# Light and Charge simulation

Version 0.1

ToDO's explain the light simulation in rough detail

October 6, 2022

## Installation

The install process should be very straight forward. It requires a `root` install, tested with `root 6.26/06` but should be also useable with root5. After cloning the directory, make sure that root is active (`root-config --version`). If not, source your favourite root build. Then, in the main folder just run `make`.

Sometimes this process can fail. The most common cause for this is the existence of older library files (`*.o`), that where compiled with a different root install. In this case, remove all library files (`yes | rm *.o`) and run `make` again.

- **TL:DR** : Run `make`.

## Usage

`./analyze_light <G4_input_file> <SiPM_Placement_File> <Size of PD (cm)> <OutputFile>`

- `G4_input_file`: Result from the Geant4 simulation (`qpixg4`)
- `SiPM_Placement_File`: Discussed below. Center of each detection element
- `Size of PD`: The full width of a single detection element
- `OutputFile`: Name of the output file
- Optional arguments are:
  - `--number` Run over a subset of events (first n) instead of all in the file
  - `--exclOut 0` Disables the saving of the G4 input data to the output file
  - `--charge 1` Enables the charge simulation including diffusion
  - `--diffusion 0` Disables the diffusion during the drift

### Output File

The output of the charge and light simulations are stored in the `ScintSim_tree`. The charge and light objects are of type `vector<vector<double>>`. The first axis corresponds to PD ID, see below, the second axis runs along the number of detected photons in this PD, and stores the times. To access the number of photons detected in PD 17, one needs to run

```
total_time_vuv->at(17).size()
```

If one wants to get the hit times of all photons detected on PD 17, you have to

```
for(int photonIt = 0; photonIt < total_time_vuv->at(17).size(); photonIt++){
    time = total_time_vuv->at(17).at(photonIt);
}
```

Same holds for the charge output.

- **TL:DR** : Returns `vector<vector<double>>` of PD-IDs and hit time for each photon/electron
- Example in `Docs/ExampleMacro`

## Placement Files

The idea of the code is to be able to run light simulations with arbitrarily placed light detectors. For this, a so called placement file is required. These placement files give the center coordinates of the placed light detectors. The size of the detectors are determined in a later stage. Below an example of such a placement file is presented:

```
0 5.0 5.0 0 1 3
1 5.0 15.0 0 1 3
2 5.0 25.0 0 1 3
3 5.0 35.0 0 1 3
4 5.0 45.0 0 1 3
5 5.0 55.0 0 1 3
6 5.0 65.0 0 1 3
7 5.0 75.0 0 1 3
8 5.0 85.0 0 1 3
9 5.0 95.0 0 1 3
10 10.0 5.0 0 1 3
11 10.0 15.0 0 1 3
12 10.0 25.0 0 1 3
13 10.0 35.0 0 1 3
14 10.0 45.0 0 1 3
15 10.0 55.0 0 1 3
16 10.0 65.0 0 1 3
17 10.0 75.0 0 1 3
18 10.0 85.0 0 1 3
19 10.0 95.0 0 1 3
```

The first column gives the ID of each detector. These have to be unique. Next, the `x`, `y` and `z` coordinates are given. In the case above, we place 20 photo detectors on the `z=0` plane. The first ten are all placed with the same `x` coordinates and distanced 10 cm in the y-direction. The last ten are moved 5 cm along the x-axis. See the example in figure 1. The units are understood to be in cm.

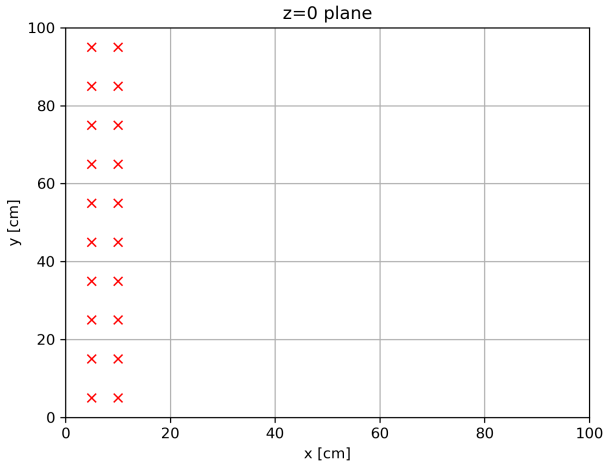


Figure 1: Placements for the example placement file above

The 5th variable is the detector type. This variable is currently not used. Its intension is to be able to implement different types of photo detector, meaning non-rectangular ones, i.e. circular PMTs. The last variable is the direction in which the active surface is pointing. If the detector sits on the `z=0` plane, it probably faces along the z-axis. This is noted with the value `3`. For detectors facing along the `y` axis, this value should be `2`, and for detectors facing along the `x` axis, `1`.

To generate these files, a helper script is supplied in `PlacementFiles/generate_detpos.py`. It generates a grid placement of a given distancing. Currently the same distancing is assumed for the `x`, `y` and `z` coordinates. But this can be adjusted easily, when required. The way to do that would be to implement different distances for each direction. Currently you comment out the corresponding parts that you don't want any PD placed at.

To then get the placement file you run `python generate_detpos.py > placmenFileName`. To verify if this placement

file you can adjust `1_plot_detector_locations.py` to plot the detectors in a 3D plot. Things to adjust here are read in placement file in

```
with open('../SoLAr_10cm_FieldCage.txt', 'r') as csvfile:
```

and then a little bit below the sizes of the detectors you want to plot. In the case we want to have $2 \cdot 2cm^2$ pixels, the placement file above would produce the example in figure 2. Then run `python 1_plot_detector_locations.py` and you will see a 3D plot of the placements and sizes of the detector you created.

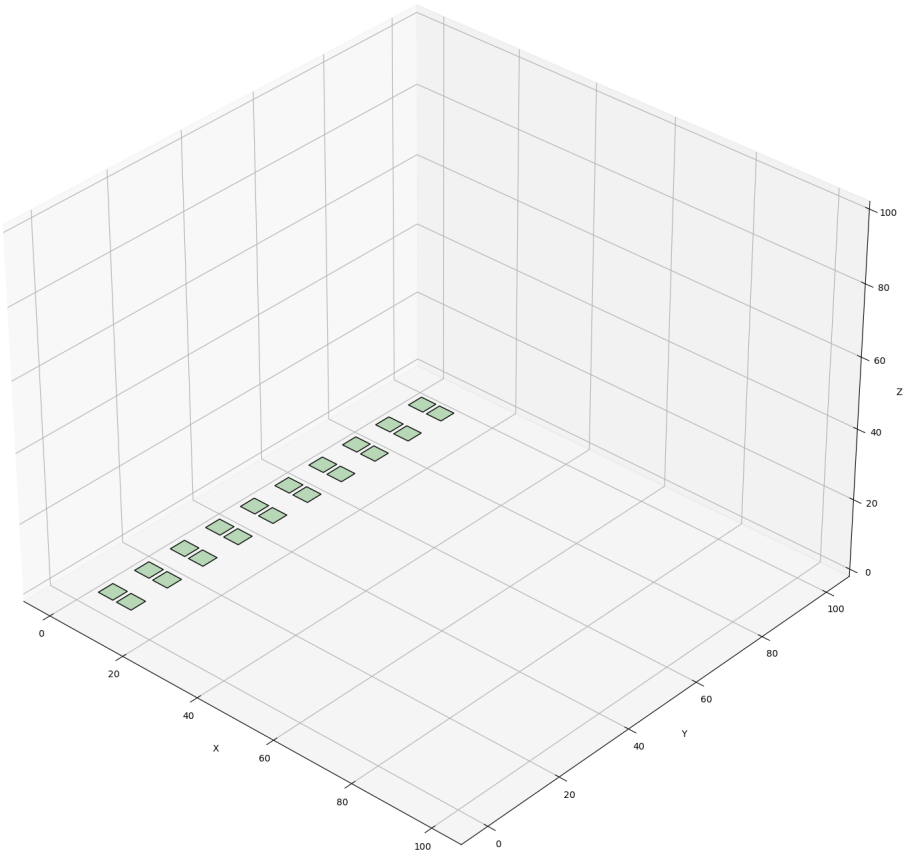


Figure 2: Example of the placement file on the X-Y plane with $2 \times 2cm^2$ pixels

## Simulation procedure

### Light

- Based on this Paper
- Removed the visible light and enabled placement of PD on all planes, implemented LArQL model
- Split the total amount of photons produced in an event by the fractional solid angle of each photo-detector element

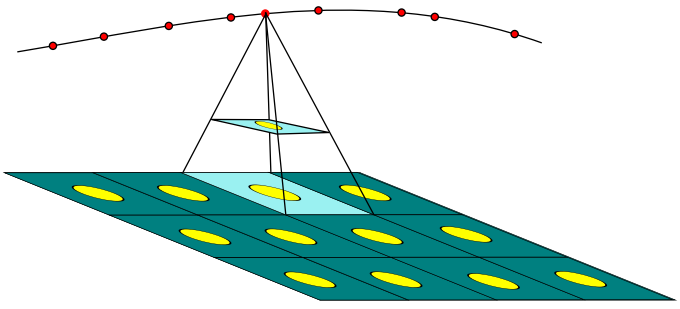

Figure 3: Conceptual Drawing of the Semi Analytical Model

- Drawbacks: Scales linearly with the number of PD.

### Charge

Similar approach, where the number of electrons are calculated from the hits using LArQQ. These electrons are assumed to be produced at the hit-point. Then their starting position is smeared in the x-y plane according to the $D_T$ drift constant, with the drift-time calculated as $t_{drift} = z/v_{drift}$. Along the z-axis the electrons are smeared with the $D_L$ drift constant in the same manor. These starting positions are then stored in a vector. Here we assume that the drift-direction is along the z-axis, and the charge-readout sits on the z=0 plane. In the main event loop we loop through all the produced starting points, figure out if a PD/CD sits below this point. If yes, a time entry is produced according to $t = z/v_{drift}$.

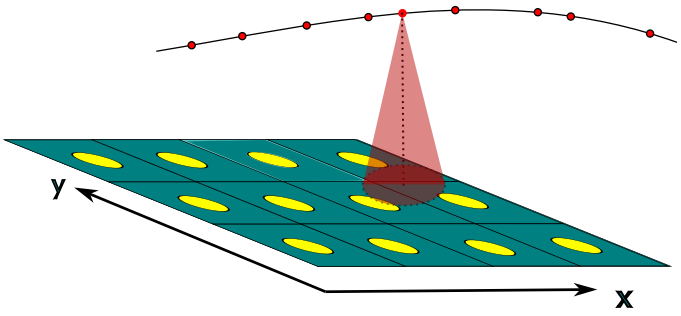


Figure 4: Conceptual Drawing of the Charge Simulation

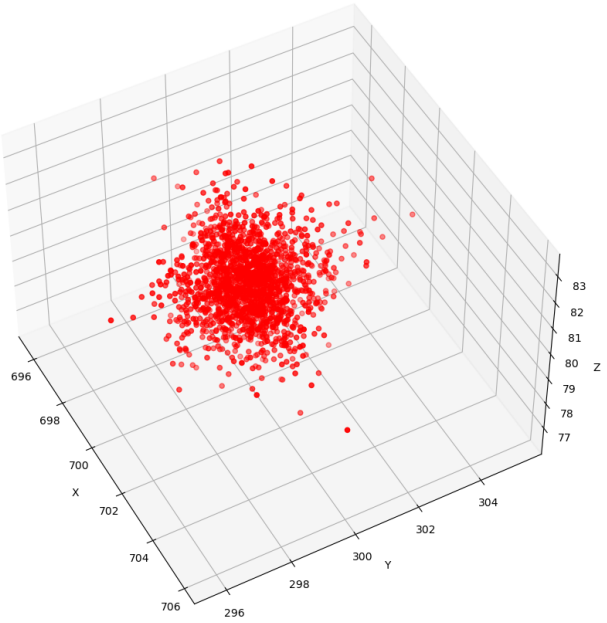


Figure 5: Starting Positions of electrons of a single energy deposit after smearing

- Current Status: The runtime is a major problem. This can be solved by the following things:
  - The smearing currently calls gRandom 4 times for each produced electron. This can be speed up by calling `RndmArray` once for each hit.
  - One can multithread the loop over the number of electrons in the beginning
  - One can multithread the loop over hits in the down-projection part of the code (~line 216)
  - One could assume a readout that covers the full plane, and just project down the x-y coordinates. This would alleviate the need for the loop over the detectors, which would be a major performance improvement. It comes with its own drawbacks. Could be implemented as an alternative function.
  - The main loop can also be rewritten in a different way evading the loop over loops.
- If code should be used long term, a refactoring in its own class would be necessary