geometric Radiative Transfer Calculations
  + CRRELPolyData.py -> Object code to hold CRRELPolyData functions for 3D meshes
  + DrawShapes.py -> Code containing functions to create various shapes and to call CRRELPolyData class.
  + RenderFunctions.py -> Code that simplifies rendering VTK meshes in python
  + ImageSeg.py -> Python code to process image stacks
  + BRDFFunctions.py -> BRDF functions for lower-boundary surface properties (in development)
  + SetSlabProps.py -> Script to calculate slab properties from MicroCT samples
  + Solarposition -> File containing solar position code from: <https://github.com/s-bear/sun-position>
* Materials -> Folder containing csv files with refractive indices for various materials, generally you just want ice, but the other ones are used for contaminates
* Surfaces -> Folder with csv files that have reflectance of various surfaces as a function of wavelength (e.g., as determined from the ASD).
* Examples
  + PackagedSlab.py -> Example script that runs various iterations of the slab model. Great for learning how to use
  + ReadASD.py -> Script for reading ASD data and comparing to saved model output.

# Getting set up:

Included in the NewRTM directory, is a .yml file. You’ll want to create a dedicated Anaconda environment from this file. Doing so will create an environment that has all of the program dependencies compiled so that you don’t need to install individual dependencies to a preexisting environment. To this (assuming you have the anaconda package manager), open up a terminal and cd into the NewRTM directory and enter:

conda env create -f CRRELRTM.yml

Then, if you’re on a mac or linux box, to activate the environment once it’s installed

source activate CRRELRTM

It might be the same command for a PC, but I do not know. Just because that when you’re working with the model, you’re always working in the CRRELRTM environment.

For Windows users, you may need to install the C++ build tools workload in Build Tools for Visual Studio, found here: <https://docs.microsoft.com/en-us/cpp/build/building-on-the-command-line?view=msvc-160>

In addition, if you are getting pip errors with vtk or pyvista, try removing them as dependencies in the crrelRTM.yml file. These packages are both dependencies of pymeshfix and so will be installed through pip. If they have already been installed with conda, this can cause issues.

Using ImageSeg.py

To use ImageSeg.py to generate meshes requires the most python dependencies. This script allows for lots of user-tunable parameters and configuration to generate a .vtk 3D mesh file for EACH individual grain in the sample, as well as the full sample. Each “grain” is determined using a peak-local-max python function to find grain centroids and a watershed filling algorithm to find grain boundaries. This works … okay … most of the time. The big benefit of this is it reduces the time required to generate meshes by a factor of 100. In the end, the whole mesh is generated by merging all the different grains together. I should note, that this is just a script, and nothing formal that you would call or work with otherwise.

User Tunable Options / Parameters are described in the comments of the ImageSeg.py script.

Note a few things:

* There are somewhat subjective tuning parameters (e.g., grainsize) that will be different for each MicroCT sample, so you will need to play around with each sample before you find a good match.
* I do not recommend trying to run the WHOLE sample through at once, this will probably cause your computer to melt. Also, many of the image pixels in x/y fall outside of the cylindrical sample VOI and are junk data.
* Use the “check” flag often to inspect your sampling and grain separation.
* Alternatively, in order to accurately calculate the extinction coefficient through curve fitting, you need a sample volume large enough to build the curve. I recommend a minimum volume size of 5x5x5 mm, unless you are working with a sample with tightly packed fine-grained snow (typically 250x250x250 voxels)
* I recommend installing Paraview on your computer to visualize the output VTK files of the grains.

Using SetSlabProps.py

This function sets the extinction coefficient, mean ice path, and phase function properties that are required for the slab model. This uses the explicit photon-tracking model from Kaempfer, 2007 to compute photon trajectories as they move throughout the 3D snow sample. It also builds off of techniques described in Xiong et al. (2015) to extract medium properties. In short, you specify a 3D mesh sample in addition to some other parameters, and it runs, and after a while, outputs a .txt file with theses properties. This file is read into the slab model.

Similarly, the user parameters are set at the top of the script.

Note a few things:

* The number of photons should be at least 1000, which will give you precision down to 1/10 of a percent for the probability of extinction curve it. More photons are better, but if you start going above 5000 photons, especially for larger samples, it can REALLY increase compute time (Several hours per sample).
* Be sure that you point to the right VTK file, and not one of the individual grain files

# Using SlabModel.py

SlabModel.py is a code that contains all of the functions contained within the SlabModel object. In practice, you will write your own python code, and run the SlabModel functions as needed. This is different than the other model components which are just scripts you run. A sample script in the Examples directory called “PackagedSlab.py” has a lot of different examples of ways to use SlabModel.py

In this configuration, there is a namelist, that you can configure up front, which populates the model parameters upon object initialization in the code:

Import SlabModel as SM

Slab = SM.SlabModel(namelist=’Mynamelist.txt’)

Where “Mynamelist.txt” is the namelist with your parameters. You can access these within the script without having to initialize a new model object through the namelist dictionary.

For example, to set the diffuse fraction of incident radiation to 50%:

Slab.namelistDict[‘DiffuseFraction’] = 0.5

Once the namelist is set, an initialize function is required to essentially hardcode the namelist parameters into the model object and perform some quality assurance checks to make sure things are okay. Note, that any changes you make to the namelist will not affect the model until you run (or rerun) initialize.

Slab.initialize()

Once the model is run, there are several functions that you can use to simulate radiative transfer. For example, you can compute the BRDF for a specified wavelength and incident angle. You can compute the spectral albedo over a range of wavelengths. You can run a sample tracker to see the position and energy of individual photon packets as they move through the snowpack. These are all depicted in the PackagedSlab.py example script.

For example:

Albedo,Absorption,Transmiss,transmissionDict=Slab.GetSpectralAlbedo(WaveLength,

Zenith,Azi,nPhotons=10000)

Transmission dict will return as a zero, unless a list of transmission depths is supplied to the function into the transmission argument:

Albedo,Absorption,Transmiss,transmissionDict=Slab.GetSpectralAlbedo(WaveLength,

Zenith,Azi,nPhotons=10000, transmission=[490,450,400])

In this example, transmissionDict will return estimated transmission power at the listed depths (in mm) within the snowpack.

This model can ALSO take a really long time to run, because with the low absorptivity of ice at the shorter wavelengths, it can take on the order of ~100,000+ photon steps to deplete the photon energy, so it can take a while. It’s much faster in the NIR. So, in practice, I recommend using the WriteSpectralToFile function for spectral albedo, so that you can do analysis on the model output without having to take 1-2 hours to rerun the model.

Slab.WriteSpectralToFile('outputFile.txt',nPhotons,Zenith,Azi,WaveLength,Albedo,

Absorption,Transmiss,filename='File description here!')

Note that in the above, the albedo, absorption, transmission are all model output that matches the wavelength. Zenith and Azimuth angles and nPhotons are included to additional metadata, and the remaining metadata comes from the namelist dictionary and set parameters contained within the slab object. At some point I’ll write an additional function to write out the BRDF to either a gridded ascii file, or netCDF or something.