Improvements to pipeline for growing large flat sheets on scroll surfaces

William Stevens

31st July 2025

Improvements to pipeline for growing large flat sheets on scroll surfaces

This report describes progress on an automated segmentation pipeline described previously [1]. Over the last month work has been done on improving the data structures and algorithms used by the pipeline. Three problems described in the previous report were: 1.) How to efficiently represent a vector field. 2.) How to efficiently represent a growing surface. 3.) How to address misalignment between a surface constructed from patches that fork apart then come back together again. Work has been done to address all of these issues. Example output measuring 12cm by 2cm is shown.

Introduction

In previous reports I described a method of using surface predictions for growing flat surfaces based on a physics-based simulation of a flat surface [2]. This method seems to be able to detect at least some areas where the surface predictions are wrong, and avoid those areas.

The first step is to produce a vector field from a surface prediction zarr, where the vector field at each point near a surface points towards the surface. The surface predictions for scroll 1A produced by Sean Johnson in May 2025 were used: `https://dl.ash2txt.org/community-uploads/bruniss/scrolls/s1/surfaces/s1_059_ome.zarr/0/`

Next, flat patches are grown, constrained by the vector field. Patch growth stops when stress in any part of the patch exceeds a threshhold, or when the patch reaches a prespecified size limit. Pathces are stitched together to make a larger flat surface.

Efficient representation of a vector field

Previously I had used a $2048 \times 2048 \times 2048$ vector field zarr where each component of the vector field was represented as a 32-bit float. I have since found that I was able to use a single byte for each component instead, resulting a reduction in the storage space needed for the vector field zarr from 32.2Gb to 13.8Gb after compression. I also increased the size of the smoothing window from 5 to 7, and used Gaussian smoothing rather than window-average smoothing - this was to attempt to reduce the ripples that previously appeared in the growing sheet - there is now a small force further into the gap between the vectors that point towards the scroll surface. This does seem to have reduced the ripples in most places, but they remain in a few places notably at the bend shown in figure 1.

It may be possible to make another small improvement to the space required to store the vector field : the precise direction of the vector is probably not important, so the vector

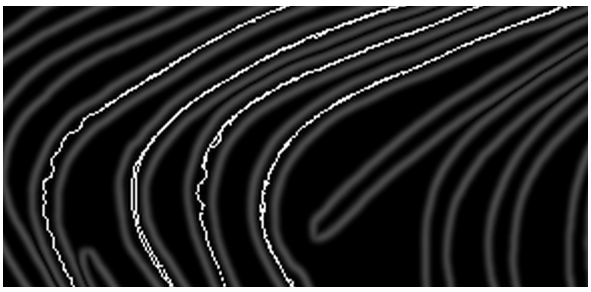

Figure 1: Image showing how the surface grows between the smoothed vector fields. In this bend in the surface, ripples appear on some surfaces. Also, it is possible to make out overlapping patches - the white line has a two-stranded appearance - each is a different patch.

could be represented as an index into a table of evenly spaced points on the sphere, along with a magnitude. This could probably be done using two bytes - e.g. a 10-bit index into 1024 evenly-spaced Fibonacci points on a sphere, and a 6 bit magnitude. However, this has a higher computational cost, and after taking compression into account it might not make any difference to storage space requirements. With or without this improvement, it is likely that the vector field for all of Scroll 1A will be less than 1Tb.

Efficient representation of a growing surface

Previously a CSV file had been used to represent patches and surfaces, where each row contained $(q_x, q_y, v_x, v_y, v_z)$. Where q_x, q_y are the coordinates of a quadmesh vertex and v_x, v_y, v_z are where the location of that vertex in the scroll volume.

This became inefficient when surfaces became larger: the operations of aligning a patch with a surface and finding the neighbours of a quadmesh point were proportional to the size of the surface, and after a few hundred patches these were the rate-limiting steps.

To overcome this a binary chunked format has been implemented, compressed using blosc2. This format contains both the surface (chunked based on scroll volume coordinates) and an index into the surface (chunked based on the quadmesh coordinates), so that neither of the operations described above depends on the size of the surface.

A surface chunk contains a list: $(q_x, q_y, v_x, v_y, v_z, n)$ where n is the patch number that the quadmesh point belongs to.

The index is chunked based on the quadmesh coordinates, and each chunk contains a list of the surface chunks that contain any quadmesh coordinates corresponding to the index chunk. The index files are uncompressed CSV files, since they are not large and are not frequently used.

When a patch is added to the surface, it's quadmesh coordinates are transformed to align with the patches already in the surface. This is quite a slow operation - second only to the patch growing algorithm, and in future tis will be deferred until surface rendering (in part because the potential solution to the fork-misalignment problem described later on requires that patch transformations can be updated).

When patches overlap, all overlapping patches are represented - there is no need to handle overlap untli the surface is rendered, and then only one patch (the first one encounted in a chunk) is rendered.

Aligning patches with the surface

When a patch is added to the surface, its quadmesh coordinates are transformed to align with the patches already in the surface. Previously I had done this by looking for the nearest point s to each point p in the patch, and calculating an affine transformation using randomly

selected pairs of points in the patch and surface.

Even though the algorithm required a minimum separatation of 100 voxels between pairs of points, this sometimes resulted in an accumulated misalignment over the course of adding several patches.

To try and improve this, the algorithm for adding a patch to a surface was modified as follows:

1. For each point p in the patch.
2. For each patch currently in the surface
3. Find the nearest three points s, t, u to p
4. Check that these three points are three adjacent quadmesh points that form a right-angle in the quadmesh. Let s be the corner point.
5. Using volume coordinates, construct vectors $\vec{st}$ and $\vec{su}$ and $\vec{sp}$
6. Check that $\vec{st} \cdot \vec{sp}$ and $\vec{su} \cdot \vec{sp}$ are both between 0 and 1. (i.e. p lies roughly in the quadmesh square that s, t and u are part of).
7. The vectors $\vec{st} \cdot \vec{sp}$ and $\vec{su} \cdot \vec{sp}$ can then be used to give the fractional part of the quadmesh coordinates that p must have to align it with the surface - i.e. where within the quadmesh square that s, t and u are part of does p lie?

Compared with the previous method, the variance among the transformations found by this method was always smaller. Inspecting the output surfaces produced using the two different methods (Figure 2) shows improved alignment.

Results and future work

Figure 3 shows example output, which measures about 12cm by 2cm and wraps four and a half times around the scroll umbilicus (as shwon in figure 4. Unexpectedly, where growth of the surface forks due to a high stress region, the two paths resulting from the fork are sometimes able to realign. Not all forks were able to realign however, and an idea for how this can be resolved is described below.

This took about 1 day to produce. It needs a 1000-fold improvement in performance to be able to process all of scroll 1 in a reasonable time. Given that I am using a single-threaded pipeline on modest hardware about 10 times slower than the fastest available, that I am using 1-point per voxel resolution for the surface (which I beleive could be reduced to 1 point per 3 voxels, since this is the patch growing resolution), and that I haven't yet written the patch-growing algorithm in CUDA, a 1000 fold improvement seems completely reasonable.

As the surface grew, the rate at which patches were accepted for alignment slowed : this happens because sometimes a patch aligns okay with two separate patches in a surface, but the affine transformation for each is different enough that the new patch is rejected.

To solve this patch rejection problem, which is closely related to the fork-realignment problem, I have started working on a prototype program that represents patches and their relationship with each other using a ball and spring model. The model represents the distance and relative angle between patches, and whenever a new patch is added the model is simulated until it reaches equilibrium - so any misalignment between forks that are joined up by a new patch propagates back along all of the connections that indirectly link the patches. Stress measurements could be used in a similar way to the way that they are used in the patch growing program to detect any really bad patch placements and reject them. Tak-

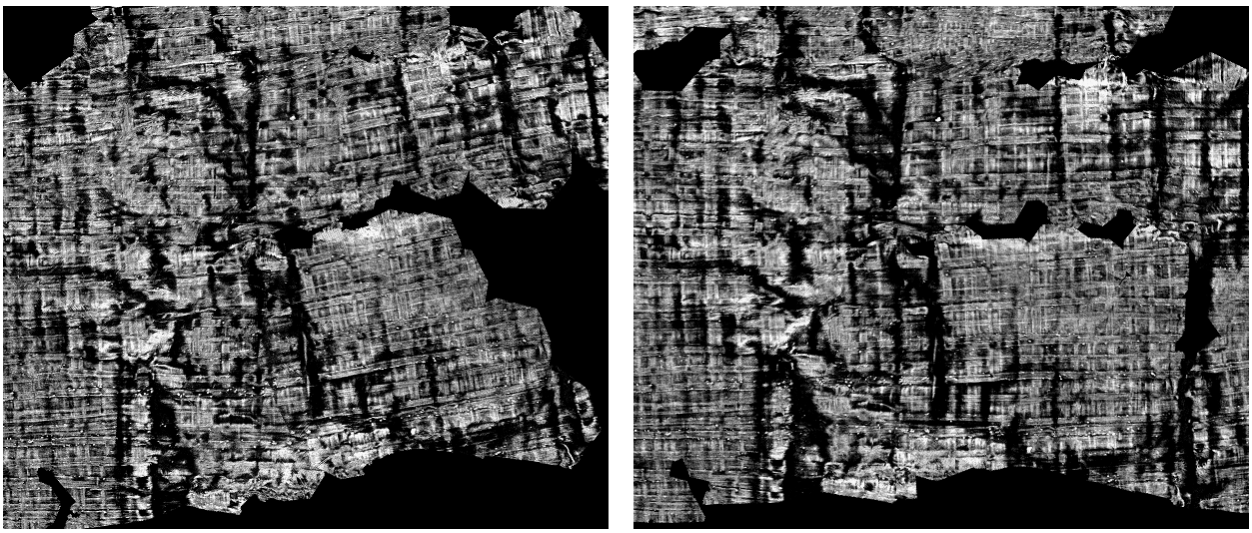

Figure 2: The left hand image shows the consequences of poor patch alignment. The right hand image shows how better patch alignment improves the situation.

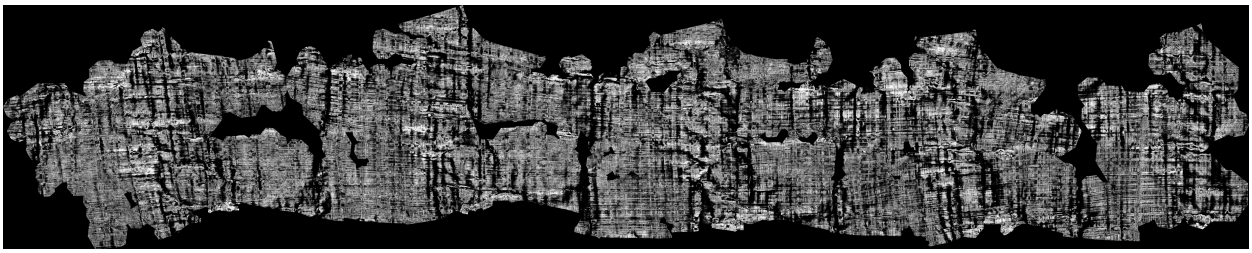

Figure 3: A 12cm by 2cm section of surface, stitched together from no more than 1500 patches.

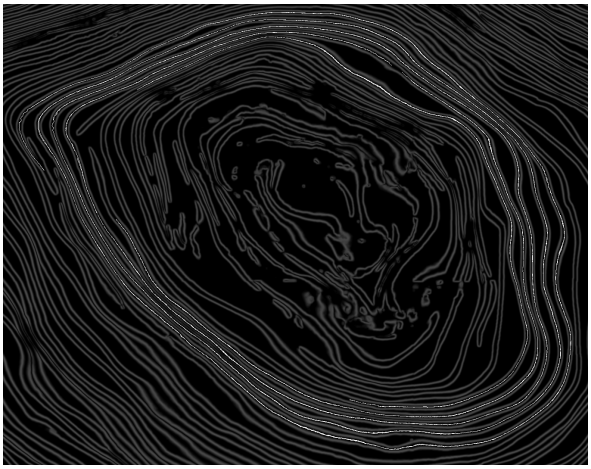

Figure 4: The surface wraps 4.5 times around the scroll umbilicus.

ing it a step further, patch alignment variance could be used to determine the spring stiffness between patches - alignments with large variance would have looser springs and greater freedom to move, because there is less certainty about how correct the original alignment was. For this method to work, patches won't be transformed to match the surface orientation at the time they are aligned - instead the alignment will be stored in an index, so that it can be updated by the ball and spring model - and this index will be used at rendering time to carry out the actual transformation.

Source code

Source code related to this report can be found at `https://github.com/WillStevens/`

scrollreading/simpaper.

The following programs related to this report have been updated or created over the last month:

1. zarr_show2_32.cpp : Show a slice through a zarr, optionally with surface points plotted on it. Updated to work with either the chunked surface data structure or CSV patch files.

2. bigpatch.cpp : Library for representing surfaces using a compressed, chunked binary format.

3. bigpatch1.cpp : Original prototype version of bigpatch that used an uncompressed chunked CSV format.

4. addtobigpatch.cpp : Add a patch into a surface.

5. mask_add.py : Add a patch mask produced by render_from_zarr4.c into a tif image representing the surface mask - used to detect the boundary of the surface. This was written to make surface mask creation much less dependent on surface size. There is still room for improvement - for very large surfaces the surface mask will probably need to be represented as a zarr so that mask updates and edge detection only run on changed chunks.

6. gaussian_kernel_3d.py : Outputs C code for constant initialisation of a 3D guassian kernel.

7. papervectorfield_closest2.c : A program for producing a vector field zarr from a surface prediction zarr.

8. vectorfield_smooth_32.c : Smooth a vector field using a Gaussian filter with a window size of 7.

9. sp_pipeline2.py : The pipeline program that calls other programs to implement the sheet growing and stitching pipeline.

10. simpaper6.cpp : A C++ version of the sheet-growing program that will run for upto 300 iterations (corresponding to a patch width of about 900 voxels) or until high stress is encountered. Produces a CSV output of the particle positions along with data for the stress plot.

11. extent_patch.cpp : Output the x,y,z extent of a patch

12. align_patches4 : Given a surface and a patch, work out a rotation and translation for aligning the patch with the surface. Returns the six non-constant elements of an affine transformation matrix. Also output the variance of each element of the affine transformations resulting from a random sample of pairs of points that the two patches have in common. The variance is used to detect when one patch is flipped w.r.t the other.

13. find_nearest3.cpp : Find nearby points to a specified point. Used when working out plane vectors for a new seed point.

14. render_from_zarr4.c Render a patch or surface using the scroll 1A zarr, making use of zarr functions generated by zarrgen.py. Can also be used to produce a mask showing when a patch is present, which is used by the boundary detection code.

15. patchsprings.py : Prototype program for solving the fork realignment problem. This is being actively worked on at the time of writing.

References

[1] https://github.com/WillStevens/scrollreading/blob/main/report.pdf to https://github.com/WillStevens/scrollreading/blob/main/report7.pdf

[2] https://github.com/WillStevens/scrollreading/blob/main/report6.pdf to https://github.com/WillStevens/scrollreading/blob/main/report7.pdf