Parallel Optimised View Synthesis

Aayush Kumar, Prateek Sogra, Jahnavi Kairamkonda, Keshav Birdi, Bhuvan Marri

CS677A Course Project

November 27, 2023

1 Motivation

1.1 Background

In the context of 3D data visualization, a deliberate selection of viewpoints is important to obtain a set of meaningful images from these viewpoints. While random selection methods offer a baseline exploration of the data set, this project proposes a more sophisticated approach. We initiate the process with randomly generated viewpoints scattered across the data set. Subsequently, we employ an optimization method to refine these viewpoints, seeking local optima for image generation. One of the methods to find non-defocussed images is the Pyramid-based method [4]. It chooses the images based on Laplacian and Gaussian calculations, however, the most focused image might not provide the most information. Entropy is a good measure to determine the amount of information contained in an image. To use optimisation methods based on finding maximum entropy images, we went through [3]. This paper proposed an algorithm based on gradients which might not be feasible as it is difficult to find a function that calculates the entropy of an image of any dataset and is differentiable. [2], on the other hand, works with Shannon entropy measurement and probability distribution functions, which is a promising technique. While this paper used a gradient-based method, it can be adapted for our needs. Till now, entropy-based optimisations that do not use gradient-based methods have not been sufficiently explored as of yet.
These methods have also ignored implementation in terms of parallelism. Large datasets often require a significant amount of time for image rendering, and parallelising these processes may often be necessary for real-time usage of the generated images [1]. In this project, we implement a parallel approach to local optimisation of images of a dataset.

2 Problem Statement

Our task in this project is to locally optimise the viewpoints of a 3D dataset so that they give the most informative 2D image of the dataset, that is, they reveal the most areas of interest of the image. To measure how informative an image is, we use entropy. The higher the entropy, the higher the information given in the image. We perform this optimisation over a given collection of viewpoints, and to improve efficiency we parallelise all these optimisations. Hence, our problem statement becomes:

How can we efficiently parallelise the synthesis of a set of locally maximal entropy images from a 3D dataset given a collection of initial viewpoints?

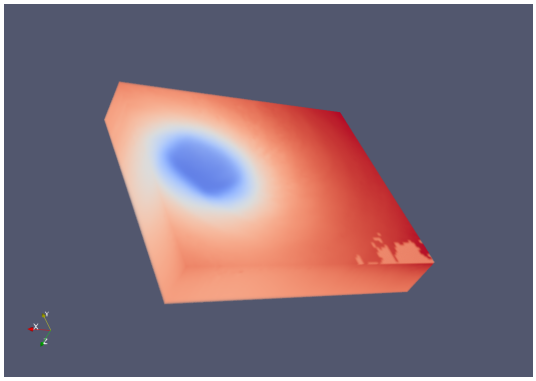
3 Methodology

3.1 Datasets

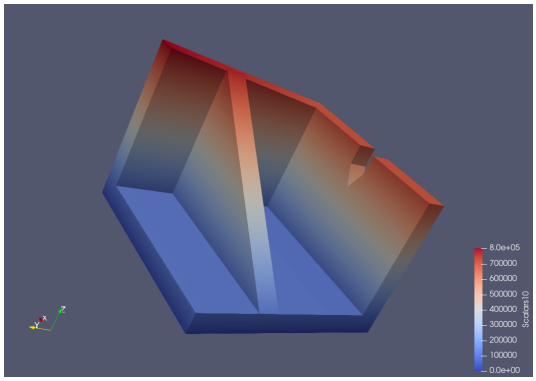
We run our final parallel optimisation pipeline on two separate datasets - one rendered using surface rendering and the other using volume rendering.

3.1.1 Geometric Figure - Surface Rendered

We created this dataset (Figure 1b) from scratch on Python using the PyVTK library. It has geometric figures that provide for interesting changes of images based on viewpoint. For example, there is a small notch on one side. When we view through this region, we can get information about the other side of the dataset. Similarly, the triangular figure at the center can occlude some part of the dataset from some angles but not from others. It also has a consistent gradient, which makes it an ideal candidate for image optimisation.



(a) Pressure simulation data



(b) Geometric shape made using pyvtk

Figure 1: Datasets

The dataset was made using an unstructured grid of 100 × 100 × 80 points. It was given scalar values, and the topology was introduced through the definition of hexahedra, triangles, and quadrilaterals. The finalized dataset was then exported to a VTK file. We used surface rendering of this dataset for our optimisation pipeline.

3.1.2 Pressure Simulation - Volume Rendered

This dataset was provided to us by Prof. Soumya Dutta for our project. It is a large dataset with densely packed values that simulates a pressure system. Unfortunately, since we use only one KD lab system for our project (see 4), we were unable to repeatedly perform volume rendering on this dataset and produce images for optimisation. Thus, we cut parts of the dataset so that its final grid contained 275 × 225 × 50 points.

3.2 Optimisation Pipeline

We used Python for all of our code. Python is required for the implementation of Paraview routines, and it also provided us the flexibility of using optimisation libraries as well as MPI.

3.2.1 Image Generation using Paraview

There is no direct way to generate a continuous function that converts input viewpoints and dataset to the entropy of an image of this dataset taken from these angles. Hence, to derive this entropy, we first render an image from these viewpoints and find the entropy value associated with it.
Rendering an image from viewpoints is a non-trivial process, that usually requires the usage of a visualisation client. We are required to automate this process. For this, we used ParaView. ParaView is a visualisation client that can generate a Python script for any processes performed using the client. We thus used ParaView to create a script that loads the required dataset and renders (and saves) an image using it. We then modified this script to change camera parameters based on the input viewpoints. In this manner, we created a function that saves an image of the dataset based on the given inputs.

3.2.2 Objective Function

Our objective function takes two inputs: viewpoints that indicate the elevation and azimuth values of the angle of the camera. This function first calls our image generation routine (described in the previous section) to save an image corresponding to the given inputs. Following this, we load the image and calculate the Shannon entropy for this image using the *scipy.stats.entropy* function of the Scipy module for Python.
We finally return the entropy value, multiplied by −1. This is done as our optimisation routine is used to minimise functions, and minimising this gives us the maximum entropy.

3.2.3 Optimisation Routine

We use the Nelder-Mead algorithm, a zero-order method, for our optimisation. This algorithm is again implemented using the Scipy module (*scipy.optimise.minimise*). Since we perform local optimisation, we also supply bounds to the optimiser that restrict the domain to ±20° of the given input angles.

3.3 Parallelisation Approach

3.3.1 Initial Approach

We decided to use a combined parallelisation approach that utilises both multithreading and multiprocessing, where each thread of each process optimises for a single input (and thus generates a single image) at a time. This was inspired by the approach in [1]. Multithreading is a viable approach for this procedure since each thread can operate independently; they just have to visualise the dataset and do not need to edit it. Hence, there will be no data races. This lets us exploit shared memory and reduces the number of copies of the dataset we need to store. Multiprocessing is implemented using mpi4py (the python interface for MPI) while multithreading is implemented using the Threading module in Python.
We have one manager process that distributes viewpoints to all other processes, and one manager thread per process that distributes viewpoints to all other threads of that process. The manager process initially distributes viewpoints to all other processes; the manager thread receives these viewpoints and distributes one to each of the other threads which independently run the optimisation routine.
Each process maintains a queue (implemented using the Queue module) that all threads can update. Once a thread is finished optimising for a given viewpoint, it adds its own thread name to the queue and then dies. The manager thread runs an infinite loop that waits until there is something to read in the queue. As soon as a thread puts an entry in the queue, the manager thread reads (and in doing so, removes) this entry. It then requests the manager process for a new viewpoint, by sending its own rank (or process ID). As soon as it receives a new viewpoint, it starts a new thread with the same name that now performs the optimisation for the new viewpoint. This continues infinitely.
The manager process in turn waits for other processes to communicate with it. Once it receives the process ID of any process, it sends a new viewpoint to this process ID. It continues doing so until the list of viewpoints has been exhausted. Finally, it sends a message to all other processes that tells them to clean up all their threads and die.

3.3.2 Revised Approach (ParaView-Specific)

Unfortunately, our proposed mechanism could not be implemented with ParaView. This is as ParaView is by itself multithreaded, and it uses all the cores present by default. We even observed that if one thread performs image rendering, and that thread is killed, ParaView will not support a new thread that would subsequently run image rendering processes.
Thus, we had to resort to a multiprocessing approach, retaining the basic dynamic nature of our initial approach. We continued using a single manager process and all other rendering processes, where the manager process supplies inputs viewpoint-by-viewpoint. The major change occurs within the rendering processes - they do not pass the viewpoints to a thread; they instead run an infinite loop within which they perform a single optimisation, request and receive the next viewpoint, and then continue to the next iteration. These processes stop when they receive a specific message from the manager process that indicates all images have been generated.

3.4 Experiments

Our experiments were conducted on the KD lab systems. Currently, we have only been able to test our methodology on a single system (see 4).
For the geometric figure (surface rendered) dataset, we ran tests with a varying number of both processes and total input viewpoints. We were restricted to using a maximum of 8 rendering processes as more processes on a single system would not run parallelly.
The pressure simulation (volume rendered) dataset is much denser and larger, and loading it into memory every time a new image generation task is conducted adds further restrictions to the number of input viewpoints we test for. We were thus restricted to using 10 viewpoints.

4 Bottlenecks

Our project has some constraints that cannot be overcome given our allocated time and resources. These include:

- **Optimisation methods:** The entropy of a generated image is not necessarily a differentiable function of the viewpoints, and even if it is, this function is difficult to determine. Thus, to find local maximas of the entropy of images, we cannot apply any optimisation methods that use gradients. This restricts us to applying zero-order methods.
- **Dataset limitations:** It is evident that the entropy of an image of a dataset containing scattered and randomly distributed values or largely similar values is unlikely to change significantly based on viewpoint. Zero-order optimisation methods especially struggle for such datasets where the direction of change in entropy is not clear. Hence, our optimisation method is only practical for those datasets in which there are clear gradients or patterns that can be highlighted by changes in viewpoint.

- **Optimisation time:** It is not possible to determine beforehand the path that the optimisation algorithm will take to reach the local maxima for any given input viewpoints. The number of steps taken to converge to the maxima might vary significantly for initial angles that are very close to each other. Thus, we cannot determine the time taken for optimisation of a particular viewpoint, and hence cannot balance the load on different processors beforehand through an optimal distribution of viewpoints.
- **Distributed Computing Limitations:** Our code uses ParaView to render images from viewpoints. This requires the ParaView package for Python to be installed on the system running the code. We were able to use ParaView for one system on our experimental setup by setting up a conda virtual environment. However, conda virtual environments cannot be shared across different systems, and we could not install the ParaView package on the native system Python (this requires `sudo` access to build ParaView from source which was not available to us).

5 Challenges

We faced challenges in both the optimisation and parallelisation aspects of our project. Some of these are outlined below:

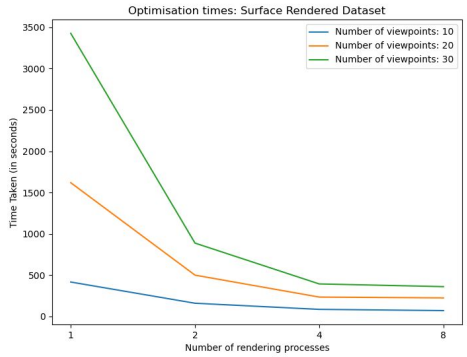
- **Entropy generation:** To create entropy values from a viewpoint, we first had to understand how to convert viewpoints to images. This is not a trivial task to automate in code. We sought the advice of Prof. Soumya Dutta who helped us in obtaining code for image generation using ParaView.
- **Load balancing:** As highlighted in section 4, we could not statically allocate viewpoints to different processes to balance the load between them. To overcome this, we had to construct a dynamic allocation methodology that allowed us to supply viewpoints to processes as and when they became free to ensure load balancing.
- **Parallelisation with ParaView:** Importing the paraview package for Python and running many MPI processes with a python interpreter caused issues - firstly, the processing was quite slow, and secondly, we occasionally encountered errors related to WebGL rendering (truncation of images, etc). Initially, we felt this might be due to rendering being on-screen, but this wasn't the cause as the parraview installed by conda did not render on-screen. Upon research, we found out that neither python nor pvpython (which is the standard python interpreter of paraview) is MPI parallel - we had to infact use pvbatch as the interpreter for MPI commands if we wanted paraview to run parallely. While this resolved the errors and improved processing time significantly, one warning persists - ParaView has to sometimes reset the view-up vector of the camera for some specific viewpoints during optimisation.

6 Results

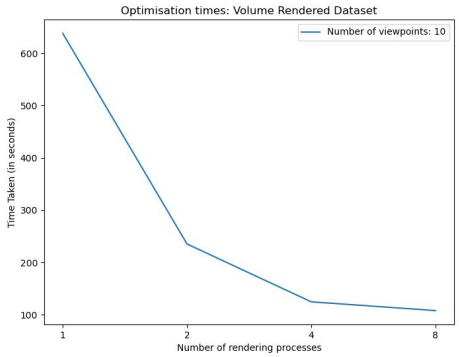
Our results revealed that parallelisation has a large speedup upto a certain number of processors. This number of processors increases as the number of viewpoints increases. This is true for both surface rendered and volume rendered datasets.
All our readings were for one trial. We observed that there is not a significant change in total processing time between different trials, both in the case of parallel as well as serial execution (this can be observed in Figure 4).
Our experiments on both datasets revealed the typical processing time curve with an increasing number of processes. There is a drastic drop in the processing time for a low number of processors (Figure 2) with a saturation point being reached shortly after. What is interesting to note is that the parallel efficiency (speedup divided by the number of processors) is often more than 1, which is the theoretical limit (Figure 3). This stems from the fact that reloading the dataset on the same memory space becomes quite a tedious task as we increase the number of input viewpoints. Loading a dataset for the $(n+1)^{th}$ viewpoint takes more time than the same for the n^{th} viewpoint. Splitting across double the number of processors means a lesser number of times the dataset must be loaded per processor, which can save more than double the time as loading times are now reduced. This impact is so significant that for a higher number of viewpoints we can actually see that the parallel efficiency increases to an extent before falling due to saturation.

7 Conclusion

In this project, we implemented a method for parallelised synthesis of images with locally maximal entropy from a 3D dataset. We used the ParaView and Scipy Python packages to create an optimisation routine based on the entropy of generated images, and ran this routine on a parallel approach that used multiprocessing through MPI. We also created implementations for a mixed parallel approach that exploits both multiprocessing and multithreading, that can be used for optimisations that are not dependent on ParaView.
We further analysed the results on the speed-up provided by parallelisation and found them to vary from our our theoretical expectations, which spawned interesting domain-specific inferences on parallel efficiency.

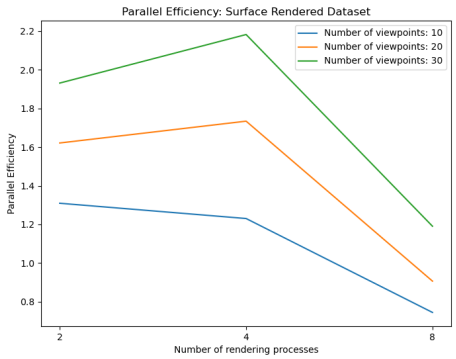


(a) Surface Rendered Dataset

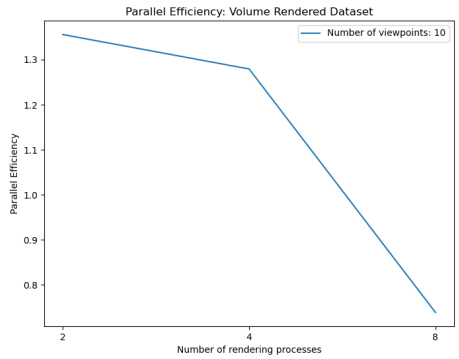


(b) Volume Rendered Dataset

Figure 2: Total optimisation times

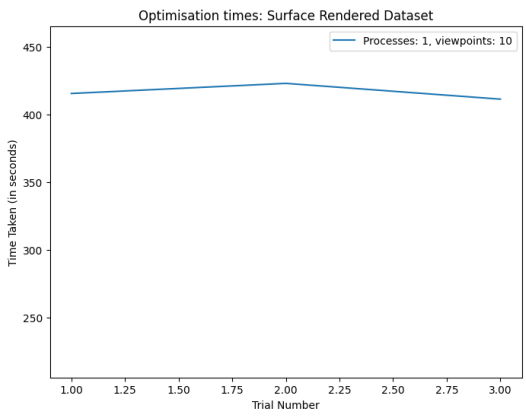


(a) Surface Rendered Dataset

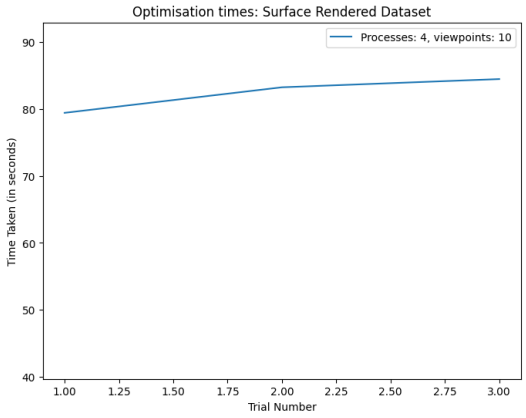


(b) Volume Rendered Dataset

Figure 3: Parallel Efficiency



(a) 4 rendering processes, 10 input viewpoints used



(b) One rendering process, 10 input viewpoints used

Figure 4: Variation in optimisation time (for Surface Rendered Dataset)

8 Future Work

Our work provides a baseline that can be expanded upon to fully realise the potential of parallel optimised view synthesis. Avenues of future work include:

- Testing our methodology over multiple systems.
- Fixing the caching error by developing a technique to share memory.
- Direct methods of entropy generation that bypass the need for rendering an image every time we need to find entropy. This would significantly reduce optimisation time and the dependency on ParaView.
- Testing our mixed approach with methods that do not depend on ParaView or other multithreaded operations.
- Testing performance of our approach with other zero-order optimisation methods.
- Exploring static allocation of viewpoints to processes in a way that ensures load balancing (that can reduce waiting times and communication overheads).

9 Team Member Contributions

- **Keshav Birdi** Geometric figure dataset, parallelisation, experiments
- **Aayush Kumar** Entropy generation, optimisation and parallelisation code, results visualization and analysis, report writing
- **Prateek Sogra** ParaView rendering, parallelisation
- **Jahnavi Kairamkonda** Threading implementation, MPI setup, results visualisation, literature review, presentation
- **Bhuvan Marri** Presentation, report

10 GitHub Link

All our code is available here.

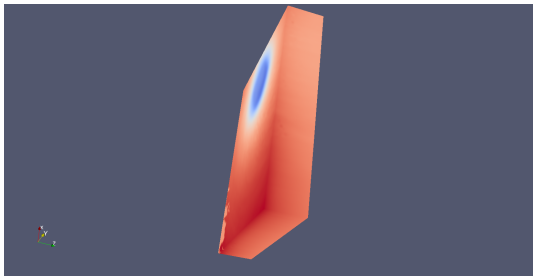
References

[1] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney. Grid -based parallel data streaming implemented for the gyrokinetic toroidal code. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, page 24, New York, NY, USA, 2003. Association for Computing Machinery.

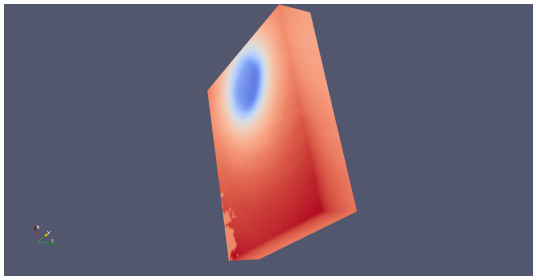
[2] David Ramirez, Javier Via, and Ignacio Santamaria. Entropy and kullback-leibler divergence estimation based on szego's theorem. *Journal Name or Conference Proceedings*, Volume Number or Conference Proceedings:Page Numbers, 2023. This work was supported by the Spanish Government, Ministerio de Ciencia e Innovacion (MICINN), under project MultiMIMO (TEC2007-68020-C04-02 and TEC2007-68020-C04-03), project COMONSENS (CSD2008-00010, CONSOLIDER-INGENIO 2010 Program), and FPU grant AP2006-2965.

[3] J. Skilling and R. K. Bryan. Maximum entropy image reconstruction. *Department of Applied Mathematics and Theoretical Physics, Silver Street, Cambridge CB3 9EW*, (Accepted).

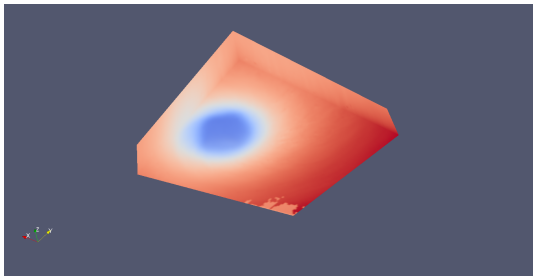
[4] K. Wohn. Pyramid-based depth from focus. 1988.



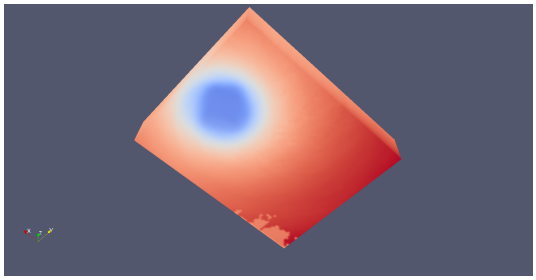
(a) Initial Viewpoint



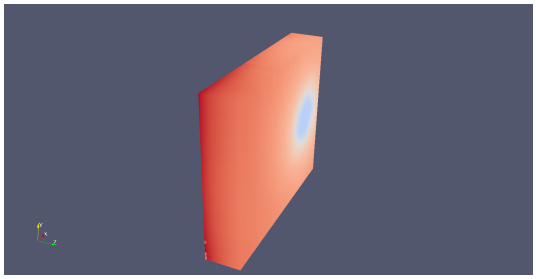
(b) Optimized Viewpoint



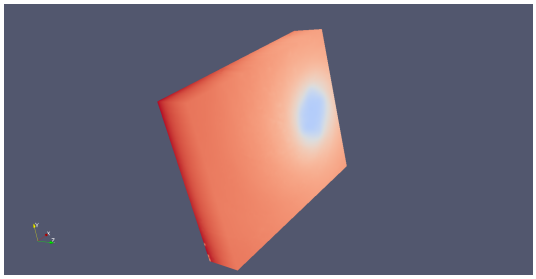
(a) Initial Viewpoint



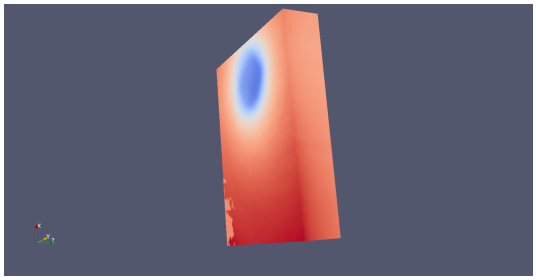
(b) Optimized Viewpoint



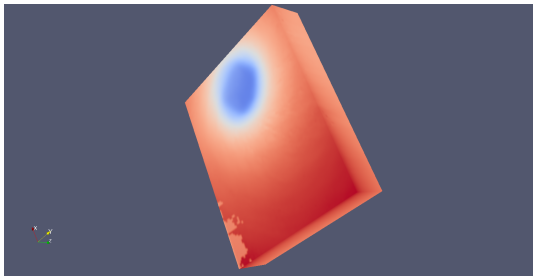
(a) Initial Viewpoint



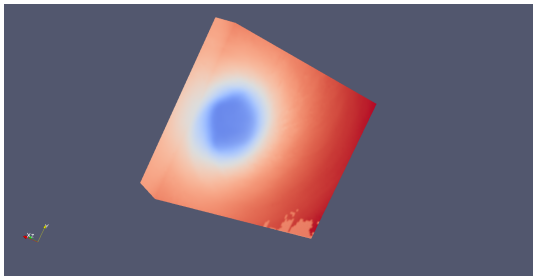
(b) Optimized Viewpoint



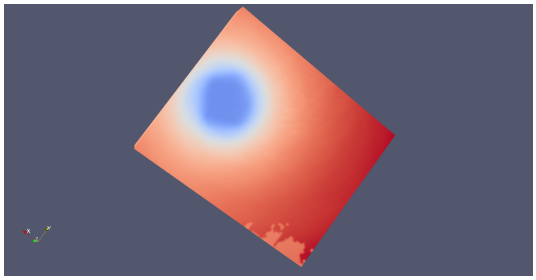
(a) Initial Viewpoint



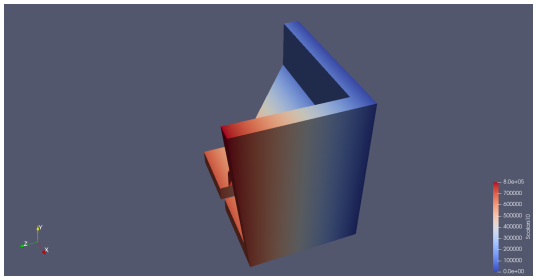
(b) Optimized Viewpoint



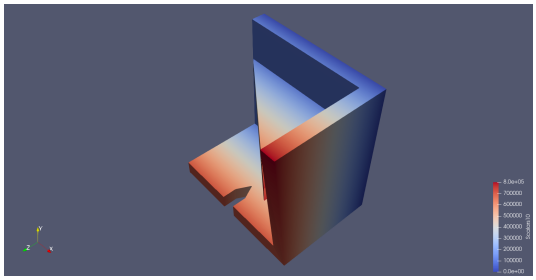
(a) Initial Viewpoint



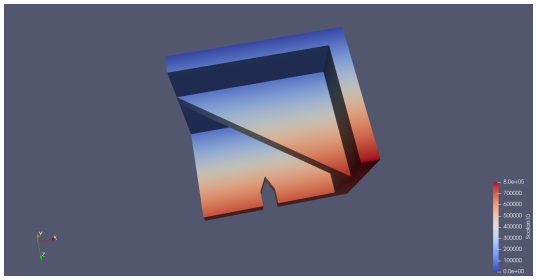
(b) Optimized Viewpoint



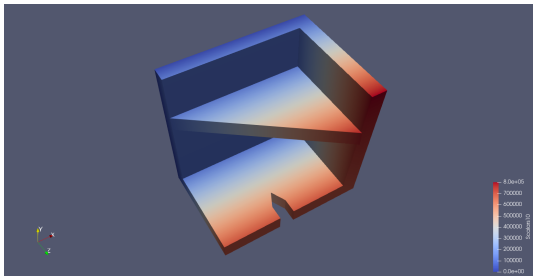
(a) Initial Viewpoint



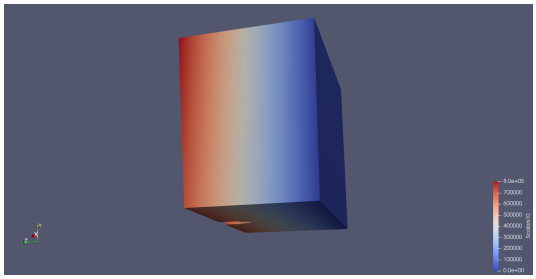
(b) Optimized Viewpoint



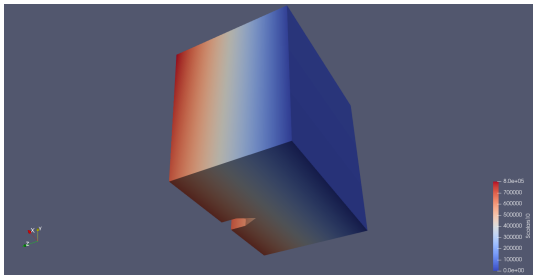
(a) Initial Viewpoint



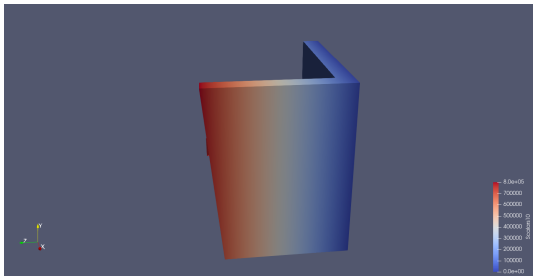
(b) Optimized Viewpoint



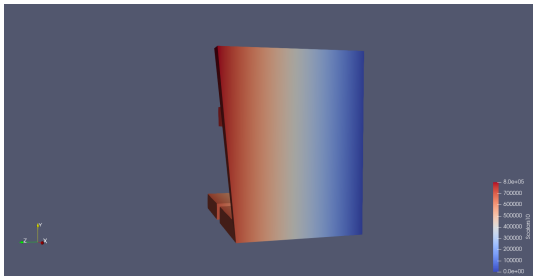
(a) Initial Viewpoint



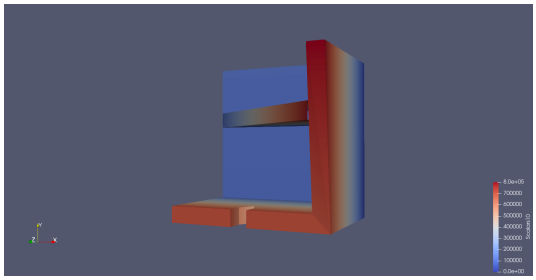
(b) Optimized Viewpoint



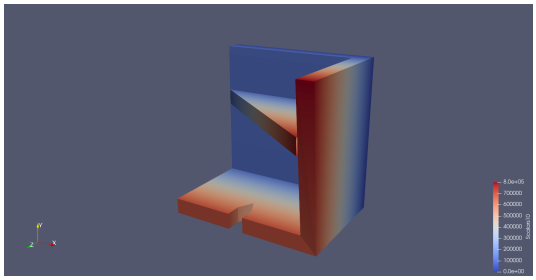
(a) Initial Viewpoint



(b) Optimized Viewpoint



(a) Initial Viewpoint



(b) Optimized Viewpoint