Pipeline for growing large flat sheets on scroll surfaces

William Stevens

30th June 2025

Pipeline for growing large flat sheets on scroll surfaces

This report describes progress on an automated segmentation pipeline described previously [1]. Over the last two months work has been done on exploring how to detect and avoid areas where surface predications contain errors. This is done by growing a simulation model of a flat surface, and measuring the strain energy at each point as the surface grows. It was found that areas of high strain energy often correspond to locations where the surfae predication has gone wrong, including where sheet switching takes place. This information has been used to grow patches from a seed, and stop when the strain energy exceeds a threshhold. The result is a lot of patches with low strain energy that can be joined together to make a larger surface. To facilitate this work, a code-generating zarr library written in Python and C was developed. This library supports any number of dimensions and a range of data types, including structured data types.

Introduction

In a previous report I described a method of using surface predictions for growing flat surfaces based on a physics-based simulation of a flat surface [2]. An advantage of this method is that stress in the simulation seems to correspond to areas where the surface prediction is going wrong. It has the potential to detect sheet switching and other problems with surface predictions.

I have developed this work by creating a pipeline that will grow many flat surface patches and stitch them together, avoiding areas of high stress.

I used the surface predictions for scroll 1A produced by Sean Johnson in May 2025 at this location: `https://dl.ash2txt.org/community-uploads/bruniss/scrolls/s1/surfaces/s1_059_ome.zarr/0/`

A $2000 \times 2000 \times 2000$ volume of this zarr starting at $x = 2688, y = 1536, z = 4608$ was turned into a vector field zarr using the method described in the previous report [2] : essentialy for each voxel in the volume that lies near a surface prediction, a vector points towards the nearest surface voxel. This vector field is used to exert a force on the growing flat surface to keep it contained within the surface prediction.

Examination of high stress areas

Several areas of high stress were examined when a patch is grown from a seed.

Figure 1 shows a large patch and the corresponding stress plot. An area of high strain energy is highlighted in the red oval in both images. From looking at the surface predications dataset, it is clear that this is an area where the predictions lead to sheet switching. When a flat surface is grown in this area it ends up following the wrong surface (illustrated in figure 2), leading to high stress.

Figure 3 shows a small area of stress on a stress

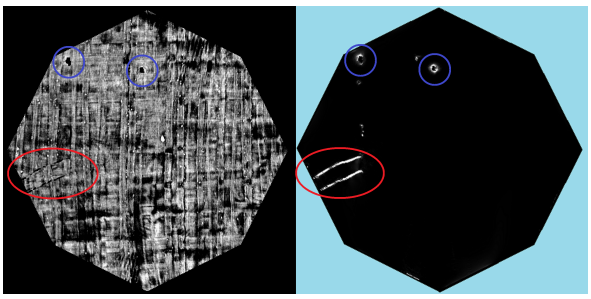

Figure 1: The red oval highlights an area where sheet switching takes place. The rendered surface between the parallel lines is from the wrong sheet. The stress plot shows that the edges of the wrong sheet have a high strain energy.

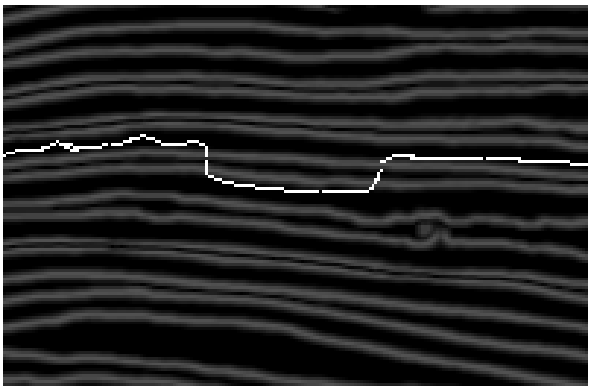

Figure 2: If the surface predictions contain a sheet switch then the growing flat surface may follow the wrong surface, leading to high strain energy.

plot, which occurs on an otherwise okay patch. Looking at how the flat surface grows within the surface prediction for this area shows that this is an area where the surface prediction drops out in a small area, also shown in figure 3. In this case, continuing to grow the sheet around the high stress area seems to do no harm.

How should this information be used? I've attempted to use this information to grow small patches that stop growing (in all directions) as soon as high strain energy is encountered anywhere. The small patches are stitched together.

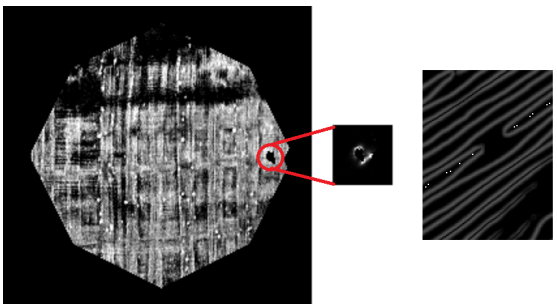

Figure 3: A small area of stress that corresponds to a small dropout in the surface prediction.

This seems to successfully avoid the area that causes sheet switching in this example. Figure 4 shows the same area as figure 1, but where the problematic area is left alone. This shows that two other small high stress areas (highlighted in dark blue in figure 1) are also avoided.

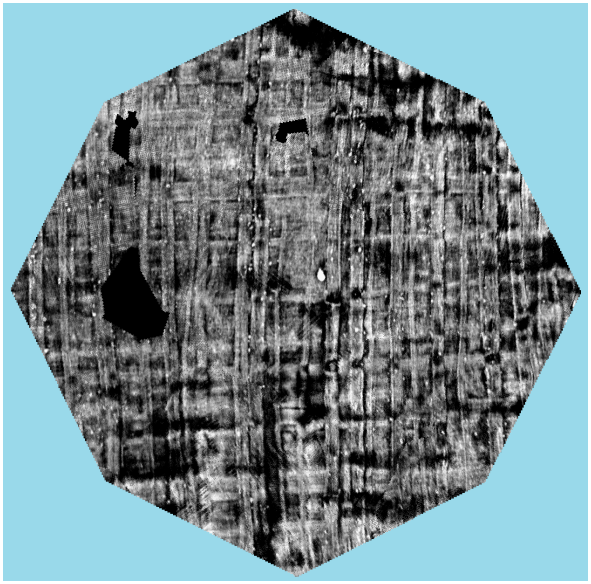

Figure 4: The same surface shown in 1, produced from stitching together stress-avoiding patches. The high stress areas are not present, and the surface between the parallel lines is now correct.

Pipeline

I created a $2048 \times 2048 \times 2048$ vector field zarr, centred roughly on the scroll umbilicus, so that I can explore making large surfaces that wind around the umbilicus several times.

The pipeline has the following steps:

1. Manually specify the coordinates of a seed, along with two vectors that specify the plane that the seed should grow in initially.
2. Grow a patch. Stop after 150 growth iterations or when high stress occurs.
3. If it's the first patch, make it the current surface. Else try to align the patch with the current surface (flip it and try again if it has points in common with the current surface but won't align).
4. Transform the patch and add it to the current surface.
5. Detect the boundary of the current surface.
6. Pick a point at random on the boundary of the surface.
7. Make a new seed at this point. The plane vectors for this new seed are obtained using singular value decomposition on all of the points within a small radius of the new seed point - it will start growing in the same plane that it's neighbours approximately lie on.
8. Go to step 2

In step 3, a patch aligns with a surface if multiple points on the patch can be tranformed to points on the surface using the same transform (determined by checking that the variance of a random sample of such transforms is small). If multiple points can be transformed to the surface, but each needs a different transform (determined by detecting that the variance of a random sample of transforms is large) then this is an indication that the normal of the patch points in the wrong direction - in this case the patch is flipped an alignment is reattempted.

During development, patches and partially grown surfaces were rendered after every step to check on progress and look for errors. When the rendering steps are removed, the slowest parts of the pipeline are steps 2 and 4.

Preliminary results

Figure 5 shows the area where a test surface was grown - it extends for just over one wrap around the umbilicus. Figure 6 shows this 3.5cm by 2.1cm section of surface, stitched together from 340 patches, each of which has a diameter of no more than about 3.6mm.

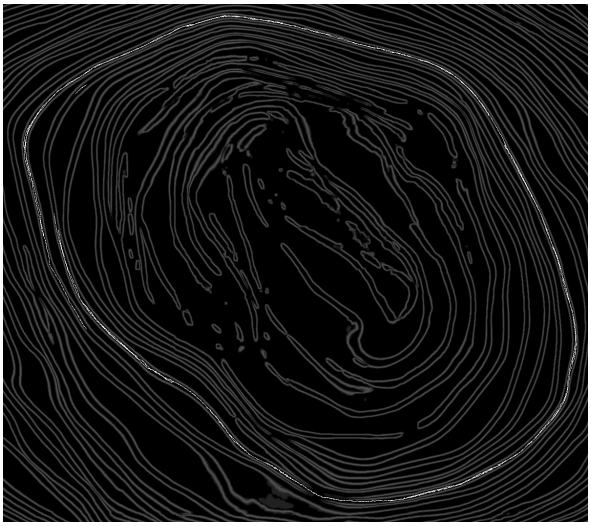

Figure 5: A cross section of surface predictions for scroll 1A, showing where the surface was grown.

In the left hand side of the image, the large scale flatness looks reasonable. There is confusion around the middle of the image - exactly what is causing this has not yet been examined - and the right hand side of the image has split

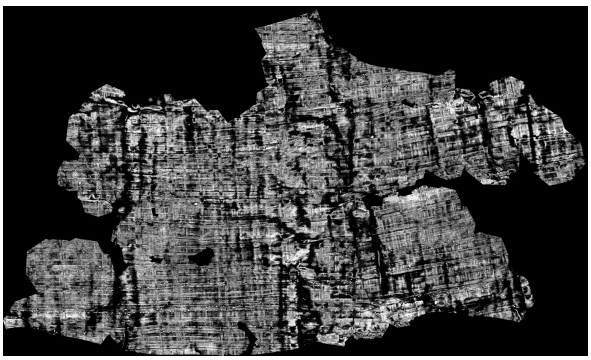

Figure 6: A 3.5cm by 2.1cm section of surface, stitched together from about 250 patches.

into two separate flat areas. They are both from the same surface, but sligtly misaligned with each other. In future this misalignment issue could perhaps be avoided by being careful about the way that the next patch is chosen in the pipeline : currently a boundary point is chosen at random, which can lead to forking of the surface - if the two forks misalign then generally they won't rejoin. Rather than randomly choosing a boundary point, the boundary point likely to result in the minimum increase in boundary length could be chosen - this would cause any concave parts of the boundary to be filled in and the surface would maintain a roughly circular shape is it grew.

Another possible way of improving the quality of the output could be to change the way that high stress regions are handled during patch growing. Currently growth stops when high stress occurs. A better approach might be to simply disregard vector field forces in any area where high stress develops. Since high stress often seem to arise from problems in the surface predictions, ignoring the surface predictions and allowing the surface to grow subject to flatness constraints alone might give better results in high stress regions.

Code generator for reading and writing zarr arrays

Zarr is a format for storing large multidimensional arrays [3]. I wanted to make a low-dependency, simple, flexible and hackable way of accessing zarr data, having tried (and in some cases failed due to lack of time) to build existing C++ zarr libraries on Cygwin and on an AWS EC2 image. The dependency problems encountered included having a C++ compiler that didn't support the right language version, and having to install an extremely large scientific data format library simply to use one small feature of it (i.e. zarr support). Since the zarr format is quite simple, it seems like there ought to be a simple native C library supporting it. I was able to use the vesuvius-c zarr functions successfully, but this only supported 3D zarrs with u1 data type. The vesuvius-c source code was very useful for helping to understand the zarr format, and how to use the blosc2 library.

The solution I settled on was to write a code generator in python that generates C functions for reading and writing zarr files, given zarr metadata in JSON format. Any-dimensional array support is relatively easy to implement using this approach because it simply requires generating code for C-arrays and C-array access using the correct number of dimensions - the generated code doesn't have to worry about the array index calculations. It is also relatively easy to support any data type - simply map zarr data types onto C data types. This includes zarr's stuctured data types - they can be mapped onto a C struct type. I also wanted to have both random-access functions (useful for protoyping) and more efficient access functions that make assumptions about which chunk is currently selected and how data is arranged in memory. I was able to use this approach to access the Vesuvius Challenge zarrs, and to write 4-dimensional zarrs that store the vector field data described earlier in this report.

Some limitations include: The only compressor I have supported is blosc2, I have so far only mapped data types that I have used, endianness conversion is not supported. The generator must be run to generate functions for each different zarr structure used by a program - this is not much of a problem when writing a data pipeline where the format of all relevant zarrs is known in advance, but would be a problem for other use cases. It also only supports zarrs on a file system. For accessing web-based Vesuvius Challenge zarrs I have worked around this by using the vesuvius-c libraries ability to create local zarr buffers: I used vesuvius-c to download zarrs by reading a single voxel in each chunk that needs to be downloaded.

The code generator takes as parameters the zarr metadata, a suffix to use for the generated functions, and the number of in-memory chunk buffers that should be used by the generated code. The number of buffers needed depends on what the zarr data is being used for. For example, reading surface predictions and writing the vector field described earlier can be done efficiently by writing one chunk at a time (so only 1 output buffer is needed), and making sure the zarr functions for reading the surface predictions have 27 buffers so that almost all of the time all neighbouring voxels of a surface preduction voxel are in-memory. Afterwards, I realised that 8 buffers would probably suffice if the data were processed one-eighth of an input chunk at a time.

The flat surface growing and simulation program (simpaper5) can access the vector field zarr most efficiently by using a small chunk size (e.g. $32 \times 32 \times 32 \times 4$) and a large number of buffers (e.g. 5000). This is because a growing surface patch has growth fronts that cover a small number of points but in widely spaced parts of the zarr - a small chunk size doesn't waste memory and access time, and a large number of buffers means that each chunk is in memory for the whole duration that it is needed for.

The zarr code generator generates the following code:

- A typedef corresponding to the zarr element type : ZARRType<suffix>
- A struct that holds ZARR handling data (including in memory buffers and a pointer to the currently selected chunk for quick access) : ZARR<suffix>
- Functions for opening and closing zarrs: ZARROpen<suffix>, ZARRClose<suffix>
- Functions for flusing a single buffer, and all buffers: ZARRFlush<suffix>, ZARRFlushOne<suffix>
- A function for checking whether a chunk is in memory, and if not then load it and select it: ZARRCheckChunk<suffix>
- Functions for reading and writing single elements from a zarr: ZARRRead<suffix>, ZARRWrite<suffix>
- Functions for reading and writing several elements from a zarr: ZARRReadN<suffix>, ZARRWriteN<suffix>
- A function for writing several elements to a zarr, assuming that the chunk being written is already the currently selected chunk: ZARRNoCheckWriteN<suffix>

Next steps and future work

Sometimes small ripples are apparent in grown surfaces. These can probably be avoided by further smoothing of the vector field so that the gap that the surface resides in is smaller. This will probably also solve the problem of different patches putting the surface in slightly different positions.

The vector field zarr represents a vector as 3 32-bit floats. It will probably work just as well using 3 8-bit values, but take up one quarter of the space.

The generated zarr code doesn't look up in-memory buffers efficiently - it simply searches lineary through a list. The most recently used can be accessed quickly, but others take longer. This becomes particularly noticable when growing large patches when there are 100s or 1000s of buffered chunks. This problem could easily be solved by having an array-like buffer structure, where a low-resolution array with the same number of dimenions as the zarr stores pointers to chunk buffers (or null if the chunk is not in memory). E.g. a $2048 \times 2048 \times 2048 \times 4$ zarr with chunk size $32 \times 32 \times 32 \times 4$ would have a buffer array of size $64 \times 64 \times 64 \times 1$, where the elements of the array are pointers to chunks.

Currently the code generator does not need to know the size of the zarr - this information isn't used anywhere. This feature could be useful when writing a scroll surface zarr that could grow in size to an unknown extent. It doesn't currently support negative chunk indices, but this would also be useful so that surfaces could grow in any direction.

After several hundred patches into a pipeline run, it takes about 5 times longer to produce and stitch a patch to the growing surface than at the start of the run. This is because the runtime of some of the algorithms in the pipeline is proportional to the number of points in the surface. This problem could be avoided by changing the way that surfaces and patches are stored on disk. A surface or patch is a list of 5 numbers: flat surface x,y coordinates, and the 3D coordinate of the scroll voxel at that point on the surface. Currently these are stored in CSV files. It would be better to use a storage format more similar to the zarr format (but zarr is not directly suitable), in which the scroll volume is divided up into chunks, and each chunk contains the list of surface points within that region of the scroll.

Source code

Source code related to this report can be found at `https://github.com/WillStevens/scrollreading/simpaper` (the patch growing and stitching pipeline) and `https://github.com/WillStevens/scrollreading/simpaper` (the zarr code generator).

There are several programs:

1. `zarrgen.py` : Code generator for zarr functions
2. `zarr_show.c` : Show a slice through a zarr, optionally with surface points plotted on it
3. `stripgrid.cpp` : Removes the first 2 columns in a 5 column CSV file containing floating point numbers. Useful for making surface point files that `zarr_show.c` can use.
4. `papervectorfield_closest.c` : A program for producing a vector field zarr from a surface prediction zarr.
5. `sp_pipeline.py` : The pipeline program that calls other programs to implement the sheet growing and stitching pipeline.
6. `simpaper5.cpp` : A C++ version of the sheet-growing program that will run for upto 150 iterations (corresponding to a patch width of about 450 voxels) or until high stress is encountered. Produces a CSV output of the particle positions along with data for the stress plot.
7. `interpolate.cpp` : Interpolate output from simpaper5 to increase the resolution by a factor of 3×3
8. `extent_patch.cpp` : Output the x,y,z extent of a patch

9. `align_patches2` : Given two patches, work out a rotation and translation for aligning the second with the frst. 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.

10. `flip_patch.cpp` : flip a patch about the x-axis. Useful if `align_patches2` shows that the patches overlap but no single transformation aligns them.

11. `transform_patch.cpp` : Transform a patch using an affine transformation.

12. `normalise_patch.cpp` : Make the minimum x,y coords of the patch equal to 0,0.

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

14. `render_from_zarr2.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. `render_stress.c` Draw a stress plot using output from simpaper5.

16. `render_stress.c` Render a stress plot.

References

[1] `https://github.com/WillStevens/scrollreading/blob/main/report.pdf` to `https://github.com/WillStevens/scrollreading/blob/main/report6.pdf`

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

[3] `https://zarr.dev`