Photon simulations of camera view (guidebook)

Wei Zha

Northwestern University

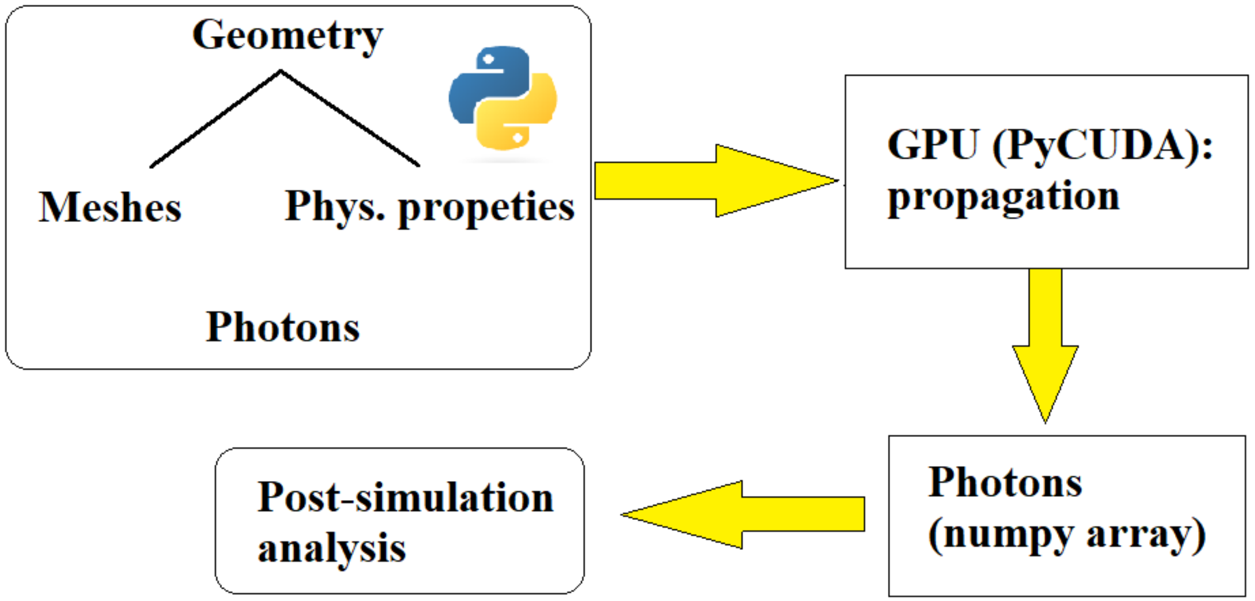
Apr / 2023

Table of contents

What is chroma?

A python-based package aiming at quick photon simulations using GPU (CUDA, or PyCUDA, specifically)

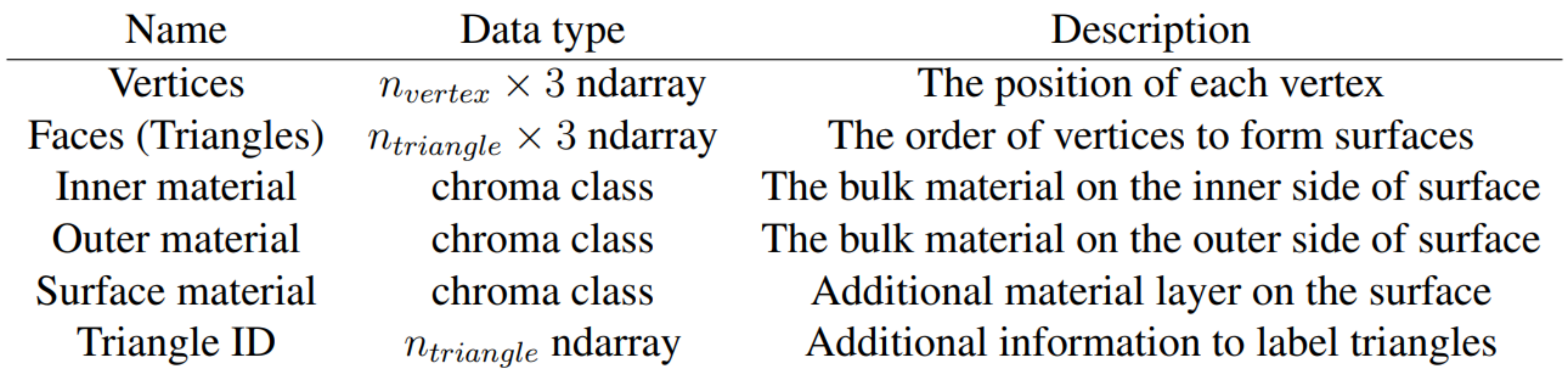
- G4/ROOT friendly
- Faster than G4: 10M photons in 15-20 min
- Flexibility of python



Surface model of chroma

- **Resembles Geant4** in geometry construction but with **surface model**: each physical object is made of a mesh.

Table 2.1: Table of meshes' attributes in chroma



Name	Data type	Description
Vertices	$n_{vertex} \times 3$ ndarray	The position of each vertex
Faces (Triangles)	$n_{triangle} \times 3$ ndarray	The order of vertices to form surfaces
Inner material	chroma class	The bulk material on the inner side of surface
Outer material	chroma class	The bulk material on the outer side of surface
Surface material	chroma class	Additional material layer on the surface
Triangle ID	$n_{triangle}$ ndarray	Additional information to label triangles

Steps to run a sim

Also resembles Geant4…

a) Construct a *geometry*, including objects with mesh and their positions

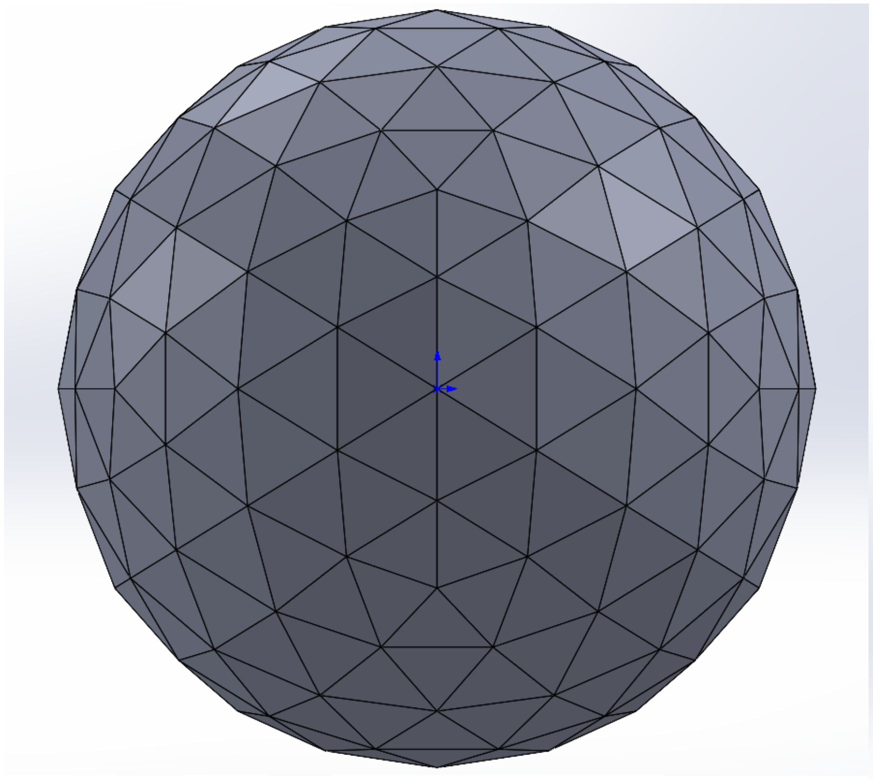
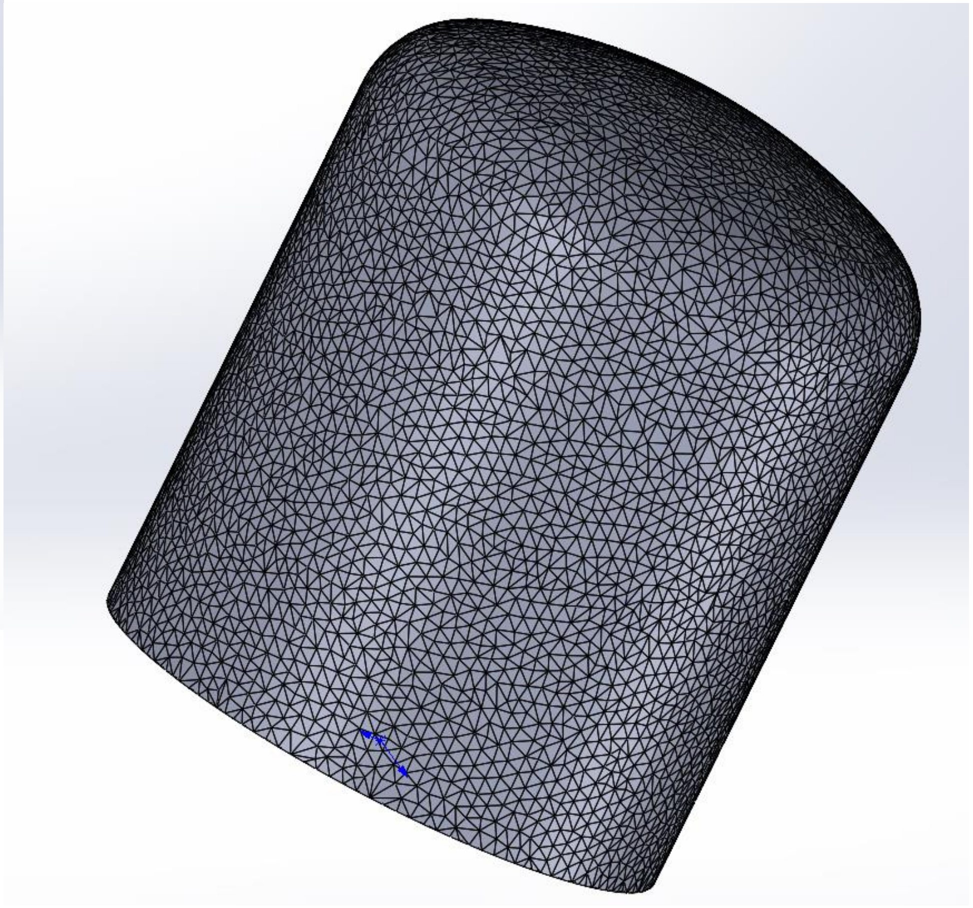
b) Define *materials*. Restriction: for every photon, its starting mesh material should be the same with ending material (see chroma whitebook for details) *Refer to materials.py in sim repository*

c) Combine *material* and *geometry* as a *detector*. *Refer to detector_construction.py*

d) Write your own *light source* (photon generation function). *Refer to source.py*

e) All above combined by using *simulation* class to run events. *Refer to simulation_old.py as an example*

Simulation in chroma resembles that in Geant4 a lot, so it is very helpful to read or run a Geant4 example before using chroma.



Bubble and jar's meshes viewed in Solidworks

Installation of chroma

Chroma and installation repository on Github.

Chroma works particularly on Linux platform. Of course, you will need at least 1 GPU core to run it.

- The easiest way to run it in a Linux machine is using *docker* to pull the image provided by the maintainer of chroma.
- For Windows and macOS users, we have a roundabout way: install and use chroma in a Linux virtual machine on cloud platforms. Refer to the installation repository for directions in this case.
- macOS users could also try to directly install it or use docker, but I haven’t tested if it works.

After installation, one could try to run simulation_old.py or benchmark.py to check it. An easier but not-so-secure way is trying to **import chroma** in python.

Details / tips in chroma installation

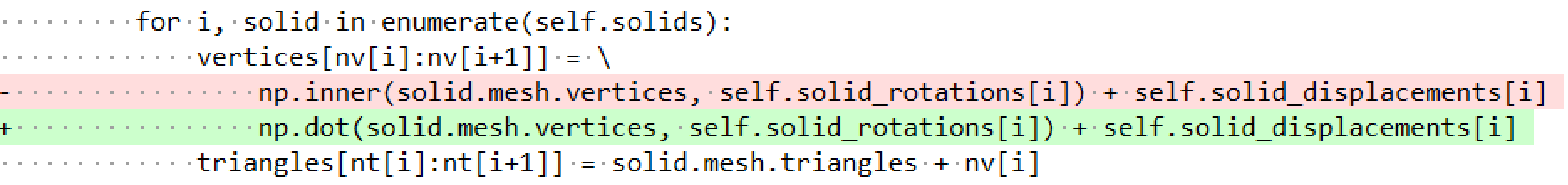
- Chroma uses **conda environment**. It is necessary to install anaconda at first.
- GPU, CUDA drive and toolkit should be the next checkpoint. There is a very detailed guide for CUDA installation on their official website. Make sure you have both drive and toolkit working, by typing *nvidia-smi* (hardware) and *nvcc --version* (software) in bash.
- Be careful about python version and environments, especially if you are using a personal computer. Python 3.7 (anaconda 2019.10) is the environment I worked on.
- Necessary site-packages / software: Geant4; ROOT; G4py; pygame; pycuda. Use *pip* and *conda install* to deal with the Python site-packages – they are usually useful. (pip preferred)
- Read the bash output. There might be errors in lines between.

IMPORTANT

- Before chroma installation, note we switched to conda environment by using *sed -i 's/VIRTUAL_ENV/CONDA_PREFIX/g' setup.py*

This step is included in **chroma.patch** so you don't need to do it.

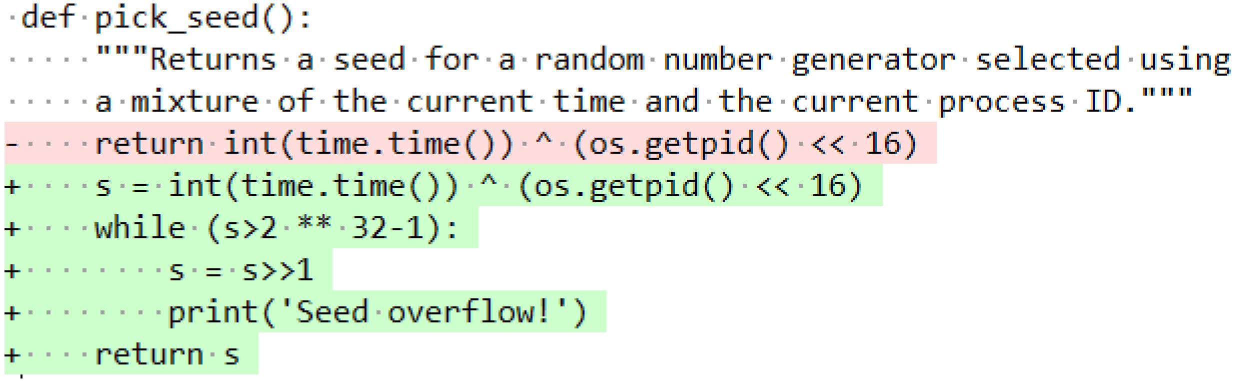
- The chroma I use is a bit different from the original version. Specifically, 2 places are modified:



```
        for i, solid in enumerate(self.solids):
            vertices[nv[i]:nv[i+1]] = \
-               np.inner(solid.mesh.vertices, self.solid_rotations[i]) + self.solid_displacements[i]
+               np.dot(solid.mesh.vertices, self.solid_rotations[i]) + self.solid_displacements[i]
            triangles[nt[i]:nt[i+1]] = solid.mesh.triangles + nv[i]
```

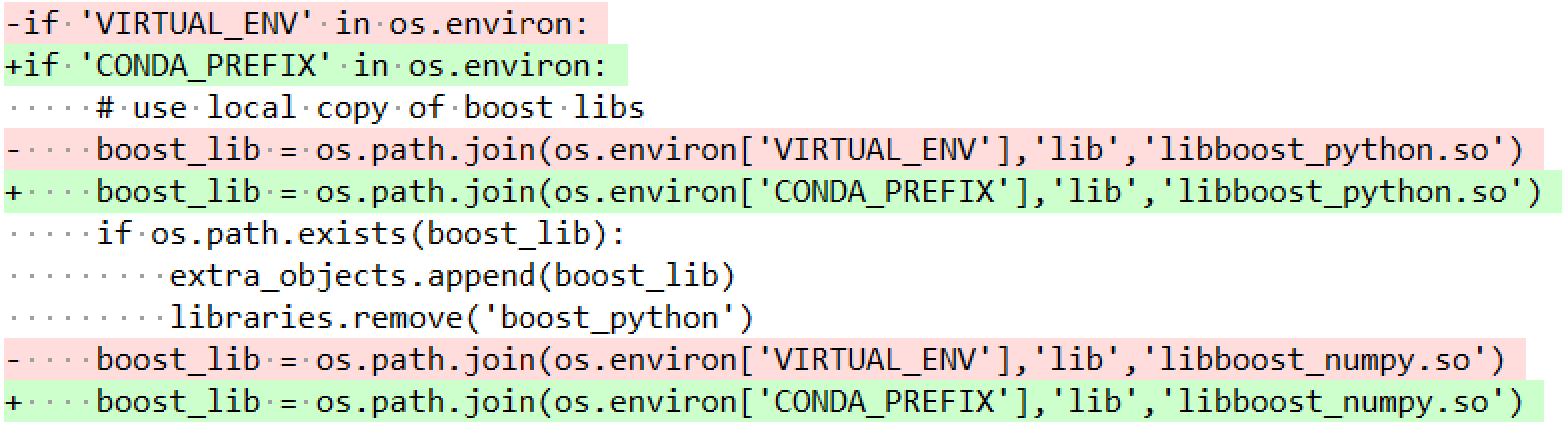
  - Definition of rotation matrix in geometry.py. I prefer the version defined in Wikipedia. Turned np.inner() to np.dot().

It is mandatory to apply the above changes by using *patch -p1 < chroma.patch*, otherwise you will need to follow their definition of rotation matrix (and math will be totally different!).



```
 def pick_seed():
     """Returns a seed for a random number generator selected using
     a mixture of the current time and the current process ID."""
-    return int(time.time()) ^ (os.getpid() << 16)
+    s = int(time.time()) ^ (os.getpid() << 16)
+    while (s>2 ** 32-1):
+        s = s>>1
+        print('Seed overflow!')
+    return s
```

- Pick_seed() in sim.py. I fixed the potential bug of seeding overflow.



```
-if 'VIRTUAL_ENV' in os.environ:
+if 'CONDA_PREFIX' in os.environ:
     # use local copy of boost libs
-    boost_lib = os.path.join(os.environ['VIRTUAL_ENV'],'lib','libboost_python.so')
+    boost_lib = os.path.join(os.environ['CONDA_PREFIX'],'lib','libboost_python.so')
     if os.path.exists(boost_lib):
         extra_objects.append(boost_lib)
         libraries.remove('boost_python')
-    boost_lib = os.path.join(os.environ['VIRTUAL_ENV'],'lib','libboost_numpy.so')
+    boost_lib = os.path.join(os.environ['CONDA_PREFIX'],'lib','libboost_numpy.so')
```

- Effect of *sed* command in setup.py
- Details can be found in **chroma.patch.**

If the patch gets out of date, do it manually – The essential lines are listed above. Otherwise, you may learn to use *diff* to make a new patch on your own.

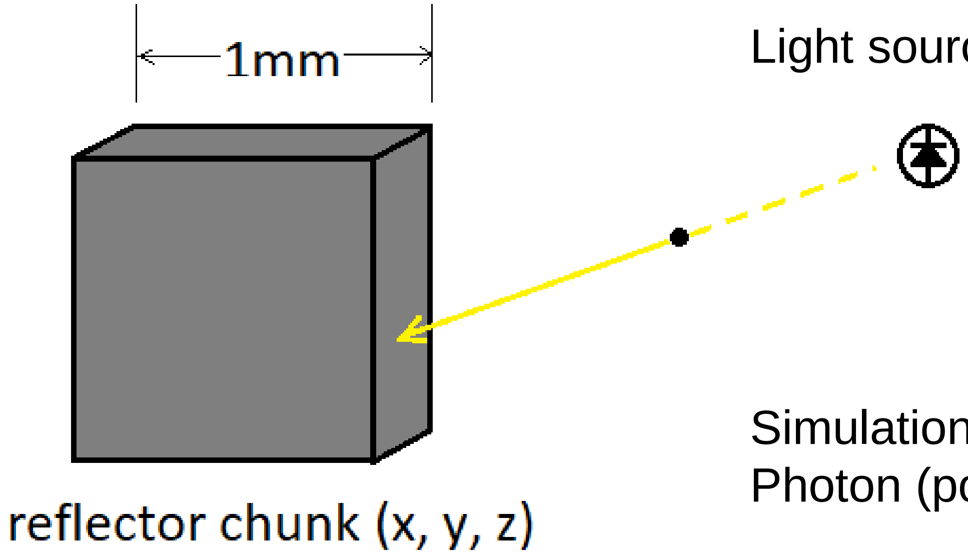
My works on camera simulation

We want to use brute force to simulate images of camera view – requires large numbers of photons being simulated.

- The crude thought: generate photons like real light source, set camera windows as detecting — turned out not working: very few of them reach the camera.
- Separate into 2 steps:
    - Step 1: real light source, detecting reflector
    - Step 2: photons start at camera, go into an illuminated “room”

- comment: applies some ray-tracing idea, by treating the reflectors as majority of environment illumination

Step 1: illumination map



Light source: LED

Simulation results:
Photon (position, direction)

Binned w/ the whole space separated into Cartesian chunks

Stored data output:
3-d ndarray (x, y, z, n) ~ 10Mb

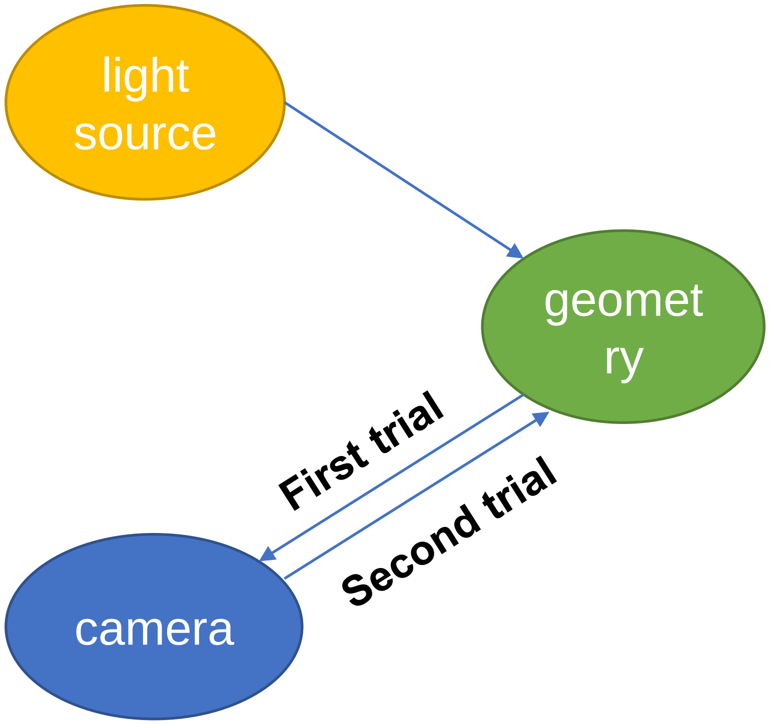
Step 2: simulating the image

Want: at least 10^3 of photons per pixel to finally form a picture

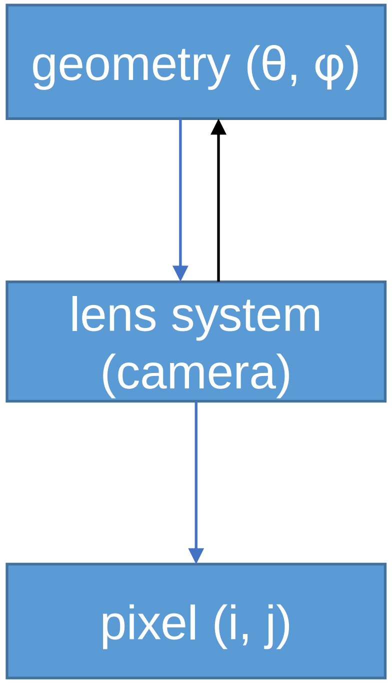
Initial trial: treat reflectors as light source

Result: only 1% of photons reach the camera
~10 photons per pixel simulated

- Photons need to be ‘effective’



Step 2: simulating the image



Invert the process!

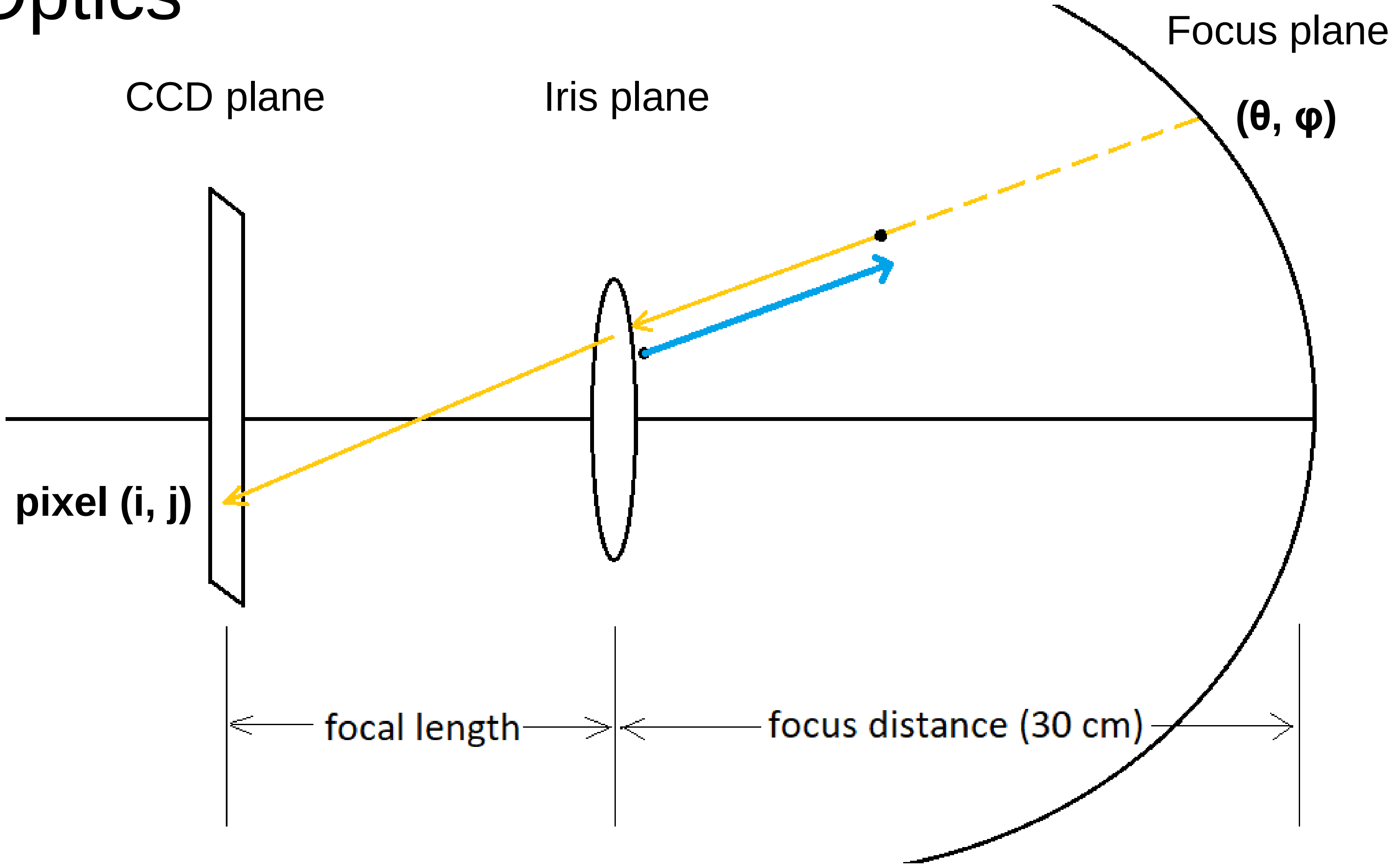
γ at camera (sapphire window) → Geometry

Simulation results: Photon (position, direction)

Position → intensity (n) in illumination map
→ add to intensity of pixel (i, j)

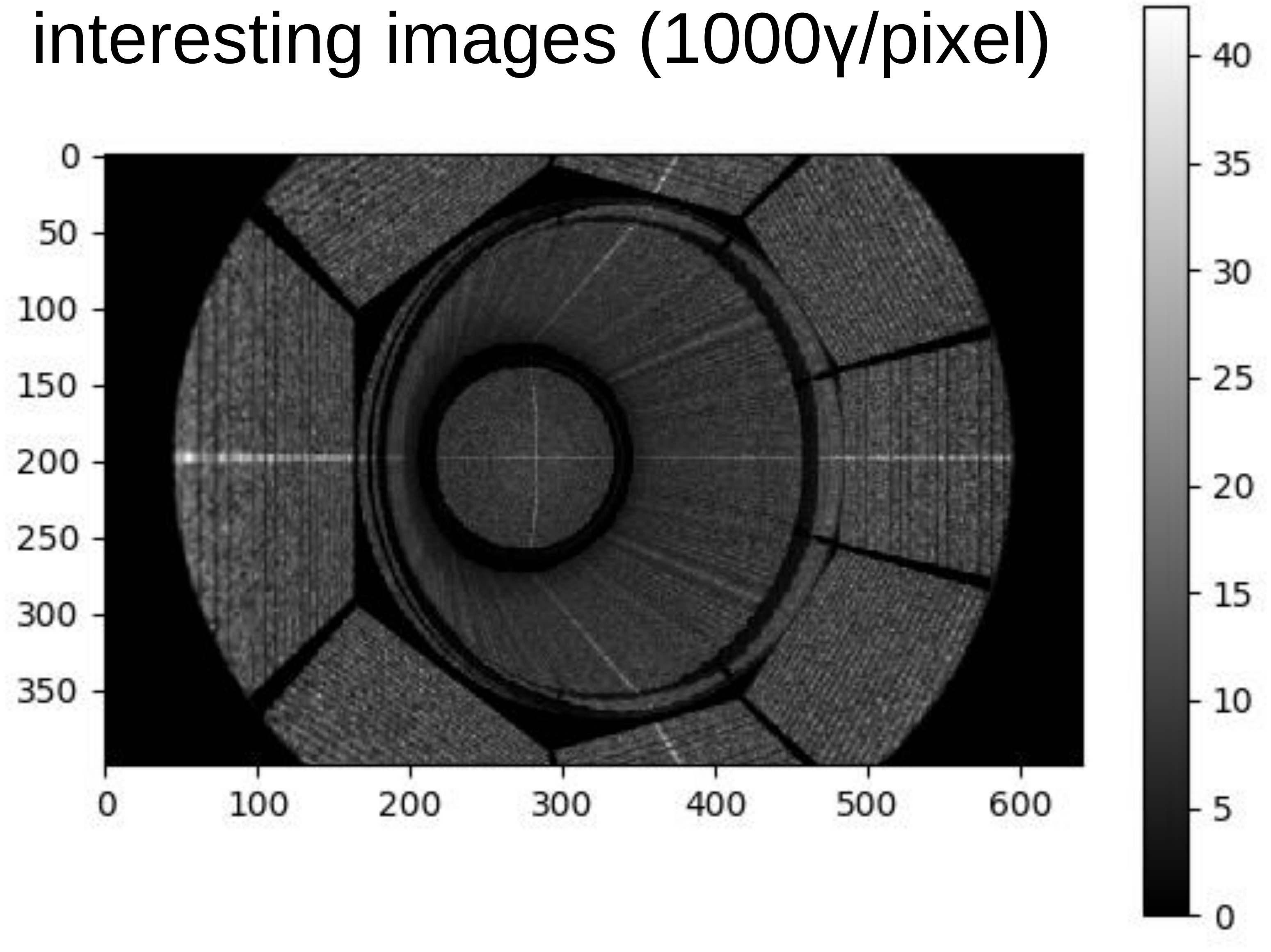
→ map of intensity at pixels

Optics

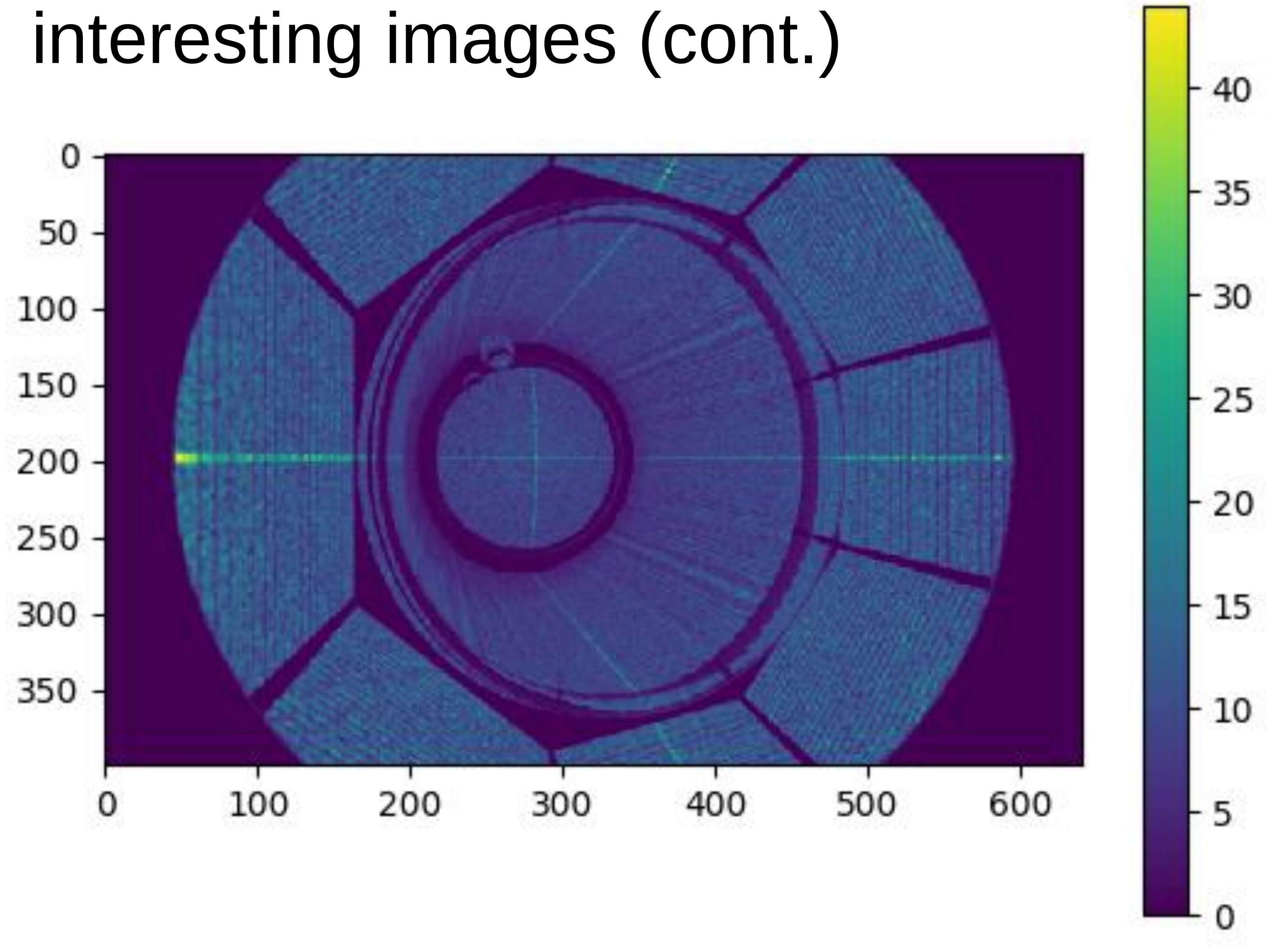


---- Math photon
---- actual photon

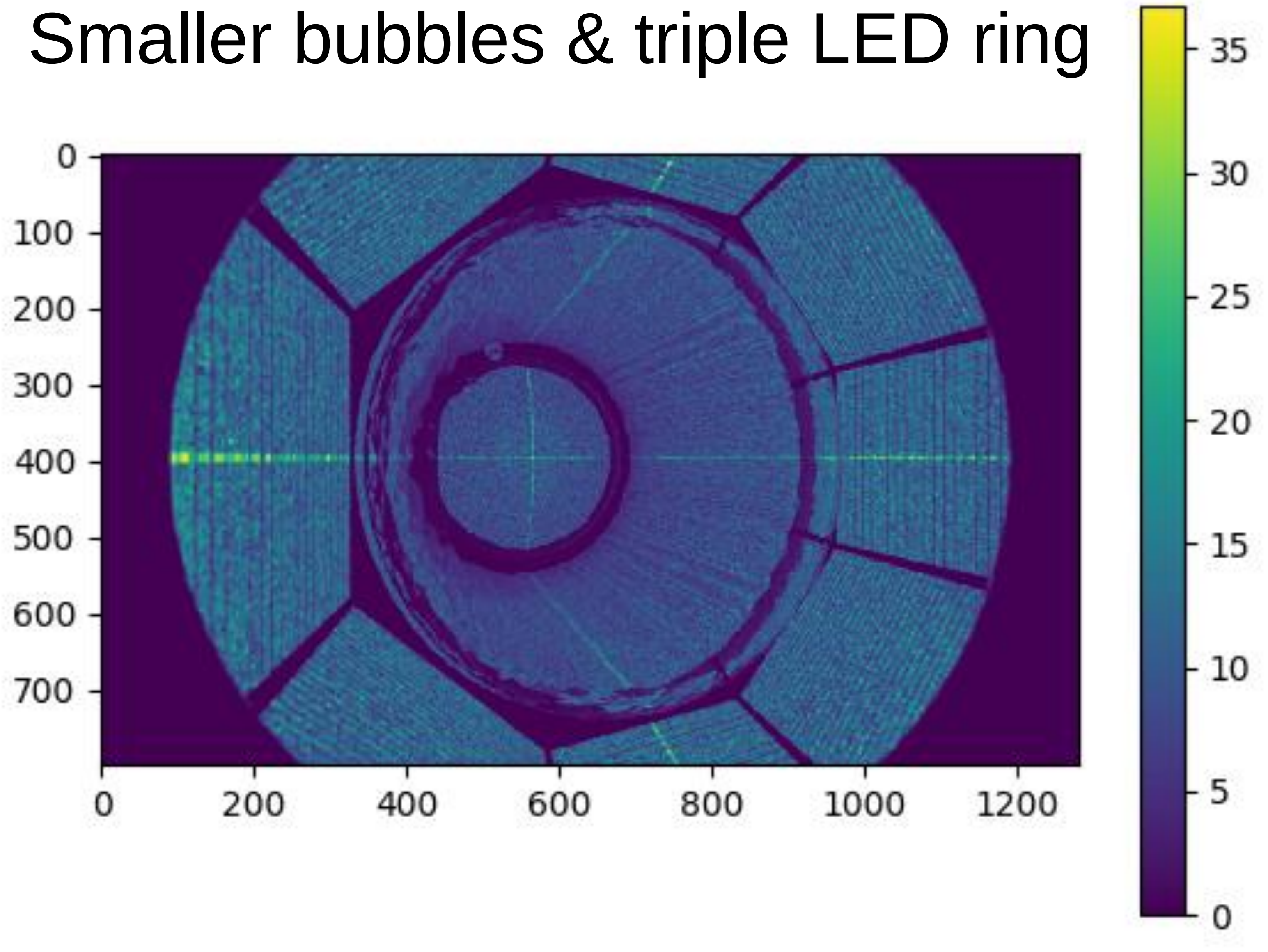
interesting images (1000γ/pixel)



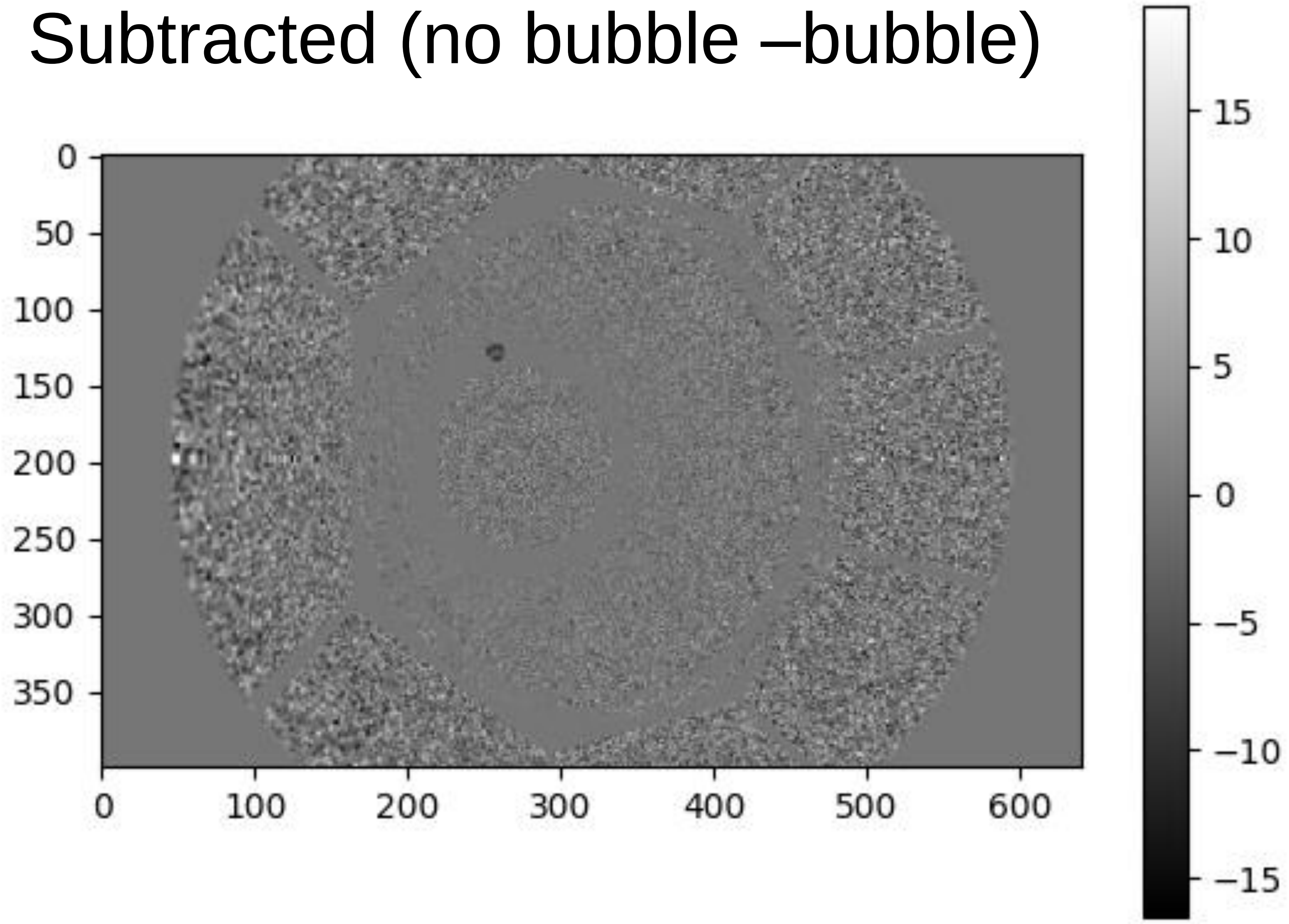
interesting images (cont.)



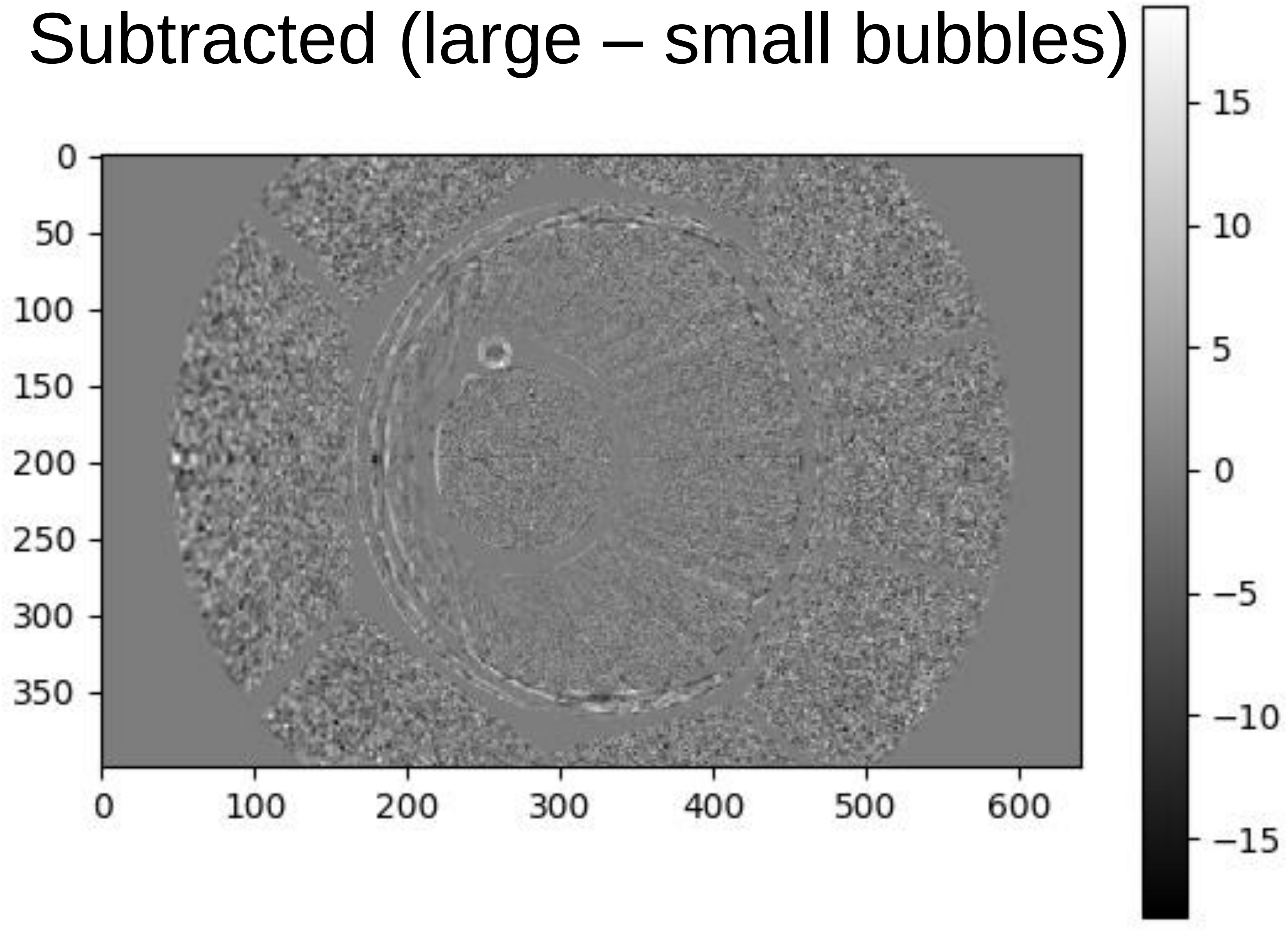
Smaller bubbles & triple LED ring



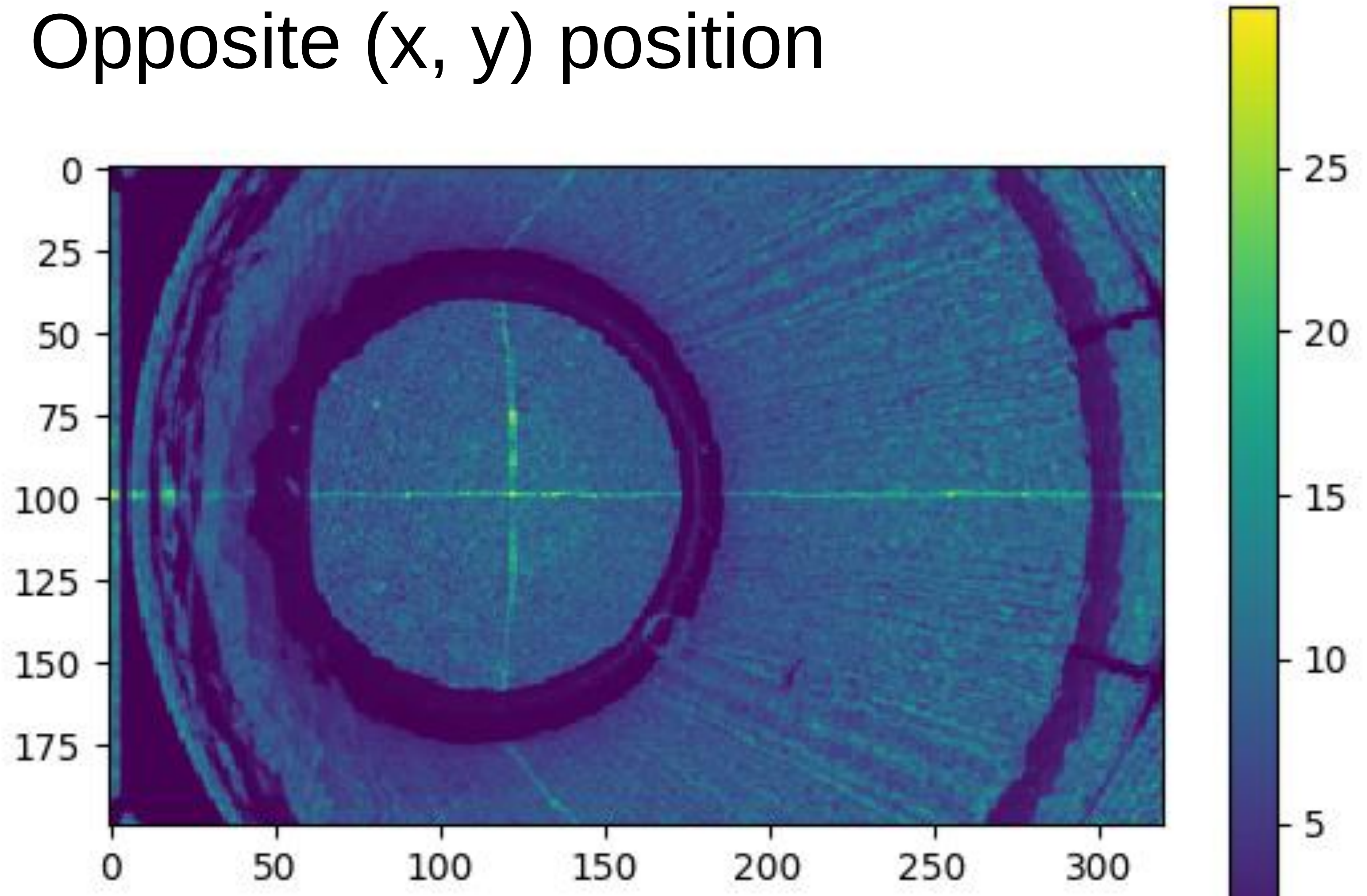
Subtracted (no bubble –bubble)



Subtracted (large – small bubbles)



Opposite (x, y) position



Some numbers

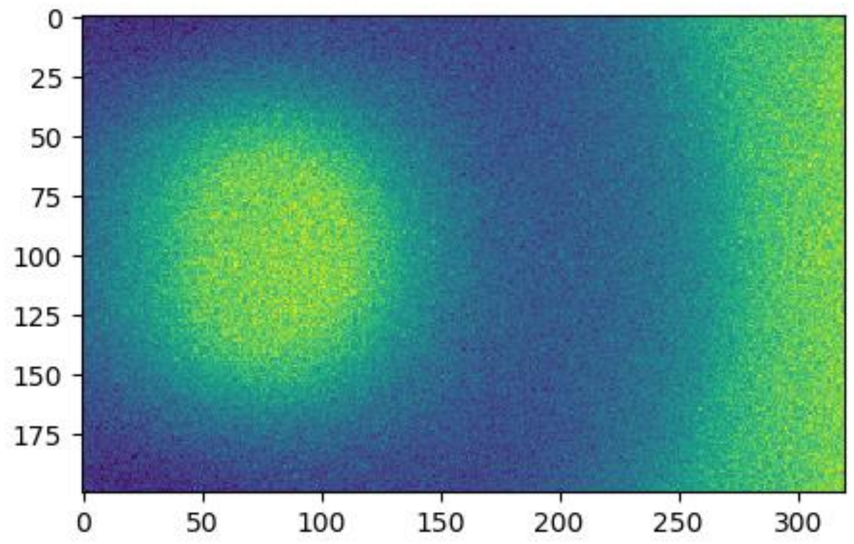
- Bubble: gas Ar w/ index of refraction ~1
- R =1cm / 0.5cm

- Pixel: 3um * 3um; For quick image sims, used 6um*6um

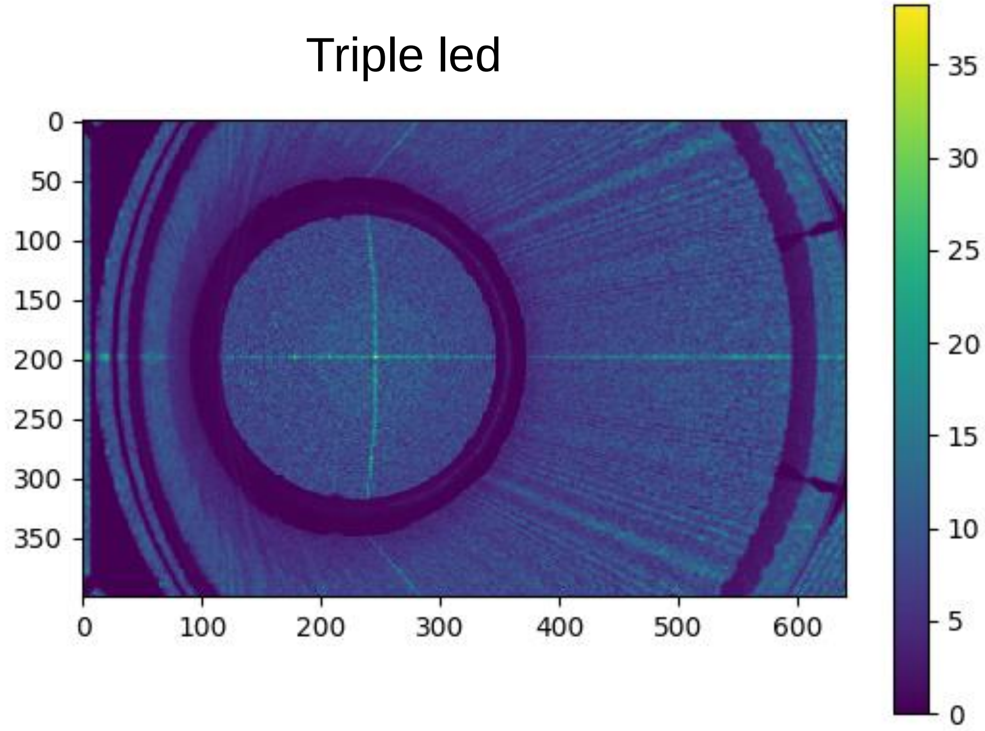
- Focal length: 1.65mm; f-number: 2.8
- Iris: 0.03mm; focal length / f-number

Some failed / unused results

Large iris

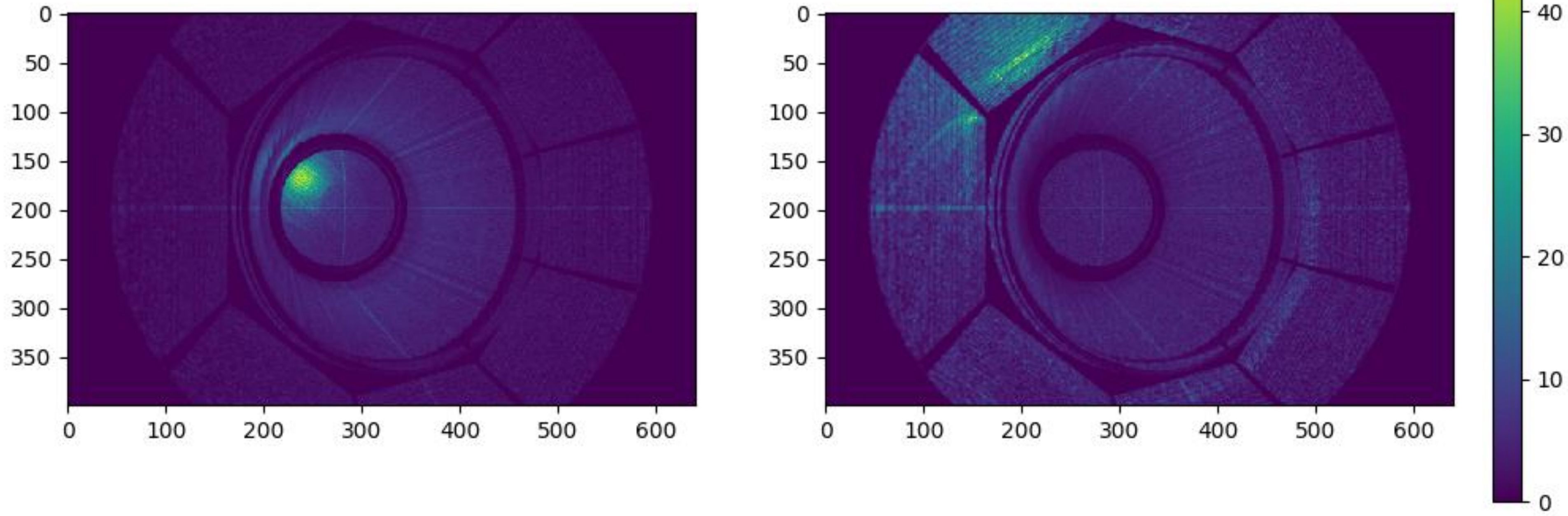


Triple led

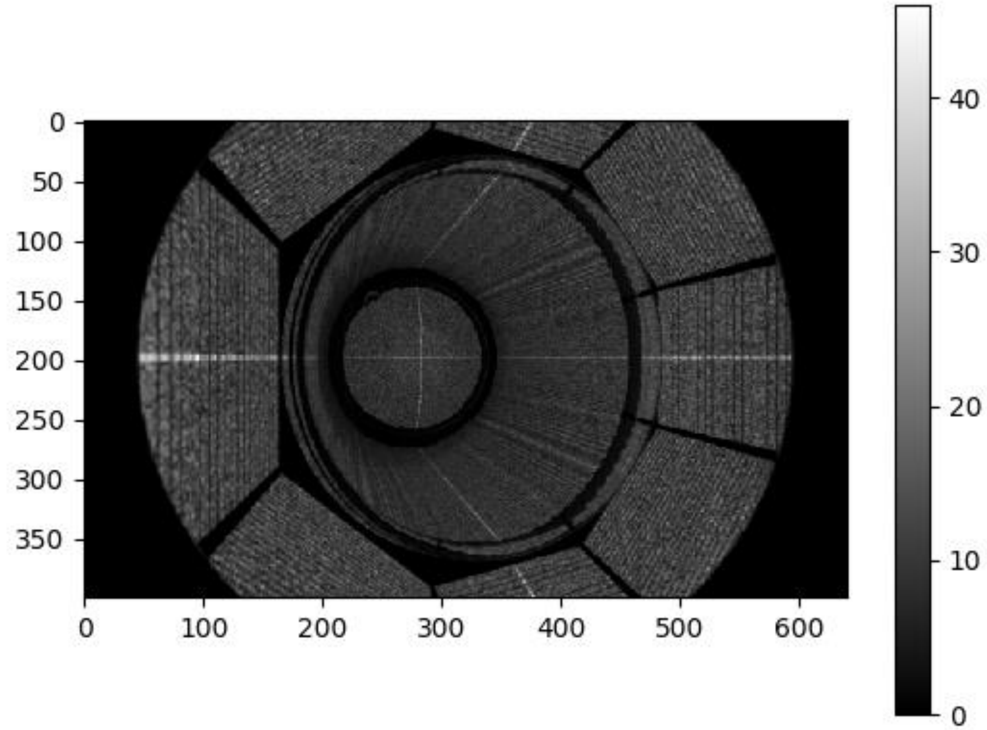


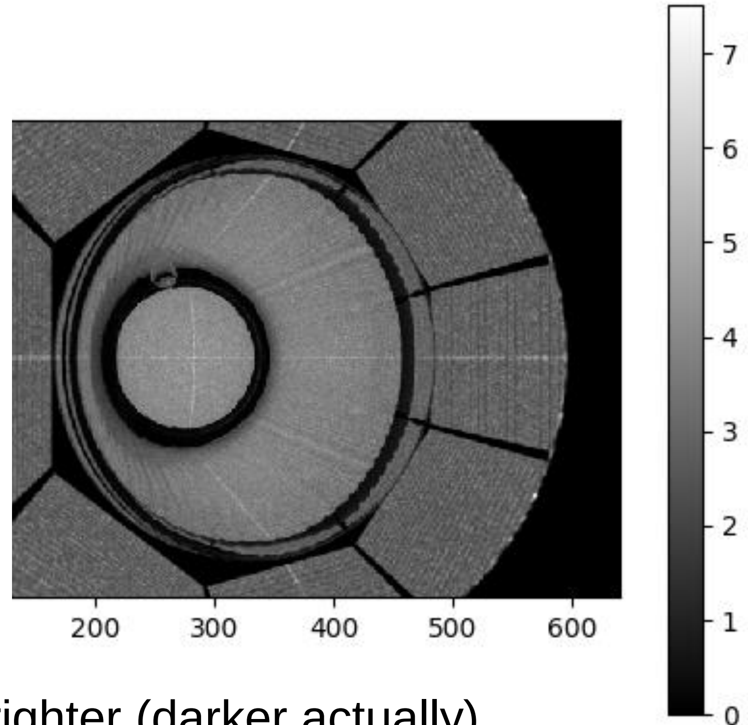
Point source

As said, I suspect the strips being generated by Cartesian mapping



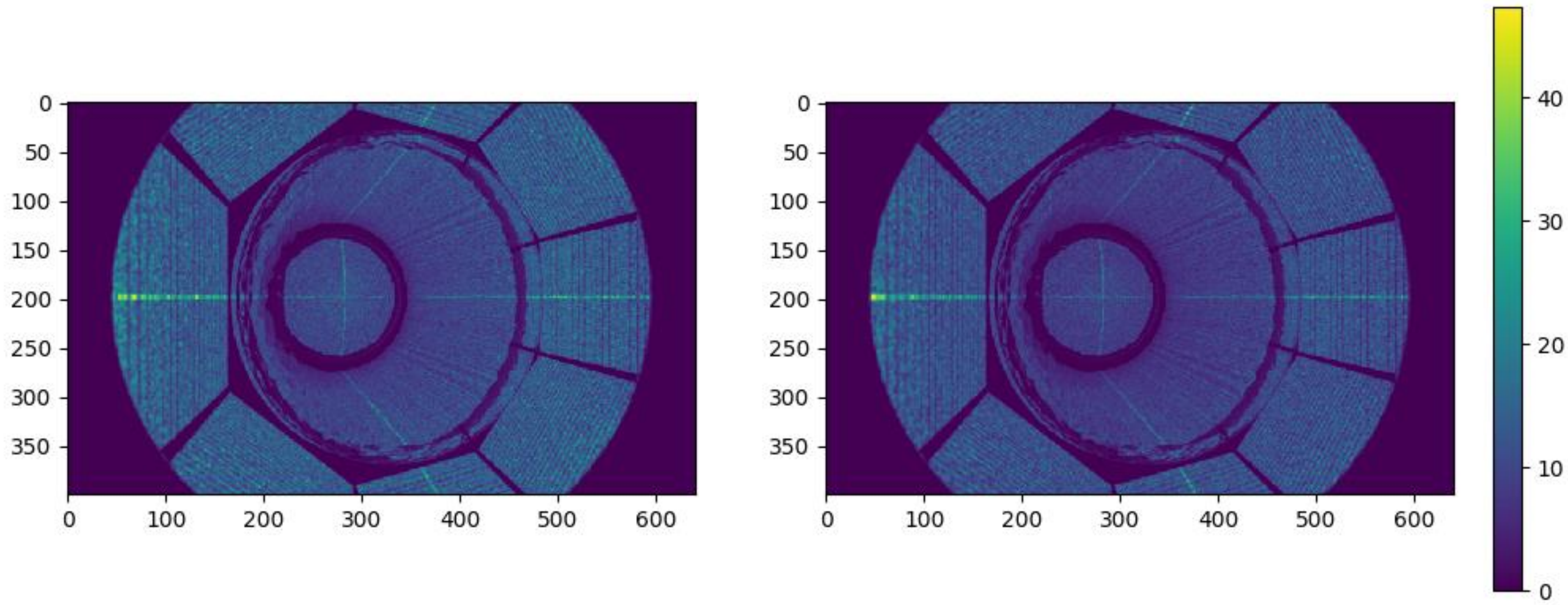
lower & upper bubble



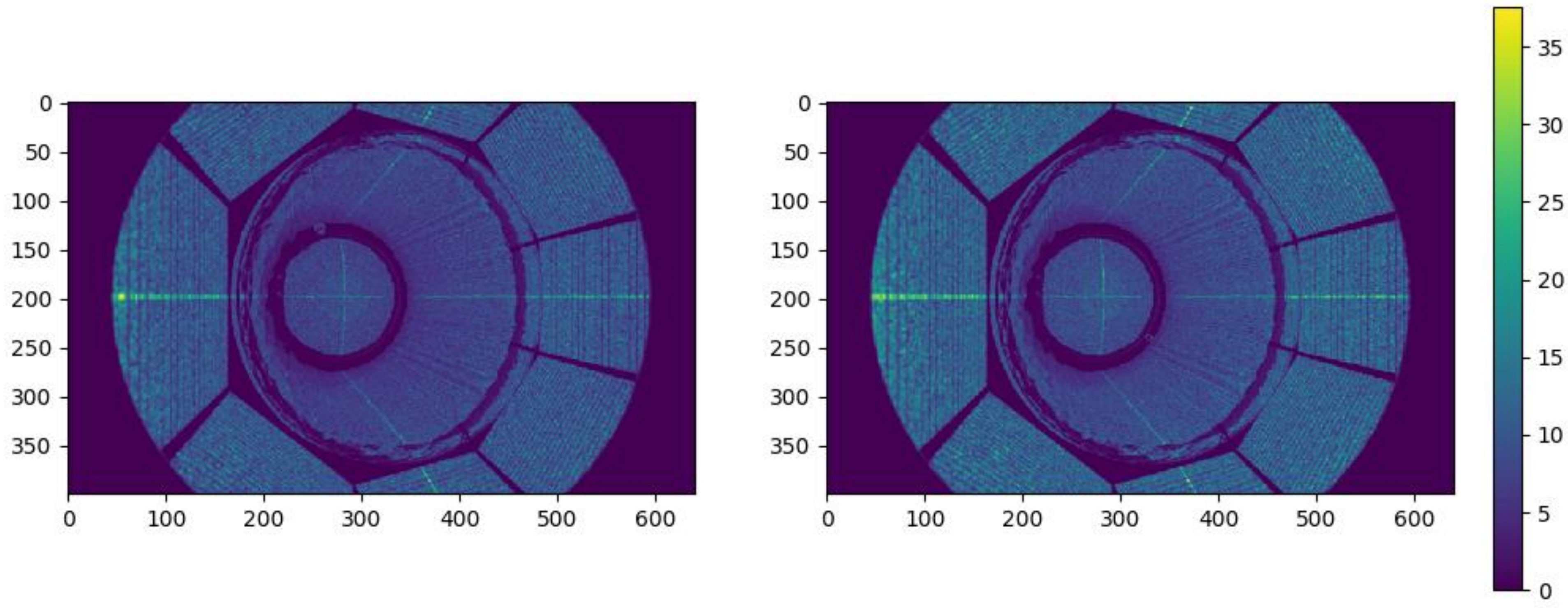


Brighter (darker actually) because I forgot to adjust optical parameter…

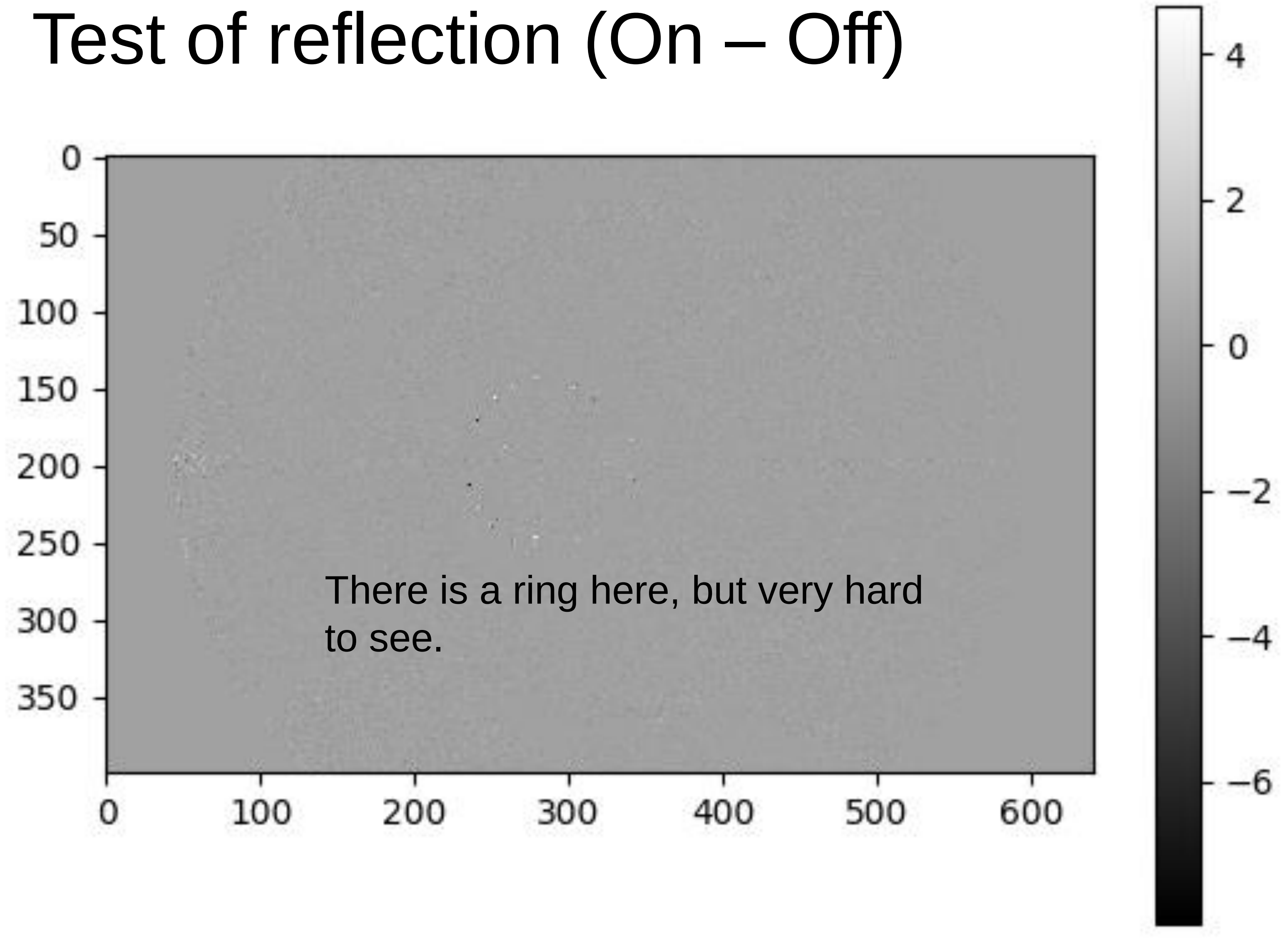
Bubble at center (0, 0, z)



Smaller bubble & reverse (x, y)



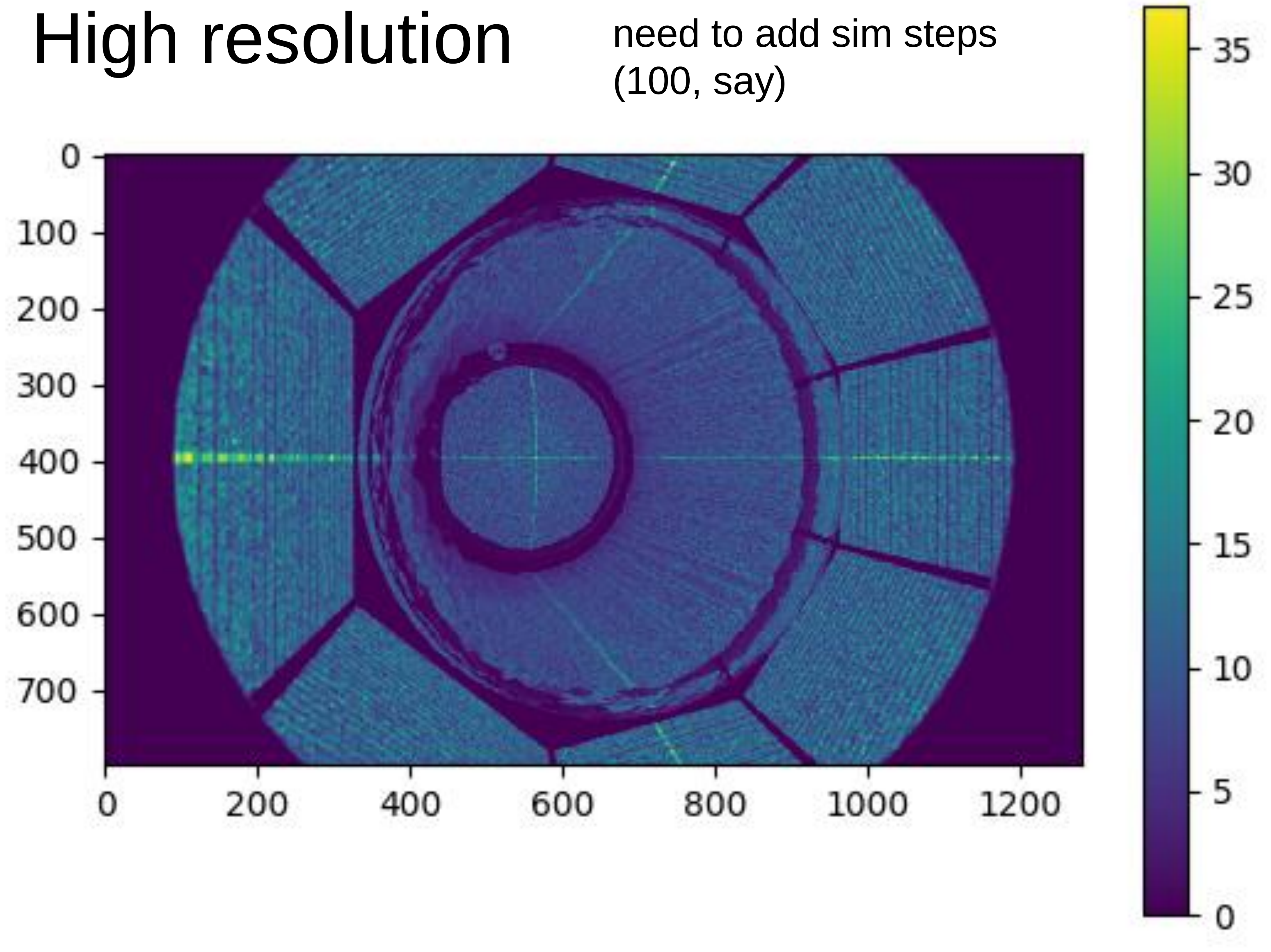
Test of reflection (On – Off)



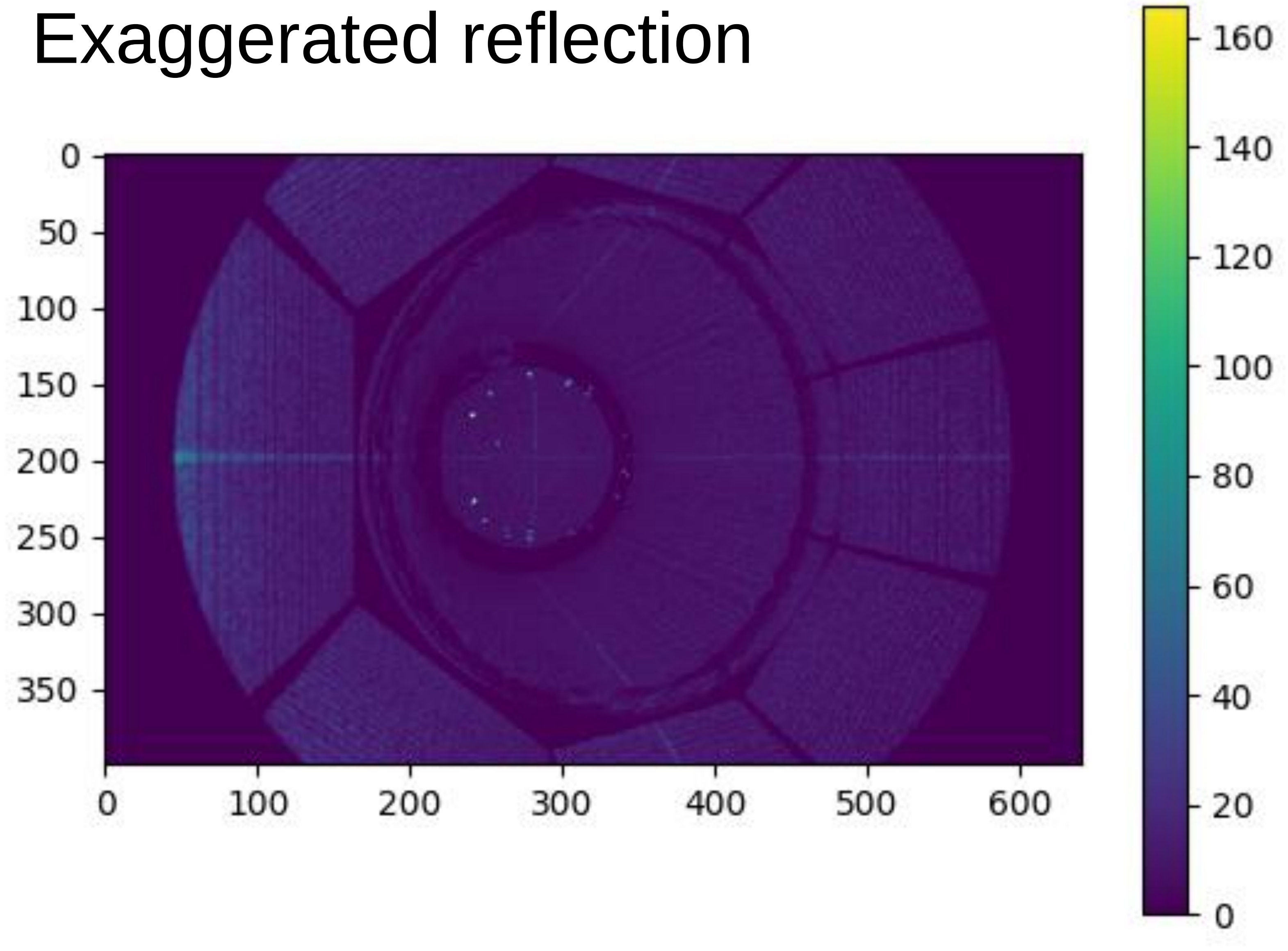
There is a ring here, but very hard to see.

High resolution

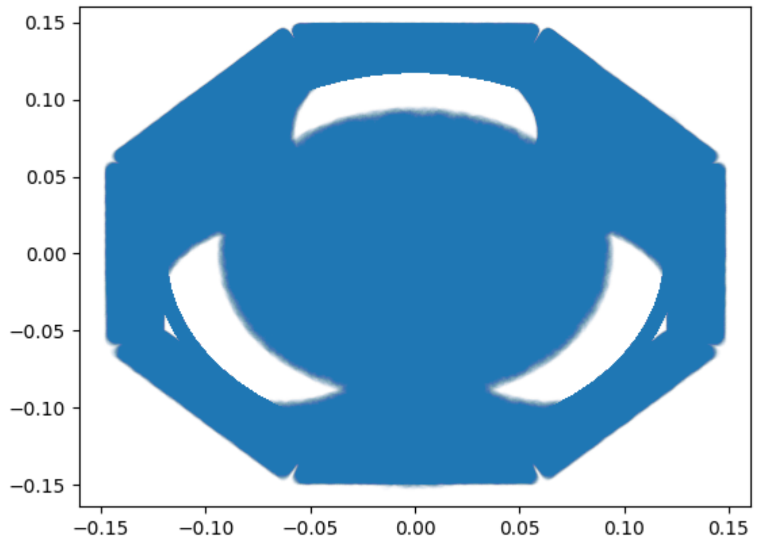
Wavy jar boundary due to small bubbles: need to add sim steps (100, say)

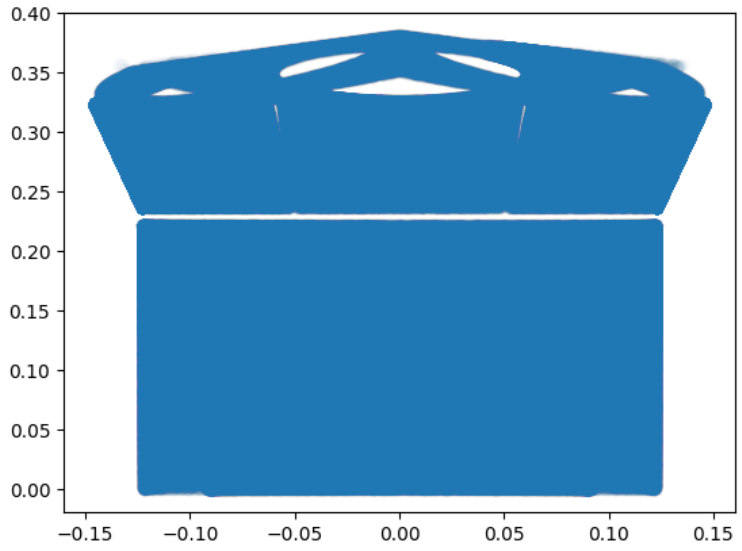


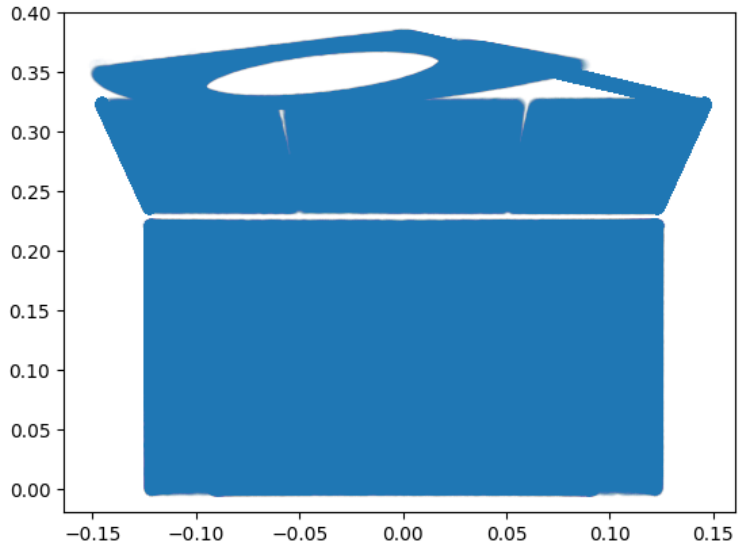
Exaggerated reflection



Scatters of sim





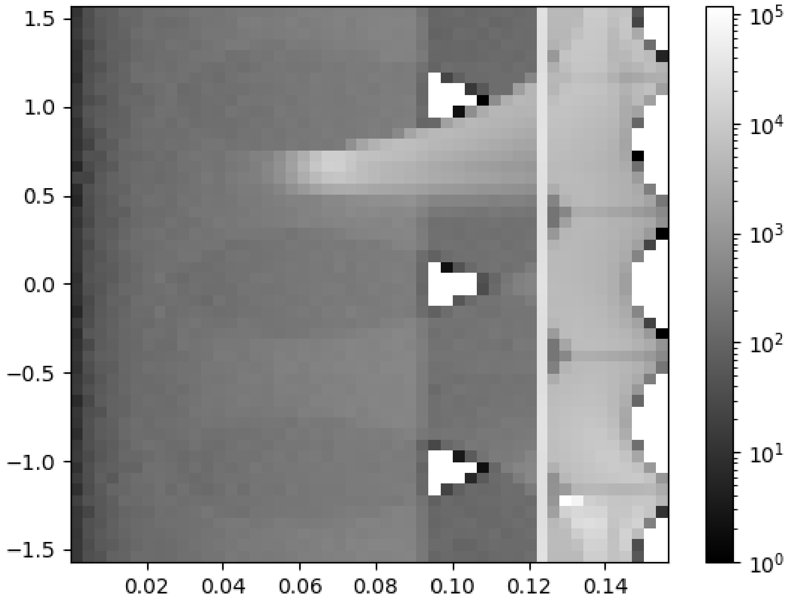


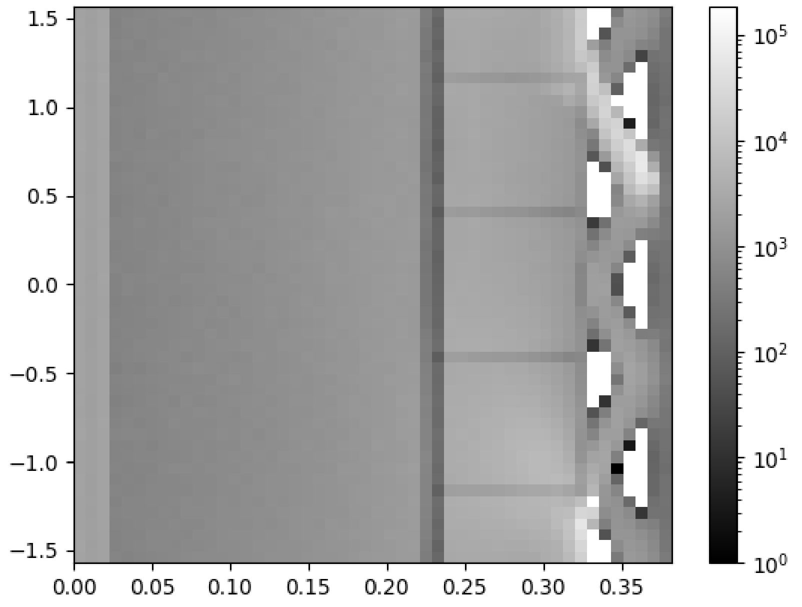
Histogram (cylindrical coordinates)

Similar pictures can be made by plotting the results of sim_old.py

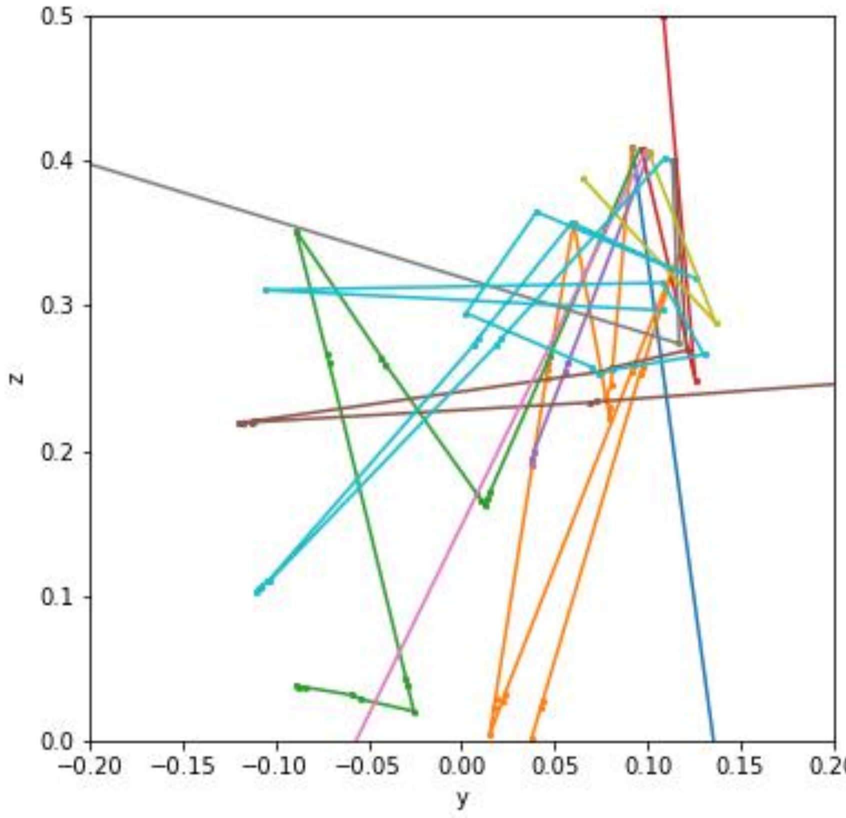
R-φ

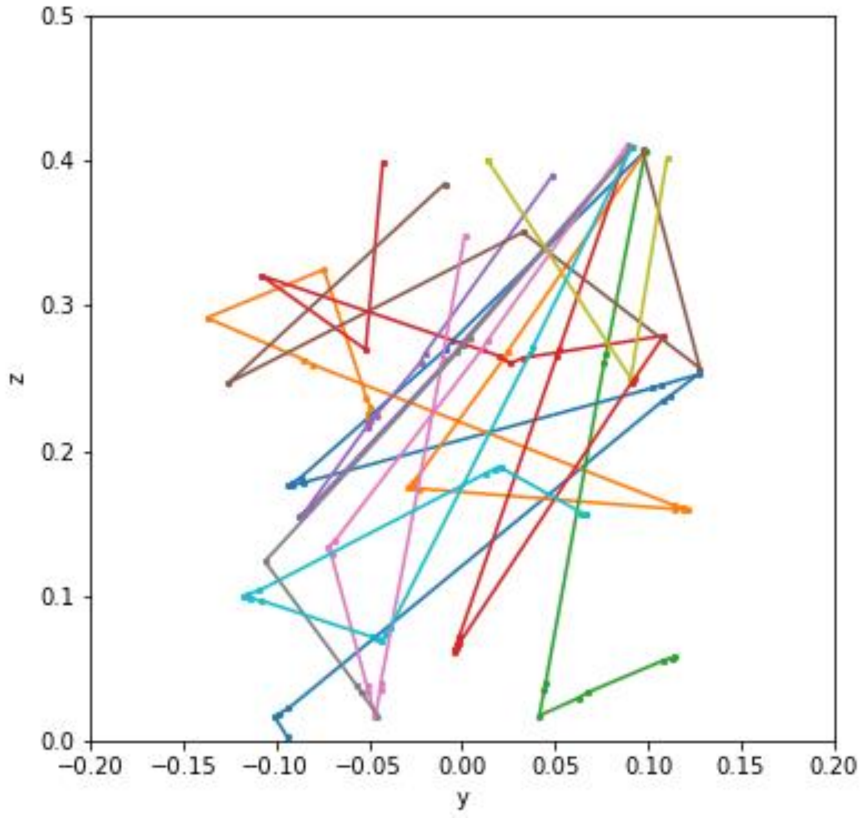
Z- φ



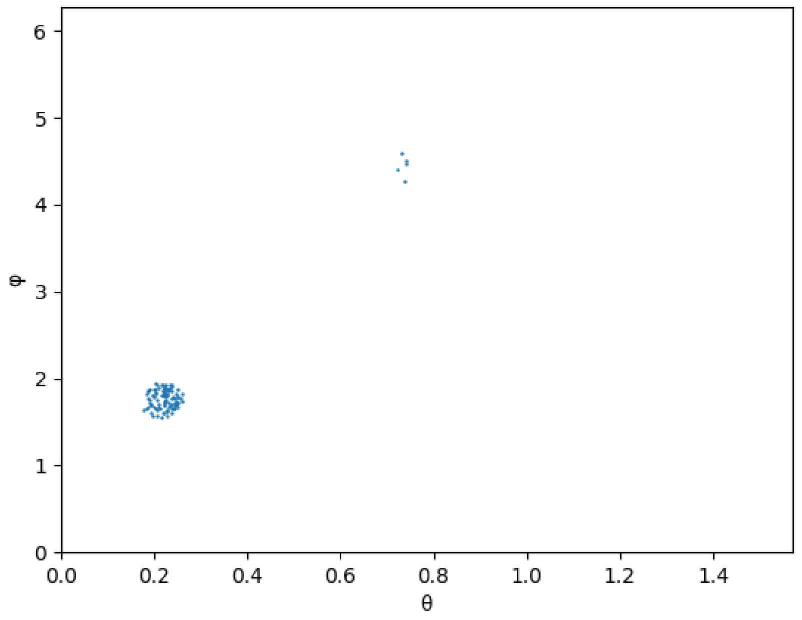


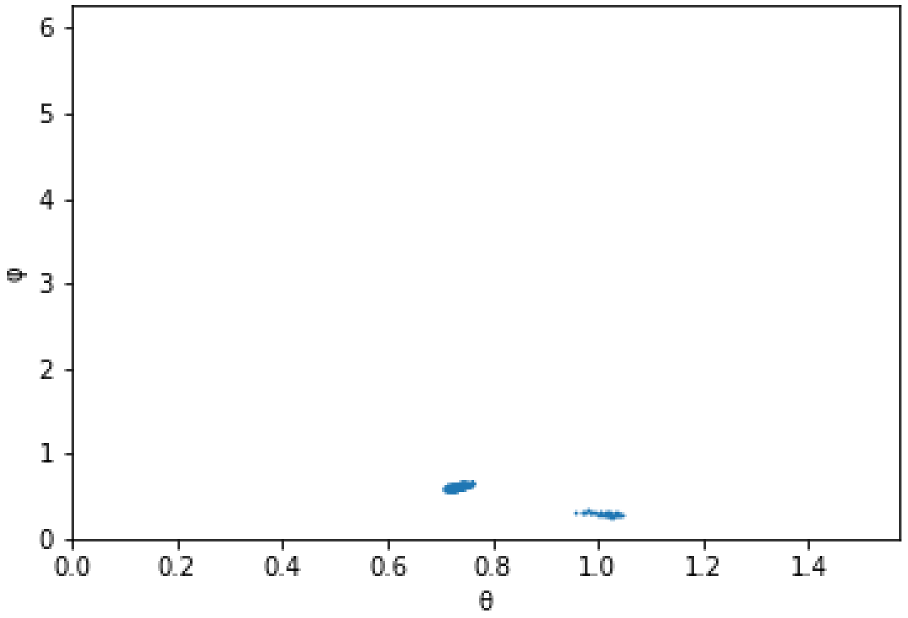
Photon tracks (LED ring)





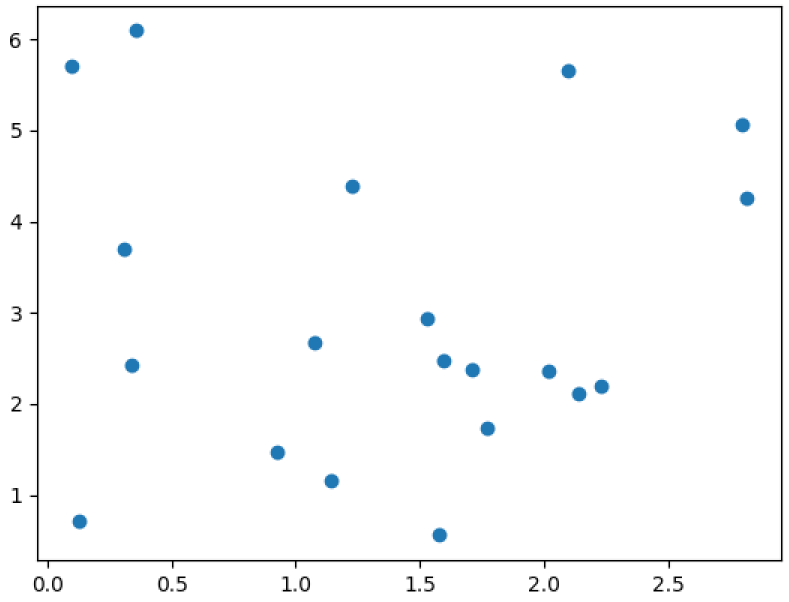
Photons' angular distribution at a piece of reflector

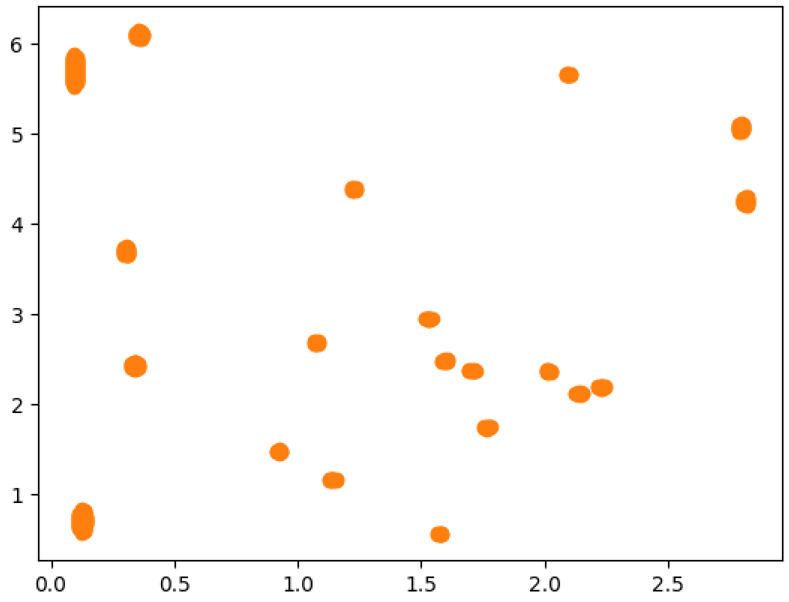




For a specific piece of reflector, the incident photons are clustered in angles

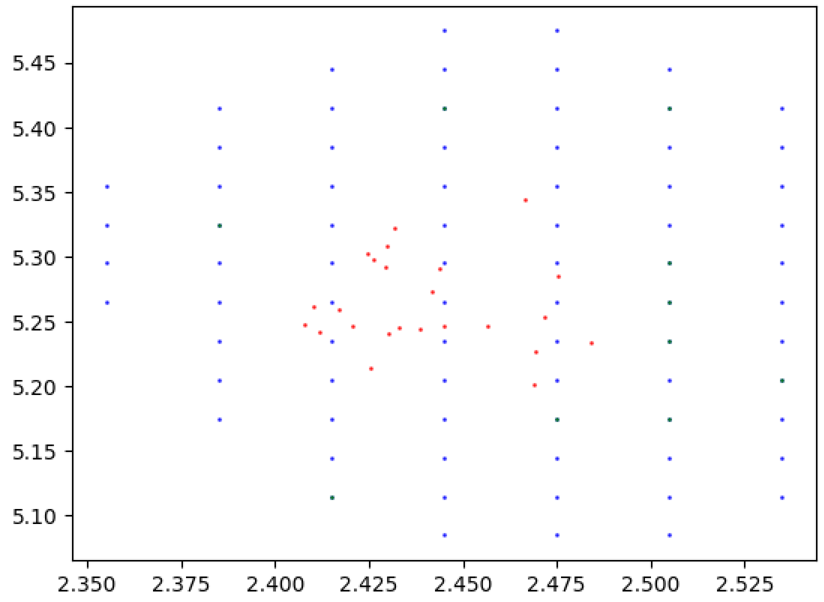
Initial trial for photon-tracing

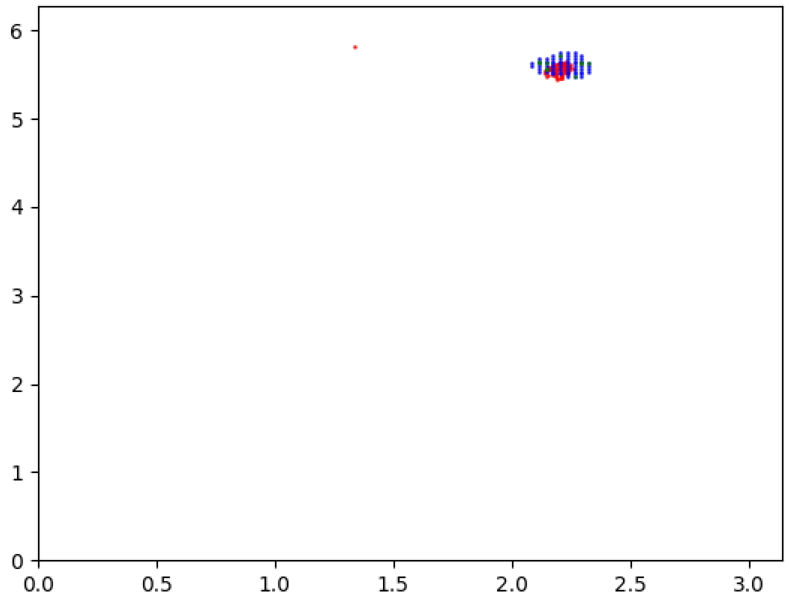




Test of math. Random angles (left) and generated ones (right)

Initial trial for photon-tracing





Actual effect for a random piece of reflector. Blue (generated)

Discussion

- In step 1 simulation (LED to reflector), only 0.6% of the space chunks are used (non-zero).
- Strips? Cartesian spacing!

- In step 2 simulation (pixel to geometry), about 64% of photons reach bright chunks – that would contribute to the image
- More photons per pixel reduces fuzziness (statistical fluctuation)

- Tried adding reflection to illumination map;
- A more detailed lens optics could be applied in math photons

Tips for simulation

- Before learning to run a sim in chroma, it is useful to read the basic definition of geometry Classes in **detector.py** and **geometry.py**. Photons defined in **event.py**. Sample materials defined in **/demo/optics.py**
- The sampling of LED light source follows the data on the product sheet (VSMY98545). The random sphere and half sphere follows the math here.
- I used pymesh to generate mesh. There are also some built-in mesh generation functions in chroma, but they have opposite normal vector with pymesh. Check **stl.py** for mesh loading.
- Since the sim results are Events, it is also very convenient to use ROOT to view them.

- Be careful about memory. Python gets really slow if you run out of memory. Check your machine's data and use pdb to monitor the status in simulation if necessary.
- Do not overcome 10^6 photons in a batch of sim. However, dealing with more photons a time do help improve efficiency: Numpy is not linear in time complexity.
- Chroma uses ndarray as output, so it is important to think about Numpy realization for any math. A useful trick is to write a simple math function for a single line of the array, then use *np.apply_along_axis()*
- It is normal to deal with large amounts of ndarrays, the fastest way (tested with a couple lines) for storing the results is putting them in a list[] then use *np.concatenate()*

Explanation of different geometries

- Intended to add SiPMs initially – the realization I could thought of was to add a piece of black paper. But actually, you will need to cut holes on reflectors and that could be laborious.
- The *detector_construction()* was the first geometry, and it is written based on Solidworks model and Geant4 geometry. I recommend using this one as prototype for any changes.
- *detector_reflector()* is the one having all the reflectors detecting. This one is the currently used geometry. Also included my trial of construction code. Making use of it might help you adjust the geometry conveniently.
- *detector_camera()* is the geometry intended for the aborted simulation in camera_as_source.py.

About run.sh

- A simple batch used to run a whole series of simulation

- Should check optics.py before running. Contains modification of the parameters using *sed*
- Used *nohup* to run the second step, it allows the detach from terminal: the second step cost hours

- A useful numpy trick reading: https://nbviewer.org/gist/rossant/4645217
- The blog of the maintainer: https://ben.land/post/2020/12/07/optical-physics-chroma/

- Contact concerning any question within this repository:

wzha38@163.com

Thank you!

Apr / 2023