CHROMA RAYTRACING SCRIPT FOR VUV SETUP: TECHINCAL MANUAL

Date Last Compiled: September 6, 2019

Contact: Ben Page
Email: benjaminpage.acer@gmail.com
Affiliation: Swansea University

Contents

1 Overview

This document serves as the technical manual for the Chroma scripts used to simulate the optics of the VUV setup. The following contains a breakdown of the content of each script, including some comments about the code itself. If more information is needed, please contact the email address listed. If you're instead looking for information on installing and running the scripts, please see the *User Manual*.

2 Breakdown: Simulation Scripts

2.1 `main.py`

Arguments: `<filetype: .txt>`

Takes file of `.txt` form containing the value expression to be read by the script (*see User Manual*).

This script serves as the frontend for the entire simulation package by passing user input to the relevant scripts. The inputs can either be given in `.txt` form or by manually inputting the values when prompted. Once the user has inputted the values, they are converted from Metric to Imperial to be compatible with the unit system of the `.stl` files. These converted values are then passed onto `simulator.py`.

2.2 `simulator.py`

Arguments: `n wavelength width beamsize regen`

`n`	– Number of photons to simulate (int)
`wavelength`	– Wavelength of photons in nm (int)
`width`	– Slit width in inches (float)
`beamsize`	– Width (height)[1] of beam in inches (float)
`regen`	– Regenerate setup geometry (bool)

This script performs the actual simulation of photons. It first checks the value of the `regen` argument - if `regen = 0/False` and the geometry config file is present (*see* `builder.py`) then the geometry is loaded from file. Otherwise, if `regen = 1/True` or the geometry config file is not present, then `builder.py` is invoked and the geometry is regenerated. Loading the geometry from file reduces the runtime – however, changing parameters such as the slit width will require the geometry to regenerate. The script should detect these changes by comparing the requested width with the setup parameters stored in `vars.txt`.

The light source `init_source()` is then defined. This function defines the starting locations and the directions of each of the photons to simulate. It does this by calling the `get_source()` function (*see* ***`mfunctions.py`***), which generates a position array which is sampled from a Gaussian distribution with `beamsize` as its width in the z-direction and is uniformly random in the x-direction. The directions of each photon are such that the angles mimic[2] the expected diffraction pattern in the x-z plane. The function then returns a `chroma.Photons` object with initialized positions, directions, polarizations and wavelengths.

2.3 `builder.py`

> Arguments: *width*
>
> `width` – Slit width (in inches)
>
> *Note: If* `builder.py` *is executed directly as* `__main__`*, it will generate and render the setup geometry, otherwise the script will just generate the geometry.*

This script generates the setup geometry from the `.stl` files contained in the `stls/` directory. It first generates a `chroma.geometry.Geometry` object called `world` which will contain all the solids of the setup. It then generates a 40in^3 cube which denotes the bounds of the world.
The script then loads the mesh blacklist (*see Section 4.3*). A 1in^3 yellow cube called `setup_solid` is generated – this serves as both the origin of the coordinate system and also the solid from which we can build the setup[3]. Each mesh contained in the directory `stls/` is checked against the blacklist and loaded (if the mesh is blacklisted, it is ignored). If the mesh is either Mesh 141 (closed mirror) or Mesh 142 (open mirror) then the surface and color arrays are generated by `mfunctions.mirror()` (see Supplementary Script: `mirror()`), otherwise the mesh is deemed 'non-optical' and is assigned a grey color and a totally absorbing surface material[4]. The constructed setup solid is then added to `world`.
Once the non-special components of the geometry have been generated, the script then starts generating the special components. Special components are components of the geometry that require some kind of specific alteration (such as material changes, rotation or translations) before being added to the setup. The `sipm_plate` solid is the copper section of the cryo-arm that holds the SiPM board. The plate model required a slight rotation correction, as it was not parallel to the xy-plane. The `detector` solid serves as the detector at the

[2]Diffraction is not native to Chroma, and since the reflector is focussed on the slit then there must be some conical-like source of photons for the reflected beamshape to be planar.

[3]Chroma allows `chroma.geometry.Solid` objects to be 'added' to each other. Creating an origin solid allows a looping-sum over all the meshes.

[4]Rather than manually coding in the reflectivity of each surface along the optical path, the reflectivity has just been set to 0

SiPM z-location. Currently, a model of the actual SiPM board has not been implemented, so a generic plate has been added to the arm which acts as a detector. The surface material is defined by `mfunctions.get_sipm_surface()` and detects 100% of the photons that hit it. The `closed_mirror` solid is the closed 45° gold reflector in the setup. This uses a CAD file from Edmund Optics[5], rather than the one provided by Resonance as the mesh resolution was too low. This meant that the position within the setup (which is encoded in the Resonance file) had to be redefined. The `door1_solid` and `door2_solid` solids are the slit doors for the second slit (the Resonance model has fixed doors). `basex` and `basey` represent the xy profile of the base of the door (constructed from measurements of the doors) – this is then extruded in the z-direction to create the door. These are then added to the setup at the location of the fixed doors. The variable `sep` dictates the separation between the doors[6]. The final solid is the `pmt_solid`. This is simply a blank plate which covers the PMT mounting point and has the QE and FF of the PMT (R8486 Hamamatsu[7])
Once all the special solids have been added to the geometry, the geometry is dumped to a `.pickle` file so that it can be loaded later.

2.4 `analysis.py`

Arguments: `<mode>` `0` – Visualisation mode `1` – Plotting mode

This script handles the analysis of the files output by `simulator.py`. It reads in the contents of the `results/` directory and prompts the user for a selection. The script then creates several markers used to indicate the photon start and end positions. The direction marker shows the rough direction of the beam.
The photon start and end positions are read in from the selected file. Photons containing the `(0x1 << 2)` flag (i.e. the `SURFACE_DETECT` flag) are collected and their positions are checked to see whether they were detected by the PMT (terminate at specific y-value) or SiPM (terminate at specific z-value). The total number of photons and detections read by the SiPM and PMT are printed to the terminal, along with the relative detection rate (i.e. the amount detected compared to the total photons).
The detected positions are then centred about their mean and a scatter plot is created. A Gaussian heatmap of these hits is also created, with the number of bins determined by the Freedman Diaconis rule:

[5] https://www.edmundoptics.com/p/254-x-508mm-pfl-90-off-axis-parabolic-bare-gold-mirror/33560/

[6] Currently the light source is generated in the gap between the doors, so the doors technically are not operational. I've left the slits functional though so that if the monochromator becomes functional then these may be used.

[7] https://www.hamamatsu.com/jp/en/product/type/R8486/index.html

$$\text{binwidth} = 2\frac{\text{IQR(x)}}{\sqrt[3]{n}} \tag{1}$$

for n the length of the dataset x and $\text{IQR}(x)$ the interquartile range.
If the script is instead run in 'render mode' then it will render the geometry and display the photon start and end positions that are selected using the seed file (currently the default is roughly 1 in 10 photons are plotted – this helps reduce the strain on the computer).

3 Breakdown: Supplementary Script (`mfunctions.py`)

3.1 `get_pmt_surface()`

Arguments: *None*

This function generates the `chroma.geometry.Surface` object that represents the properties of the R8486 PMT when called. It loads in the QExFF curve from `qe_curve.dat`, which it uses as the absolute detection efficiency. The reflectivity of the PMT itself is not modelled.

3.2 `get_sipm_surface()`

Arguments: *None*

This function generates the `chroma.geometry.Surface` object that represents the properties of the SiPM when called. Currently, it detects 100% of the light incident on it in the 100-600nm range – no QExFF curve has been implemented for this.

3.3 `get_mirror_surface()`

Arguments: *None*

This function generates the `chroma.geometry.Surface` object that represents the properties of the gold mirrors when called. It uses the absolute reflectivity curve (*see Appendix 1*) provided by Inrad Optics – Note though that gold mirrors are usually used for IR applications and so do not have well documented reflectivities below 200nm. For the 100-200nm range, the reflectivity is constant at $\sim 18.95\%$[8]. Since the data also only specifies the absolute reflectivity, the mirror surface has been set with an assumed specular reflection rate of 99.5% (the other 0.5% of reflections are diffuse).

[8] I have on good authority (Fabrice Retiere) that the reflectivity is almost constant as a function of wavelength in the ultraviolet range, hence why it was chosen even though its

3.4 `get_blacklist()`

Arguments: *None*

This function returns a list containing the mesh numbers which are blacklisted (*see Section 4.3*).

3.5 `get_center()`

Arguments: `mesh_no`	
`mesh_no`	– Mesh number to load in (str)

This function reads in the centers of all the triangles of the mesh and returns the Cartesian center.

3.6 `get_source()`

Arguments: `n wavelength width beamsize`	
`n`	– Number of photons to simulate (int)
`wavelength`	– Wavelength of photons in nm (int)
`width`	– Slit width (in inches)
`beamsize`	– Width of beam (in inches)

This function reads in the centers of all the triangles of the mesh and returns the Cartesian center of that mesh.

3.7 `mirror()`

Arguments: `mesh_no tol regen`	
`mesh_no`	– Mesh number to load in (str)
`tol`	– Tolerance to which the surface mapping is applied (float)
`regen`	– Regenerate mirror properties files (bool)

absolute reflectivity is poor. Edmund Optics said that the reflectivity in this range is predicted to be zero using their model, but they said that realisitically it is probably 'very close to, but not quite zero'. Since I don't have exact numbers, I've kept the reflectivity constant with respect to the known reflectivity at 200nm.

This function performs the surface mapping to the mirrors and returns the surface array (containing `chroma.geometry.Surface` objects) and the color array (hex color). It first checks the existence of the mirror properties file (of the form `mirror_properties_<mesh_no>.pickle` and tries to load the surface and color arrays. If this fails (or the file is not present) then an empty set of arrays are generated. It then checks the length of the arrays against `mesh_resolution` (this is the number of triangles in the mesh) to make sure that the mesh for the mirror contained in `stls2/` matches the properties file. If there is a disagreement (i.e. the stored mesh has a different number of triangles to the mesh used to generate the properties file) then the properties file is regenerated[9].
If the properties file is to be regenerated, then the function checks the centroid of each triangle in the mesh to see whether it must be assigned a mirror surface or an absorbant surface[10]. This is achieved by taking the xy coordinate of each centroid and checking whether it lies on either the internal or external (external only if the mirror is closed) radius of the mirror[11]. If the centroid does not lie on either the external or internal radius then it must be located either on the reflecting face of the mirror or the base of the mirror, and is thus assigned a reflective surface. Surfaces on the radii are assigned non-reflecting surfaces (the base of the mirrors is assigned a reflective surface, but since this is not in the optical path then it does not matter). The surface and color arrays for that mirror are then saved to the properties file (for later use) and are returned.

> *Note: In the current implementation of the simulation, the* `mirror()` *function is not called for the first reflector, but instead the gold surface is assigned to every triangle. This is partly because the body of the mirror (i.e. the non-reflecting parts) have been removed as a result of the increased resolution, and also partly because the sheer number of triangles means that this function takes too long (hours) to generate the config files required.*

3.8 Rotation Matrix Generators: `Rx()`,`Ry()`,`Rz()`

> Arguments: `t`
>
> `t` – Rotation angle in radians (float)

These functions generate the relevant rotation matrices for a rotation of angle `t` about the axis (`Rx()` rotates about x, and so on).

[9] Regenerating these files usually takes a long time – be careful not to accidentally delete or overwrite them.

[10] This is important for the open mirror, as the central portion where the light propagates through is not actually reflective.

[11] The internal radius is 0.248in and the external radius is 0.5in.

4 `config/` Files

4.1 `logo.txt`

This is the ASCII art form of the TRIUMF logo for the simulation UI (`main.py`)

4.2 `seed.dat`

This is the seed file for the random sampling of photons in the analysis. Since a simulation run can contain a large number of photons, it is not beneficial to plot every single photon. This array is a sequence of uniformly generated random floats (between 0 and 1) which is used to determine which photons are plotted – if the number is less than or equal to 1/`resolution` then the photon is plotted. The reason for using a seed rather than a unique number generated each time is because of CUDA's tendancy to fail when a geometry is rendered for the first time (*see Section 5.2*).

4.3 `blacklist.dat`

This is the mesh blacklist and is of the form:

ID	Type	Description	Blacklist

where *ID* is the mesh number, *Type* can be interpreted as the name of the mesh, *Description* is a short description of the mesh and *Blacklist* is a boolean and determines whether or not the mesh should be skipped (`1` for skip). If a mesh has *Type* = *undefined* then the mesh is just a standard[12] component. If instead it has *Type* = *defect*, then the mesh is defective[13] and is automatically blacklist.

4.4 `geometry.pickle`

This is the `.pickle` file containing the setup geometry.

4.5 `gold_reflectivity.dat`

This is the reflectivity data for the gold mirrors (*See Appendix 1*).

4.6 `qe_curve.dat`

This is the QExFF curve for the Hamamatsu PMT.

[12] i.e. not necessarily important.

[13] A defective mesh is one which is composed of a single or small number of triangles. These are usually fragmented meshes – any mesh with a filesize less than 1kB is considered defective.

4.7 `lamp_intensities.dat`

This is the intensity vs. wavelength data for the deuterium lamp. Currently, this has not been implemented into the light source for the simulation[14].

4.8 mirror_properties_<mesh_no>.dat

These files contain the color and surface arrays for the relevant mirror meshes.

4.9 `vars.txt`

This file contains the setup parameters (i.e. the slit width) so that `simulator.py` can determine whether or not the geometry needs to be regenerated.

5 Current Limitations

5.1 No Diffraction Process

It does not seem[15] that Chroma is capable of emulating diffraction processes, such as grating and slit effects. The photons seem to be treated as 'bullets', wherein they have an initial position and direction, which does not change (unless refraction occurs, which is modelled by Chroma). It might be possible to solve this using some trickery with the `chroma.geometry` class, but I have not looked into this as of yet. Currently the simulation 'mimics' diffraction by sampling the photon direction according to:

$$\frac{I}{I_0} = \text{sinc}\left(\frac{D}{\lambda}\cos\theta\right) \tag{2}$$

where I/I_0 is the relative intensity, D is the slit width and λ is the wavelength.

> *Note: Python's numpy library has an in-built sinc function, but this is the singal-processing form (as opposed to the mathematical form):*
>
> $$\text{sinc}(x) = \frac{\sin(\pi x)}{(\pi x)} \tag{3}$$

Without diffraction, the monochromator functionality cannot be properly coded in and thus there is no wavelength selection (via the monochromator, at least).

[14] This is because I did not exactly know how the width of the slit affected the wavelength selection (i.e. I could not make a reasonable approximation as to how much of the more of the spectrum is allowed through per mm of width increase).

[15] I can't seem to find anything that would hint to this in the source code, but oddly the beamshape at the SiPM seems to be wider than the iris it passes through.

5.2 Initialization Errors (`cuInit error`)

When writing and testing the code for the simulation, it became apparent that the first call to CUDA to render an object always threw a `LogicError` and aborted, but rerunning the script immediately after without affecting the geometry allowed it to render[16].

A Reflectivity Curves

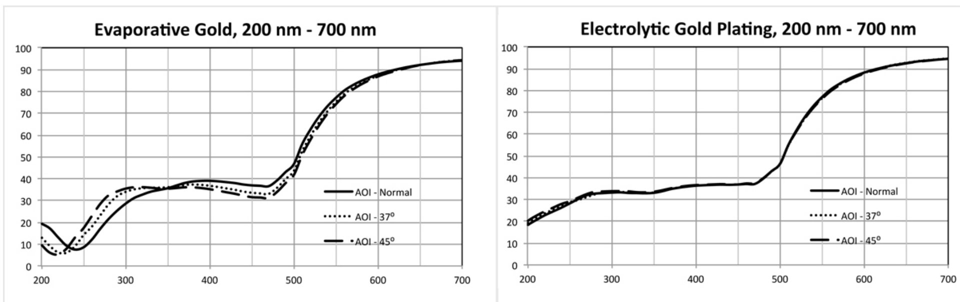

Figure 1: Inrad Optics reflectivity curves for the gold reflectors. The left curve is for gold coatings by evaporation deposition and the right curve is for gold coatings by electrolytic plating. AOI here represents the angle of incidence to the mirror surface. In the setup the AOI is 45° for both mirrors.

[16]This is why the analysis script uses pre-generated seed files to randomly select photons as opposed to generating them in-situ