Work on an automated segmentation pipeline

William Stevens

28th October 2024

Work on an automated segmentation pipeline

This report describes progress on an automated segmentation pipeline described previously [1, 2]. The project uses basic image processing operations and algorithms based on flood fill to find distinct surfaces and surface patches in CT scans of the Herculaneum Papyri. The pipeline consists of distinct programs for initial processing of $512 \times 512 \times 512$ voxel cubic volumes, identification of non-intersecting surfaces and surfaces patches, joining together neighbouring patches without overlap, and rendering to see the results. Several utility programs and some third party tools are used for debugging the pipeline. Progress since last month: the efficiency of the DAFF algorithm has been improved; the program for stitching together surface patches has been rewritten with improved efficiency; information from fibre detection work can be used in the pipeline; metrics for comparing different runs have been developed; an interpolator for filling in holes in surfaces produced by the pipeline has been written; the pipeline has been run on 20 cubic volumes from the 2023 prize banner area of Scroll 1.

Introduction

Reading the Herculaneum Papyri presents many challenges because the scrolls are badly damaged. One of the challenges is to efficiently and correctly identify the surfaces of scrolls and establish which layer of the scroll the surfaces belong to. Manually delineating surfaces with the help of automated tools has made some progress. This is time consuming, so fully automated pipelines that can unwrap scrolls are also being developed [3]. This report describes monthly progress on a recently-developed pipeline for automatically delineating surfaces.

DAFF algorithm efficiency improvements

The DAFF algorithm that was described in full in last month's report [2] has been made more efficient by making the following changes:

1. Plane vectors are now compile-time constants rather than run-time constants.
2. Prior to flood-filling, the potential fillability of all neighbour voxels of each surface voxel is pre-computed and stored in a bitmask associated with every voxel so that fillability assessment doesn't involve visiting all neighbouring voxels.
3. The floodFillSeek function (which looks for the midpoint between two intersection-causing fillings) now visits less then half of the voxels that it previously visited in order to search for the midpoint.

Stitching together patches

The algorithm for stitching together surface patches has been rewritten in C++ (jigsaw2.cpp) and its ability to assemble patches

correctly has been improved. Rather than adding patches on a first-come, first-serve basis, it now grows patches one patch at a time, looking for the best match at every addition step. This prevents worse matches (e.g. from false verso surface patches) from blocking better matches.

This algorithm has also been made to take account of fibre information produced by Sean Johnson from his work on using nnuNet to identify horizontal and vertical fibres in a region of scroll 1. When two patches are on the same horizontal fibre, this adds weight to the match score for those two patches. This appears to improve performance, but hasn't been formally evaluated yet.

An efficiency improvement was made by caching match results, so that when the NeighbourTest function is called to redo a test that has already been done, the cached result is used.

The algorithm now works as follows, growing surface patches by merging with other patches:

1. Begin iterating i through each surface patch (ordered from large to small).
2. Find the best matching patch for i, and merge that patch into i. The best matching patch is the one with the largest abutting edge, provided that the length of the edge is greater than the one quarter of the square root of the volume of the smallest patch, so long as the result of merging doesn't cause intersection larger than one-thirtieth the volume of the smallest patch. If two patches lie on the same fibre then this has the same weight as an abutting edge size of 200.
3. If a match was found for i repeat step 2 for the same patch i.
4. Move onto the next i and go to step 2.
5. If any patches were merged, go back to step 1.
6. Export all of the patches.

The constants in step 2 were arrived at through trial and error and have not been optimised. There are still some improvements that could be made to this algorithm, these are discussed in a later section.

Filling holes in extracted surfaces using interpolation

Surfaces obtained from the pipeline often contain small holes or gaps that can be filled fairly easily through simple linear interpolation from one side of the hole or gap to the other. The interpolation procedure is described below.

1. A surface is rendered via flat projection. The coordinates of each projected voxel are stored in a rectangular array.
2. For every zero pixel p in the projected image, look for the coordinates of the projected voxels for the nearest non-zero pixels in the $+x$,$-x$,$+y$ and $-y$ directions. So long as we have at least a pair of coorindates for $+x$ and $-x$, or $+y$ and $-y$, use linear intepolation to estimate the coordinates of the voxel that should be projected to p.
3. Once all we have voxels corresponding to every zero pixel that can be interpolated, re-render again using flat projection.

Metrics for performance comparison

In order to assess the impact of changes to algorithms in the pipeline and to be able to optimize parameters, it is necessary to have metrics characterising the outputs. Up until recently, this had been done informally by look-

ing at patch and surface file sizes and the sizes of output images.

This has been formalised by producing the following metrics:

1. Total surface voxels : number of surface voxels found by the initial basic image processing surface extraction step.
2. Total dense voxels : number of voxels in the volume that are above a threshhold (currently set at 110 in an 8-bit image). The ratio of 'Total surface voxels' to 'Total dense voxels' gives some indication of how good surface extraction is at finding surfaces.
3. Total patches : total number of patches produced by the DAFF algorithm.
4. Total patch size : total number of voxels in all patches. This will be smaller than 'Total surface voxels' because DAFF has a patch size threshhold (currently set at 2000) below which patches are not exported.
5. Patch size histogram
6. Total surfaces : total number of surfaces output after patch sticthing. If patch stitching is successful then the ratio of 'Total surfaces' to 'Total patches' will be low. In practice less than 0.5 seems good.
7. Total surface size : total number of voxels in all surfaces. This will be close to total patch size, the difference between the two is the number of voxels in common between patches - these are plug voxels produced by the DAFF algorithm.
8. Surface size histogram
9. Number of large surfaces (>= 131072 voxels) : When this number is large it means that patch stitching was successful. Currently the best score obtained for this was 23 in a volume in the +y direction from the scroll umbilicus with a lot of surfaces with small air gaps between each. (x=3988, y=2512, z=1500).

These metrics are produced prior to the surface interpolation step.

To illustrate what these metrics look like for a couple of volumes, figure 1 shows a plane from a volume in which the pipeline doesn't perform well. The metrics for this volume are:

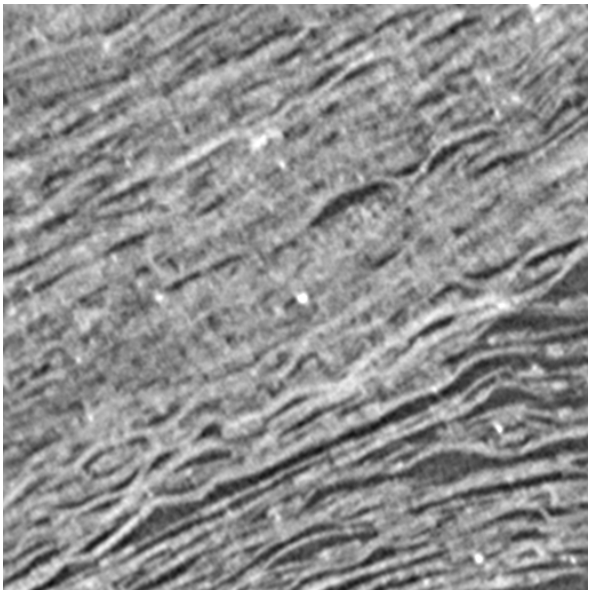

Figure 1: A plane from a $512 \times 512 \times 512$ volume in which the pipeline does not work well

```
x=3476, y=1488, z=1500

Total dense voxels: 108369109
Total surface voxels: 9913857

Total patches: 1236
Total patch size: 6146941
Median patch size: 3596

Total surfaces: 575
Total surface size: 6109656
Median surface size: 4354
Number of large surfaces: 3
```

Figure 2 shows a plane from a volume in which the pipeline performs well. The metrics for this volume are:

```
x=3988, y=2512, z=1500

Total dense voxels: 90228447
Total surface voxels: 10996202

Total patches: 1046
Total patch size: 9538552
Median patch size: 5302

Total surfaces: 373
Total surface size: 9472227
Median surface size: 4589
Number of large surfaces: 23
```

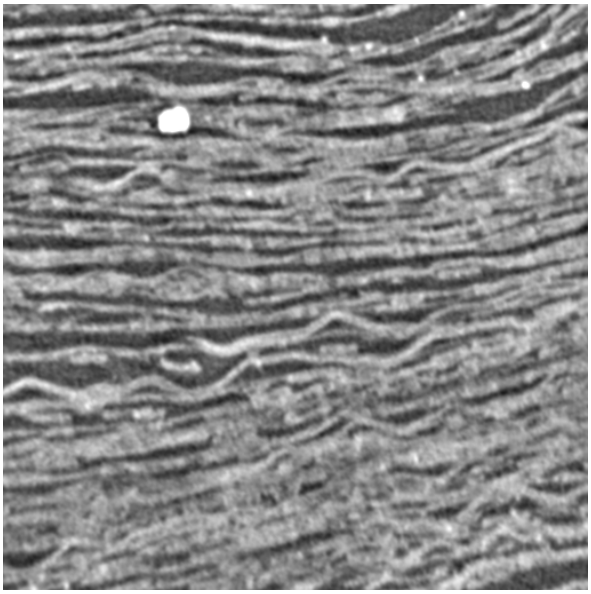

Figure 2: A plane from a $512 \times 512 \times 512$ volume in which the pipeline works well

Surface voxel identification

No work has been done on surface voxel identification this month - it seemed more important to focus on improving the efficiency of the pipeline, producing metrics for assessing performance, and incorporating work on fibres, which may be another way of getting into inaccessible scroll regions.

Example output

Figure 3 shows surfaces obtained from six $512 \times 512 \times 512$ volumes in an area known to contain easily visible ink crackle - this is almost but not quite a single wrap of scroll not far from the scroll umbilicus (which resides in a $512 \times 512 \times 512$ volume in the middle of this wrap. This image is from before the interpolation step - most of the holes in the surfaces can be filled by interpolation, but for the purpose of reporting how the pipeline performs it seemed better to present the pre-interpolation images.

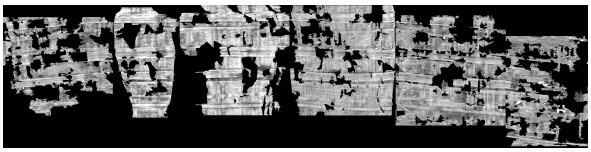

Figure 3: Several adjacent surfaces that form almost a complete wrap at z=1500

Figure 4 illustrates surfaces obtained from the volume that has the best metrics so far : a total of 13 near-complete surfaces were obtained, and a further 13 with more than half of the surface complete.

Performance

Running the pipeline starting from a stack of 512 16-bit TIFF images, each 512×512 pixels, and ending with an interpolated rendering of the sitched-together surface patches found in the volume takes about 16 minutes on a single vCPU of an AWS EC2 t3.medium instance: 12 minutes for the DAFF algorithm and most of the rest of the time for the patch stitching program. Since a t3.medium instance has 2

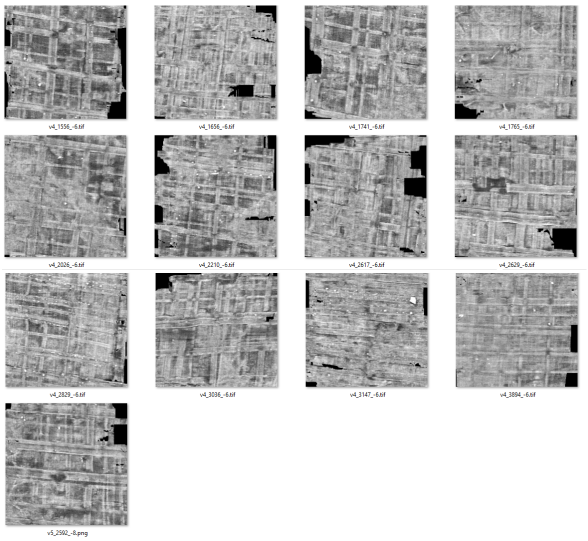

Figure 4: Some surfaces output by the pipeline for a single $512 \times 512 \times 512$ volume

vCPUs, the time per volume is about 8 minutes.

The 2023 Prize Banner is contained in about 700 $512 \times 512 \times 512$ volumes, so it would take about 4 days to process using a single t3.medium instance.

Future improvemens

Further optimizations of DAFF

The gprof profiling tool shows that 'resetFill' is one of the most time consuming functions in the DAFF algorithm, taking up 17% of the execution time. The role of this function is to restore the working volume and projection plane back to their original state. This could be made more efficient by storing a list of addresses that need to be reset, rather than a list of coordinates. Also, the conditional operation in the inner loop of this function could be replaced by a bitwise AND operation.

Further optimizations of the jigsaw2 patch stitching algorithm

Since this algorithm now takes up 25% of the pipeline time, it is worth making it the focus of future optimization. A major source of inefficiency in the current implementation is that pointset intersection of similar patches is calculated again and again. Currently caching is only implemented to avoid recalculating identical results, but it ought to be possible to compute pointset intersection once only for every pair of patches, then use algebra to calculate intersections for merged patches.

One problem that has sometimes been observed is a 'blocking' phenonemon in which a false verso surface is merged into a patch and subsequently prevents merging of true surface patches. It is possible that a simulated annealing [4] approach could help here. The objective function to be maximised could be the number of shared patch voxels (i.e. the plug voxels produced by the DAFF algorithm in the hole-filler program) minus a penalty function defined as some constant multiplied by the number of voxels in a patch that project to the same point. A connectivity matrix representing which patches are joined to which other patches would have 0 for all patch-pairs with no possibility of abutting, but could be 0 or 1 for patch pairs that abut. Simulated annealing would make random changes to the connectivity matrix, where the change probability is determined by the 'temperature' and the difference in objective function value resulting from the change. As the 'temperature' cools, the connectivity matrix moves towards an optimum solution - perhaps a local optimum, which might be good enough.

When thinking about sheet flattening, I also wondered whether an assessment of 'flattenability' could be used to score how good a surface is : if a surface is physically impossible to flatten then something must have gone wrong during it's construction.

Interpolation

Surface interpolation is currently just linear interpolation between the edges of a hole. It could perhaps be improved by trying to flex the interpolated surface so that gradients at the edges match the interpolated region, or alternatively trying to find the local minimum of the total scroll density in the interpolated surface - this might work when interpolating in a small compressed region.

Problematic volumes

Figure 5 shows an image from a volume that the pipeline can't handle at the moment because the surface orientation changes by more than 180 degrees within the volume. The most straightforward way to handle this is to split the volume up into separate parts in which the 180 degree constraint is met and process each part separately, masking out the offending area and then merging the results at the patch-stitching stage.

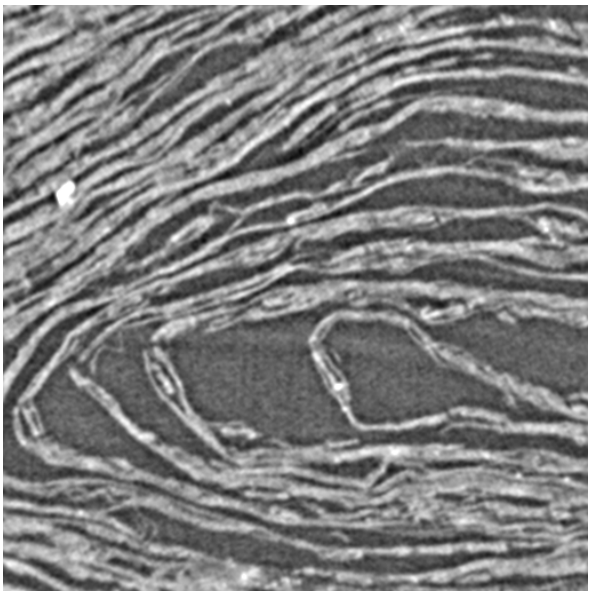

Figure 5: A plane from a $512 \times 512 \times 512$ volume in which the surface orientation changes by more than 180 degrees

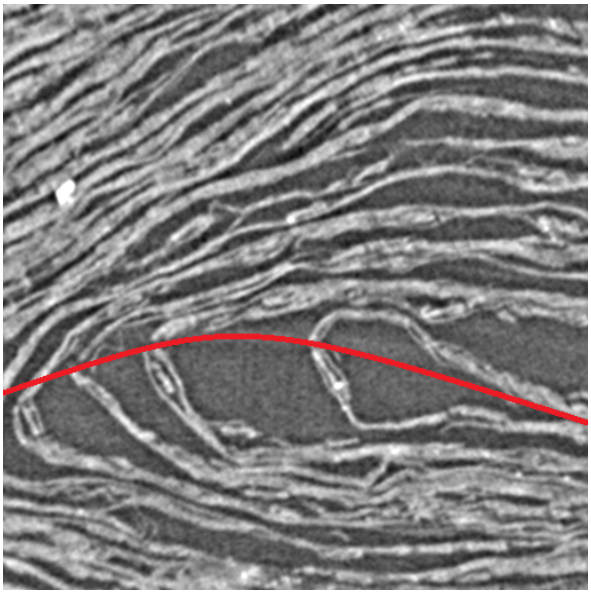

Figure 6: The problem can be resolved by splitting the volume into parts, and masking out all but one part at a time.

Using GPUs

So far GPUs have not been used for this project, except for learning about PyTorch on a g4dn.xlarge instance. The basic image processing operations that preceed the DAFF algorithm in the pipeline could easily be run on GPUs using existing image processing libraries.

Pre-computation of flood-fill neighbour fillability, described in the previous subsection, could also be implemented easily using GPUs - effectively off-loading more computationally-intensive work from the inner loop of flood-fill.

Flood-fill itself is not so easy to parellelise, but it would probably be possible to run some of the multiple flood-fills needed by DAFF simultaneously.

Since the project is running on AWS EC2 instances and GPU instances are about 10 times the price of non-GPU instances, then from a cost persepctive it is only worth using GPUs if the resulting performance increase is >10-fold. From a development perspective, a less-than-

10-fold improvement is still worth having if it increases the speed of the debug/development cycle.

The rest of the pipeline

So far I have not looked at flattening or ink detection. I hope to use 3D ink detection and physics-based sheet flattening. I have started learning how to use PyTorch to explore ink detection and run existing models. Some years ago I worked on Smoothed Particle Hydrodynamics [5] models in CUDA [6] and I want to investigate whether this type of approach could be used for sheet flattening.

Source code

Source code used in this project can be found at `https://github.com/WillStevens/scrollreading`.

References

[1] `https://github.com/WillStevens/scrollreading/blob/main/report.pdf`

[2] `https://github.com/WillStevens/scrollreading/blob/main/report2.pdf`

[3] `https://github.com/schillij95/ThaumatoAnakalyptor`

[4] `https://en.wikipedia.org/wiki/Simulated_annealing`

[5] `https://en.wikipedia.org/wiki/Smoothed-particle_hydrodynamics`

[6] `https://www.youtube.com/watch?v=TBe1chNfx8w`