2D Image Generation Script Documentation

Andre Adam

Last Updated: August 6, 2024

This document will provide insight into how the code was written, how it works, and how to use the script for generating batches (training data) of 2D images. Have in mind this simple compilation of algorithms was created with the goal of generating datasets for machine learning. The code is also shared in open-access format, that means anyone is welcome to either suggest modifications or directly modify the code presented.

Also, instead of having a standalone method for reading and writing images, we use stb image`https://github.com/nothings/stb`. While we provide some stb files here, check their GitHub for more up to date versions.

Requirements

To run the code, the following files are necessary:

```
main.cpp
helper.h
stb_image.h
stb_image_write.h
```

At the moment, there are no input files, so the parameters are set within the main file.

Compiling

At the moment, it is believed any c++ compiler will work. However, we strongly suggest a gcc based compiler, because that's where most of the testing was done. If there are issues with any other compiler, please let us know in the GitHub directly.

To compile the code, simply put all the required files in the same folder and run the following:

```
g++ main.cpp -o main.exe
```

That should create the executable file main.exe.

Setup Parameters

The first two options the user has to set are the *batchFlag* and the *pathFlag*. The batchFlag controls whether we are running a batch of images or generating a singular image. The pathFlag determines whether we will only generate images in which there is at least one path in the fluid connecting the left and right boundaries. Both are to be set as true or false.

The following parameter is the *genMethod*. Currently, there are 3 options for this parameter. Setting it to '0' means we will generate Voronoi Cells, setting it to '1' will generate Quartet Structure Generation Set (QSGS) images, and finally '2' will generate images using a random overlapping circle packing, which will also be reffered to as random sphere packing method (RSPM). Any other number entered for genMethod will return an error.

The final parameter available at this stage is the *offset*. This is for generating batches of images, it will always save images by their number with leading zeros, up to 99999 (ninety nine thousand, nine hundred and ninety nine). If you generate **n** images, and want to generate another **m** images, then you would set the offset to **n** so that the new images will be saved up until $\mathbf{m} + \mathbf{n}$ without overwriting any of the existing images.

Image Generation Methods

Voronoi Diagrams

There are a multitude of ways of generating Voronoi Cells and they all arrive at similar results. For more information on how they are defined, check the Wikipedia page`https://en.wikipedia.org/wiki/Voronoi_diagram`.

The chosen method is the Jump-Flooding algorithm (JFA) `https://en.wikipedia.org/wiki/Jump_flooding_algorithm`, which is actually an approximation to a Voronoi diagram, but is often chose because of the fixed computational time. Below is a sample pseudo-code of the JFA Voronoi generation for a $N \times N$ grid.

Algorithm 1 JFA for Voronoi in 2D

```
for $k \in \{\frac{N}{2}, \frac{N}{4}, \cdots, 1\}$ do
    for Every pixel, $p$, in grid $(x, y)$ do
        for Each neighbor, $q$, in $(x+i, y+j)$ where $i, j \in \{-k, 0, k\}$ do
            if $p$ is undefined and $q$ is defined then $p = q$
            else if $p$,$q$ are defined but different and $\text{dist}(p, s) \geq \text{dist}(p, s')$ then $p = q$
            end if
        end for
    end for
end for
```

So then what are the user defined parameters that define a Voronoi Diagram? For a single image generation, we will pick fixed values for the parameters, while in batch mode we will have to define ranges of values from which we will randomly draw parameters.

The image width and height must be defined, and they are in units of pixels. Then, we must define the number of seeds, the channel radii, and finally, we must define how we calculate the distance. The current code contains 2 methods for calculating the distance: Euclidean distance1 and Manhattan distance2.

$$d_{Euc} = \sqrt{(i_2 - i_1)^2 + (j_2 - j_1)^2} \tag{1}$$

$$d_{Mht} = |i_2 - i_1| + |j_2 - j_1| \tag{2}$$

For generating a single image, setting the variable "opts.DistCalc" to 0 corresponds to the Euclidean distance, and setting it to 1 corresponds to the Manhattan distance. In batch mode, there is an additional option; setting it to 3 will randomly draw it using either Manhattan or Euclidean distance.

For batch mode, we must present code with a minimum and maximum number of seeds, and a minimum and maximum channel radii. In discretized space, it only makes sense for these numbers to be integers.

QSGS

The QSGS method is best described in literature [1]. Basically, in this method we set a probability of any given voxel becoming a seed. We call this number C_d, the probability of any given cell becoming solid. Of course we could just randomly draw **n** seeds, but we use the aforementioned approach because it is more common in literature related to QSGS.

Then, in each subsequent iteration, these solid cells have a chance of expanding and making neighboring cells solid as well. These are two different probabilities, called D_{diag} and D_{sides}. In other words, if a voxel is diagonally connected to a solid pixel, it has a given probability to become a solid every iteration, but a different probability if instead it is directly adjacent to a solid pixel. For a single image generation, we directly set D_{diag} and D_{sides}. However, for batch mode, we set a maximum and minimum D_{diag} as well as a maximum and minimum ratio between D_{diag} and D_{sides}. D_{sides} is then defined by D_{diag} and the ratio, R.

The iterative process will repeat itself until a target solid volume fraction (SVF) is achieved. The SVF is directly set for single image generation, but once again we must define a maximum and minimum for the batch generation. Also, this algorithm will usually go slightly higher than the target SVF by nature. High values of D_{diag} and D_{sides} will cause more overshoot, while smaller numbers will get increasingly closer to the target SVF. The trade-off is that if we make them too small, it will take too many iterations to reach the target SVF and therefore the computational time of generating batches of images will also go up. The target SVF is also often reffered to as P_s in literature.

RSPM

This method, while pretty self-explanatory, has some intricacies that make it more interesting. The first parameter we must choose is a target SVF. In the past, we have tried to use this method with a set number of circles to draw, but that has not worked well, as it then becomes and empirical problem of trial and error to determine the range of number of circles to draw that will yield, on average, a given SVF.

Therefore, the current approach will set a target SVF and iterate until we go equal or above the target SVF. Then the circles drawn are of a fixed radius, which is also set by the user. In batch mode, we have to set a maximum and a minimum values for circle radius.

Pathfinding Algorithm

For this algorithm, we just want to know if there exists a continuous path from the left boundary to the right boundary via the fluid phase (black or 0). We apply a very simple flood fill algorithm to this task: all fluid pixels on the left boundary are assigned a flag, and then each neighboring fluid voxel gets the same flag iteratively. If this flag gets to the right boundary, then we know there is at least one continuous path via the fluid phase.

References

[1] Moran Wang et al. "Mesoscopic predictions of the effective thermal conductivity for microscale random porous media". In: *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics* 75.3 (2007). ISSN: 15393755. DOI: 10.1103/PhysRevE.75.036702. URL: https://escholarship.org/uc/item/7891j8jt.