VOX-FE: New functionality for new communities

Neelofer Banglawala[1], Iain Bethune[1], Michael Fagan[2] and Richard Holbrey[2]

[1]EPCC, The University of Edinburgh, [2]The University of Hull

Version 1.1, June 10, 2016

1 Introduction

This report documents the work of ARCHER eCSE project 04-11 to improve and extend the VOX-FE finite element bone modelling suite. VOX-FE has been developed as in collaboration between EPCC and the University of Hull. VOX-FE consists of three parts:

- The graphical user interface - a ParaView plugin which supports model construction from volumetric data (e.g. micro-CT image stacks), boundary condition definition (forces and constraints), and visualisation of strains and other derived properties computed by the solver. New functionality added to the GUI is outlined in Section 2.
- The solver - a parallel MPI application which given a model definition, constructs and solves the linear elasticity problem to compute the displacements of each node. A new model input and load-balancing algorithm is described in Section 3.
- The remodeller - a scripting harness to couple together multiple executions of the solver to examine how bone adapts its structure under load. The remodelling process is now largely automatic, and is discussed in Section 4.

VOX-FE is freely available from https://sourceforge.net/projects/vox-fe/ under a BSD licence. The current released version is 2.0.1, and the next release 3.0 is planned pending the implementation of a parallel I/O method to enable the solver to produce correctly ordered output displacements (currently implemented as a post-processing script).

2 New Functionality in the VOX-FE GUI

2.1 Intensity Mapping

The VOX-FE plugin can read image data in several different forms (2D formats: jpeg, bmp, tiff, png and the 3D ITK mhd format). These are converted into the canonical VOX-FE script and model file formats, which can also be read directly by the solver and the plugin, so the image files themselves are not needed for re-importing later. Since version 2.0, the VOX-FE solver supports an arbitrary number of different 'materials' to be defined, enabling each imported voxel to have its own elastic moduli, rather than assuming it is uniform across all bone voxels.

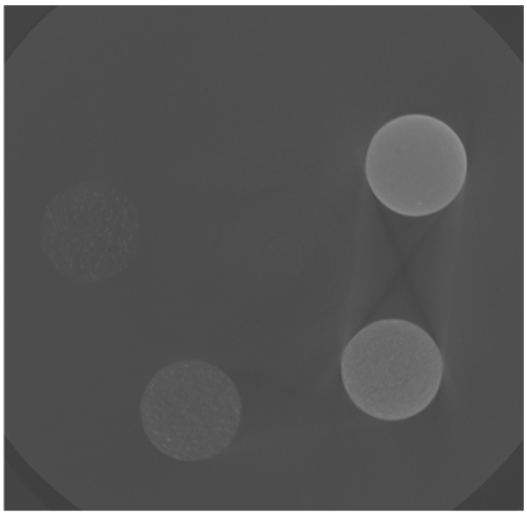


Figure 1: Example of a (2D) CT scan of four discs of known material properties, used to calibrate the greyscale-to-density mapping.

Raw intensity values (luminescence in grey-scale images) can be mapped to elastic moduli via an assessment of sample density if samples of known density are available to calibrate the CT machine (see Figure 1). A simple linear mapping between intensity and density can usually be assumed, but to compute elastic moduli from densities, the VOX-FE2 plugin implements a power mapping of the form presented by Carter and Hayes in 1977 [1] (quoted here from [2]), e.g.

Elastic Modulus, $E = 3.79\epsilon^{0.06}\rho^3$

where ϵ is the strain rate and ρ the apparent density.

In VOX-FE, we adopt the form:

$$E = A\rho^B + C$$

where A, B, C can be set in the filter dialog as the 'Equation coefficients'.

2.2 Muscle-wrapping boundary conditions

Simple point boundary conditions are easy to apply within the GUI plugin: points are selected and the VOX-FE2 interface allows that data describing the load or axial constraint can be attached. Points within the same selection share the condition, so that separate BCs require different node selections. However, users would often like to model more realistic conditions such as the force applied by a muscle which is attached at specific points and wraps the bone structure, applying compressive forces on the surface. We have implemented the model of Grosse *et al* [3] which sets out a methodology to permit the modelling of muscle forces in terms of both tangential and compressive (normally-oriented) forces distributed over the area which the muscle is thought to be attached. In essence, this wrapping model views the compression of the bone as dependent upon 2 factors:

- the path length over which the muscle fibres are seen to be attached - the fibres become thicker and/or more numerous, so the load increases over the length of the path; and
- the curvature of the bone around which the muscle passes.

In short, the traction, $\tau_N(\vec{r})$, which compresses the surface can be described by the relation:

$$\tau_N(\vec{r}) = s(\vec{r}).\kappa(\vec{r}).\tau_T(\vec{r})$$

where: $\tau_N(\vec{r})$ is the compressive load, $\tau_T(\vec{r})$ is the traction due to the intended action of the muscle (e.g. in biting), $s(\vec{r})$ is the length of the path of attached muscle and $\kappa(\vec{r})$ is the curvature of the bone attachment area.

Note that a utility (BoneLoad) exists which can compute tangent and normal forces on tetrahedral meshes, but which requires a number of different utilities to run, including the commercial packages MatLab and Strand7, and we have implemented an integrated muscle-wrapping function in the VOX-FE plugin.

Liu *et al* [4] have adapted this method for use with voxel data, using 2D splines placed by an alternate utility, which is interpolated across the mesh via a graph-based algorithm. A key difficulty is the mapping of the voxel surface to an approximation of the mesh surface: our approach is described below.

2.2.1 Surface Fitting

The muscle-wrapping algorithm of Grosse *et al* essentially requires that good estimates for curvature and the shortest geodesic path can be obtained on a given mesh surface. Unfortunately, for our voxel models, the surface is usually very coarse and assessments for these properties cannot be performed directly.

The first step is therefore to obtain a reasonable approximation of the voxel surface, for which we make use of ParaView to visualize the surface fit. As with the definition of point-based BCs, we rely upon the user to define the area of muscle attachment, which is then extracted.

2.2.2 Minimising Fitting Error

In order to minimise the error in fitting, these data are then transformed by Principal Component Analysis (PCA), so that the axis of least variation is aligned with the z-axis of the PCA transformation, and becomes the axis over which the fit is performed.

One possibility is to use splines to perform this fit, but after experimentation with various open-source libraries (e.g. Netlib's surfit[1], and alglib[2]), it was realized that the

[1] http://www.netlib.org/dierckx/surfit.f
[2] http://www.alglib.net

mesh approximation was still far too coarse. Another problem with these spline-fitting algorithms is that the fit must be retrieved on a regular grid, which would require further sampling to relate properties (curvature etc.) back to the original grid.

An alternative approach, initially tested using the `fields` package in R, was to use Kriging. This method has been developed by geologists using irregularly spaced samples to interpolate across any given region, and is therefore well suited to the problem of irregular input samples. After further work, the `gstat` package in R was tried and performed well for this purpose. The standalone gstat C library[3] has now been integrated into the VOX-FE plugin and Eigen[4] is used to obtain the PCA transform.

2.2.3 Estimates for Normals and Curvature

ParaView (and the underlying VTK library) has several routines to assist with mesh construction and curvature estimation. It was found, however, that while VTK's `Delaunay2D` filter provides a visually acceptable surface mesh approximation, used as an input to VTK's curvature estimator, the mesh gives poor results.

To improve the curvature estimation, several smoothing operators were tried (such VTK's `vtkSmoothPolyDataFilter`) but with little success. A serious drawback to these methods is that they arbitrarily refine the mesh by removing near coincident points. While this presumably has great merit in terms of smoothing it also makes mapping back to the original points much more difficult.

Instead, a point-based method was adapted from the code given in Doria's submission to the VTK Journal [5]. Since this method does not require an input mesh topology, and allows the radius of operation to be easily changed (giving a smoothing effect), it was much more readily integrated into the plugin.

A modification was added to the curvature estimator. Doria's code computed the average distance from the best fit plane to points within a search sphere of given radius. This value was then normalized to lie in the interval $0-1$. Instead, we suppose that the average deviation, h, from the best-fit plane occurs at l (which is some fraction of the

[3] http://www.gstat.org
[4] http://eigen.tuxfamily.org/

search radius L). A best-fit circle of radius r can then be found passing through this point and the centre of the search sphere. We define the angle a as:

$$a = tan^{-1}(h/l)$$

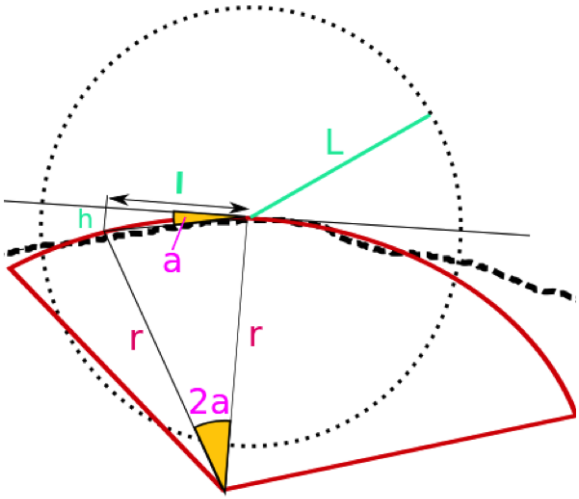


Figure 2: The modified best-fit circle algorithm, showing how the angle a and the radius r are defined.

By simple trigonometry, we know that the angle subtended at the centre of the fitted circle is $2a$. The search sphere (dotted in Figure 2) maps to a circle on the best-fit plane. Assuming that the points within the search sphere are evenly distributed, then half of the points will fall within a radius $L/\sqrt{2}$ of the mapped circle (and half without). Thus, setting l to $L/\sqrt{2}$, we can estimate the radius r as:

$$r = (L/\sqrt{2})cos^2(a)/sin(2a)$$

This produces a better estimate of curvature ($= 1/r$) then the original code. Testing the approximation using `vtkSphereSource` meshes generated in ParaView typically yields values within 5% of the true radius.

The code for estimating normals works as given, provided that an outward orientation

can be supplied. This can be readily computed from the PCA transformed point data, comparing data from the edge and central regions of the transformed space.

2.2.4 Computing the shortest path

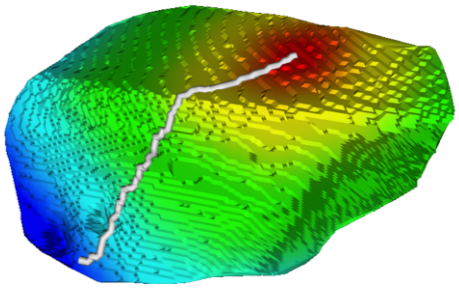


Figure 3: Example of a computed shortest path between two points on the surface of a voxel mesh.

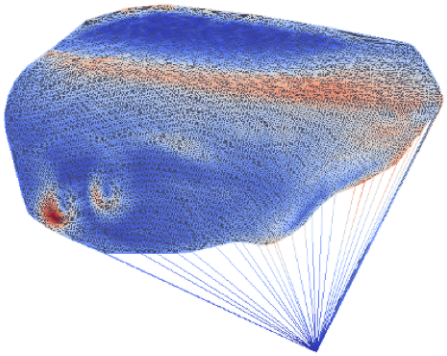


Figure 4: Multiple paths towards a single point, showing how more paths are concentrated on regions of high curvature.

VTK provides the `vtkDjikstraShortestPath` class to apply the associated shortest path algorithm on triangular mesh data. Using the Delaunay (2D) algorithm to compute the mesh topology, it is therefore possible to compute paths from each selected point towards a user-supplied target point (see Figure 3).

A problem occurs, however, where the muscle topology is constructed by connecting a selected area to a single target point. A convex edge is established which minimises the number of triangles towards the target leading to an edge effect. This causes the paths to concentrate along a few strands towards the target point, rather than along possible grooves in the underlying bone (see Figure 4; note the surface colour mapping is by curvature).

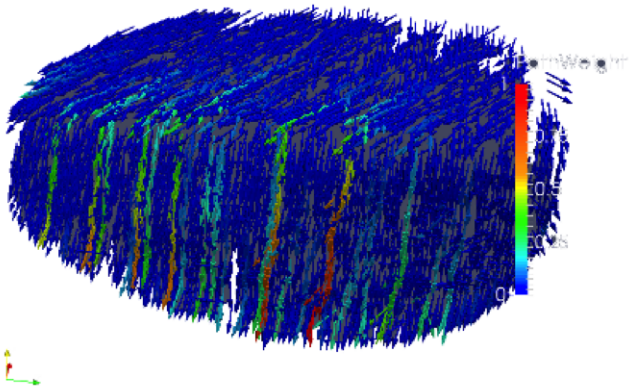


Figure 5: Distribution of strands along a path.

To correct for this, an alternative graph algorithm was devised, using the Boost graph library[5]. The Delaunay mesh can be interrogated to find the mesh triangles associated with the target point and neighbouring points are also recovered. The user is then requested to set a maximum edge side, say λ, so that all triangles with sides larger than λ can be removed and/or remodelled. This results in a much more even strand distribution, although there is still an accumulation along particular paths, see Figure 5.

2.2.5 Tangents

The Boost graph algorithm for the Dijkstra shortest path allows paths to be traced from every point to the target. Each point therefore has an associated path except where a path terminates. If this path segment is projected into the plane defined by the normal, then

[5]http://www.boost.org/doc/libs/1_61_0/libs/graph/doc/index.html

we can form the associated tangent. A resultant force can then be computed using the normal, tangent and curvature properties as described in Section 2.2.

2.2.6 Comments

The muscle-wrapping algorithm requires a number of properties to be defined which can be strongly affected by the choice of algorithm and the smoothness of the extracted surface. Time has not permitted the sensitivity of the algorithm to be determined for the factors obtained as described here. A further issue is that correct scaling of the normal (compressive) force must be determined experimentally.

3 The VOX-FE 3.0 solver

3.1 Overview

The VOX-FE solver determines the effect of constraints and forces applied to a bone model. It does this by solving a system of coupled linear equations defined by a Global Stiffness Matrix (GSM), which describes the relationship between the bone elements making up the model and their material properties, and a force vector of any constraints placed upon the bone elements. Each bone element is a cuboid with eight nodes (the corners). When the linear system of equations is successfully solved, the solver outputs the final nodal displacements.

In eCSE01-15 [6], we developed VOX-FE 2.0, in which the legacy VOX-FE solver PARA-BMU was replaced with a PETSc[6]-based solver. The VOX-FE 2.0 solver (solver 2.0 from here on) outperformed the legacy solver in several ways: by solving larger models (> 20M elements) with an arbitrary number of material types, by scaling to higher core counts (> 256 cores), by being more memory efficient in using more cores per node and by offering a wider range of solution algorithms through the PETSc library.

In solver 2.0, every MPI process loads the whole bone model. This keeps MPI communication between processes to a minimum. However, it increases the memory cost per

[6]Portable, Extensible Toolkit for Scientific Computation [7]

process as the model size increases. Construction of the GSM is divided evenly amongst processes, so that process p is responsible for the p^{th} set of consecutive GSM rows. The MPI communication involved in constructing the GSM and solving the linear system is handled by the PETSc library. For models with a dense, contiguous structure, e.g. a cylinder beam, dividing work between processes in this simple way works well. However, realistic bone models are sparse and have highly irregular complex structure which results in load imbalance across processes.

In this project we have addressed the load imbalance and memory efficiency issues of solver 2.0. We use ParMETIS[7], an efficient, scalable graph partitioning library, to partition the model between processes so that work is balanced evenly across processes. We also introduce an improved method for model loading, whereby each process reads and owns a subset of the entire model. Thus the memory footprint of each process is reduced. These new developments are built into the VOX-FE 3.0 solver (solver 3.0 from here on) and are discussed in Section 3.2. The new solver's performance results, discussed in section 3.4, confirm that with good load balancing, it outperforms solver 2.0 for realistic bone models. Furthermore, the reduced per-process memory usage means that solver 3.0 is able to solve models an order of magnitude larger than the largest models solved with solver 2.0, and on a larger number of cores than was possible with solver 2.0.

3.2 Algorithm

The VOX-FE solver is written in C++ and parallelised with MPI. Figure 6 shows the main classes of the solver. The red highlighted methods are the key developments in solver 3.0.

The overall algorithm for partitioning the model optimally is described below:

1. Model elements are evenly divided across processes.

2. Each process has a set of local elements, `ElemS`, a set of corresponding local nodes, `NodeS`, a set of non-local or "ghost" (i.e. belonging to another process) elements, `GElemS`, and a set of corresponding non-local nodes, `GNodeS`. `GElemS` and `GNodeS`

[7] http://glaros.dtc.umn.edu/gkhome/views/metis

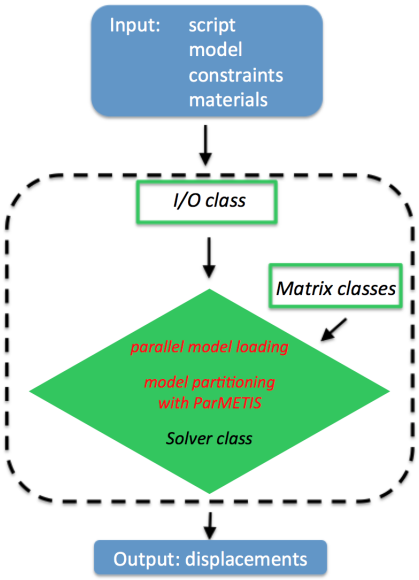


Figure 6: Schematic of solver 3.0: new developments described in this report are highlighted in red.

contain elements and nodes that neighbour the elements and nodes in `ElemS` and `NodeS` respectively. Note that it is necessary for every process to know all the neighbours of the nodes it owns (in `NodeS`) in order to construct its part of the GSM: a GSM row corresponds to a node, and each non-zero GSM column corresponds to that node's neighbour node(s).

3. Using the ParMETIS `Mesh2Dual` method, the element neighbours of each element in ElemS are found, which in turn determines the nodal neighbours of each node in NodeS.

4. Element and nodal neighbour information is distributed to all processes using `MPI_Alltoall`.

5. All nodes are renumbered so that each process owns a consecutively numbered set of nodes. This is necessary for using ParMETIS methods.

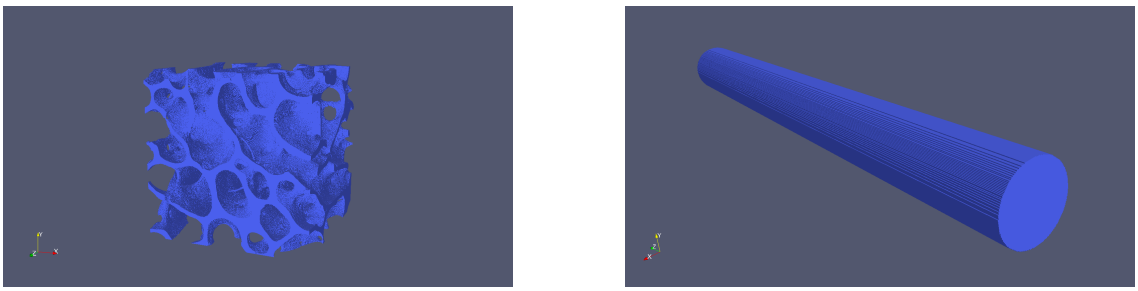
6. Using the ParMETIS `PartGeomKway` method, the optimal distribution of nodes across processes, i.e. the partition vector, is found.

7. Repartitioned element and nodal information is once again distributed to all processes using `MPI_Alltoall`.

8. All nodes are once again renumbered. This is necessary for the construction of the GSM, which requires that each node set is consecutively numbered.

Once the model has been partitioned, forces and constraints are loaded. The GSM is then constructed and the linear system is solved using PETSc. In solver 2.0, the final nodal displacements are sent to a master process that writes the displacements to a file in Cartesian order. In solver 3.0, to output the displacement data in order, it proved simplest to have each process write its set of nodal displacements to a separate file and then to collate the data in a single file using an external post-processing script. This is not an ideal solution but proved to be the most straightforward without requiring any MPI-IO methods or sending all the data to a master process which would cause memory issues when solving models with hundreds of millions of elements.

3.3 Test models



(a) Horse trabeculae cube model (200M). (b) Dense cylinder model (8M).

Figure 7: Test models with total number of elements indicated in brackets.

We tested the performance of the new solver using three different models: (a) a 200M element (sparse) horse bone cube model, (b) an 8M element (dense) cylinder model and (c) a 25M element (sparse) trabeculae cube model. Figure 7 shows models (a) and (b); model (c) is similar in structure to model (a). We chose these models to test the solver's

performance on different bone geometries (idealised and realistic) and different sized models.

3.4 Performance results

We ran all three models on solver 3.0. We also ran the cylinder model and the sparse trabecular model on solver 2.0; the horse bone model was too large for the old solver. Additionally, to maximise the per-core memory for the 25M element trabeculae model on solver 2.0, ARCHER nodes were underpopulated so that only one or two cores were used per node. Finally, we excluded the time taken to output nodal displacements from the performance data to make comparisons between the two solvers fairer (since in solver 3.0 this involves an external post-processing step to reorder the displacements).

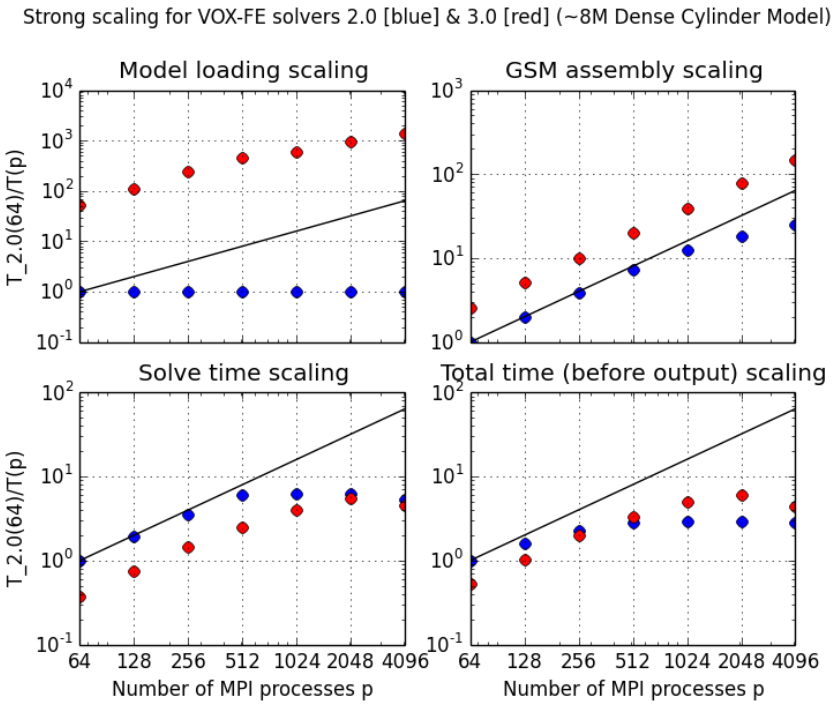


Figure 8: Strong scaling of solver 2.0 and solver 3.0 using the dense cylinder model (8M elements). $T(p)$ is the time taken by either solver on p processes and $T\_2.0(64)$ is the baseline time taken by solver 2.0 on 64 processes.

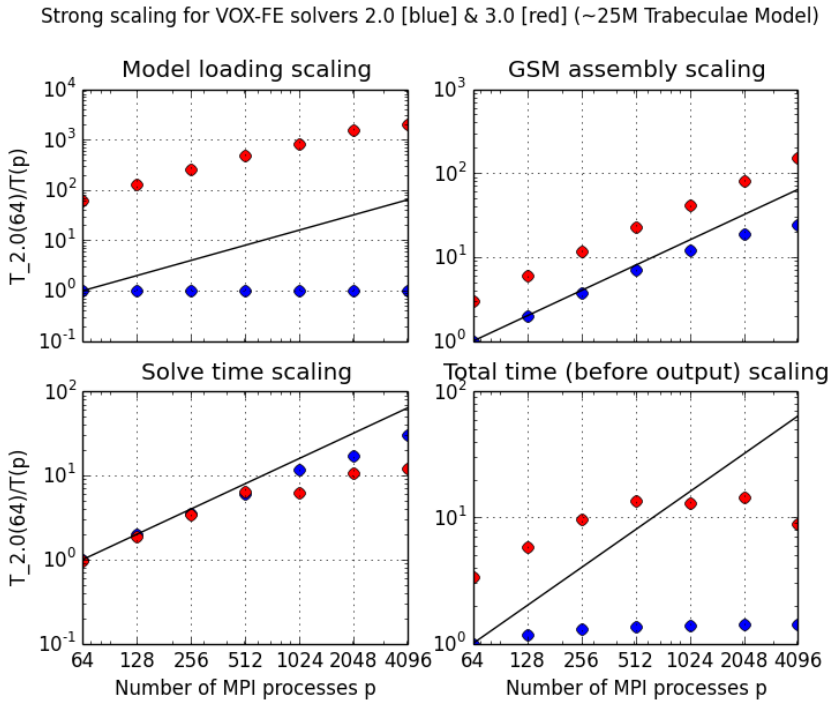


Figure 9: Strong scaling of solver 2.0 and solver 3.0 using the sparse trabecular model (25M elements). $T(p)$ is the time taken by either solver on p processes and $T\_2.0(64)$ is the baseline time taken by solver 2.0 on 64 processes.

Figures 8 and 9 show the strong scaling performance of solver 3.0 relative to solver 2.0. We see that the performance of the solvers varies between the two types of models. For model loading and constructing the GSM, solver 3.0 consistently shows super-linear scaling and outperforms solver 2.0. However, in solver 3.0, the time to solve the linear system scales sub-linearly for the cylinder model and drops below ideal scaling above 512 MPI processes for the trabeculae model. The overall solver time for both solvers (excluding the time to output nodal displacements) scales sub-linearly in the case of the cylinder model, with solver 3.0 outperforming solver 2.0 on 512 or more processes. Solver 3.0 consistently outperforms solver 2.0 in the case of the trabeculae model, and shows super-linear scaling for up to 512 processes, before scaling sublinearly.

Figure 10 shows the strong scaling performance of solver 3.0 using the 200M element horse bone model. The solver shows close to ideal scaling up to 4096 processes.

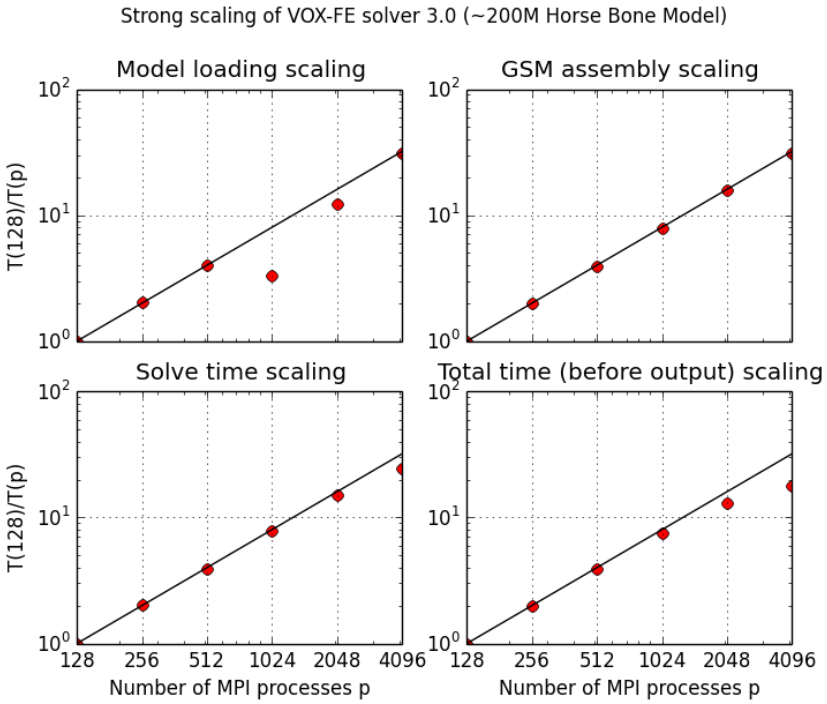


Figure 10: Strong scaling of solver 3.0 using the horse bone model (200M elements). $T(p)$ is the time taken by the solver on p processes.

In order to run million-element models on solver 2.0, we had to undersubscribe each node so that there was enough memory per core. Figure 11 shows the total number of cores (as opposed to the total number of MPI processes) against the total solver time before output. Solver 3.0 is consistently faster than solver 2.0 whilst also being able to requiring a smaller number of cores than is possible on solver 2.0.

3.5 Discussion

We have developed the next version of the VOX-FE solver, solver 3.0, by improving the overall load balance and the memory usage per core of previous solver. The performance results show that these developments have led to the solver now being able to handle models that are an order of magnitude larger than previously possible, and with good scaling on thousands of cores. Furthermore, the memory usage per core has been signifi-

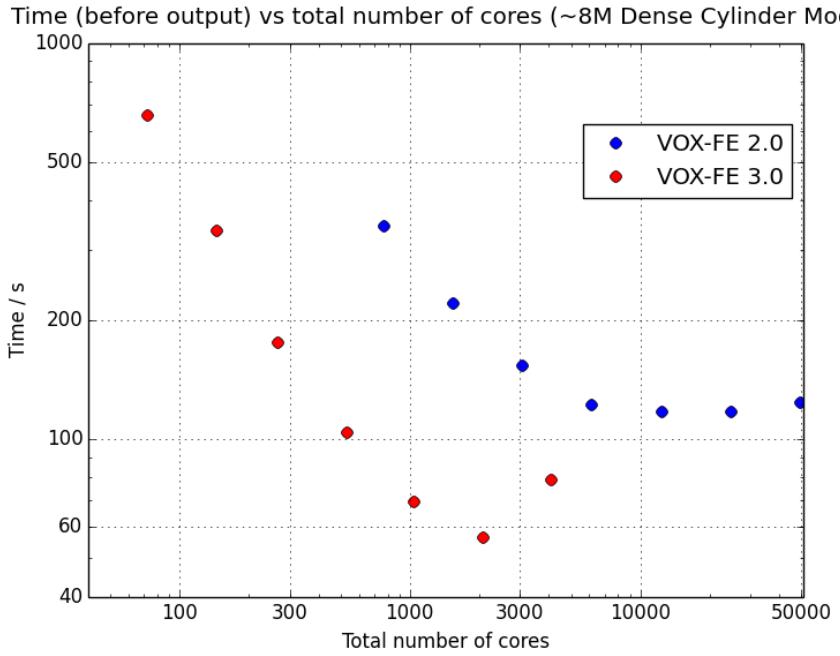


Figure 11: Total time (before output) against total number of cores (8M element cylinder model).

cantly reduced so that fewer nodes are needed to run the same model sizes on new solver compared to the old solver. This reduces computing costs and is more energy efficient. Solver 3.0 performs well for realistic bone geometries and is overall faster than solver 2.0 when considering the total number of cores used. For smaller, less realistic models such as the dense cylinder model, ParMETIS may be unable to improve upon the simple model decomposition used in solver 2.0. It is not yet clear why solver 2.0 mostly outperforms solver 3.0 in the solve phase. This may be in part due to the model size. Possible avenues for future work could include exploring why this might be and also how MPI-IO could be used to efficiently output the nodal displacement data to one file.

4 Adaptive Remodelling

A adaptive remodelling harness was added to VOX-FE in the 2.0 release, and is fully described in Section 4 of [6]. The main drawback of the initial remodeller was that

selecting the upper and lower strain thresholds for applying remodelling was difficult, and the best values would vary from iteration to iteration as the structure adapted to the imposed load and constraints. Two improvements have been implemented in this project:

- A formal command-line interface based on the Boost program options module, which can also be configured by a simple text file; and
- An adaptive histogram-based scheme for the selection of thresholds.

4.1 Configuring the execution parameters

Running the help option of the graph control utility now provides an overview of the options:

```
>./voxfeRemodelControl --help


Generic options:
  -v [ --version ]                print version string
  -h [ --help ]                   produce help message
  -c [ --config ] arg (=remodel.cfg) name of configuration file.

Configuration options:
  -i [ --script ] arg       VOX-FE script file
  -m [ --materials ] arg    Binary file mapping ordered GlobalElementID to
                            MaterialID
  -g [ --graph ] arg        Graph of node connectivity in element order
  -a [ --adaptive ] arg (=Y)   Use adaptive or hard threshold limits for adding
                            removing voxels. Enter Y/N
  -l [ --lower ] arg        Lower threshold limit (enter value between
                            0.0-1.0 eg. 0.005)
  -u [ --upper ] arg        Upper threshold limit (enter value between
```

```
                              0.0-1.0 eg. 0.95)
  -k [ --lazycentre ] arg     The lazy region (ie no further remodelling)
                              centre, denoted K in the literature, typically
                              0.0025-0.005
  -s [ --lazywidth ] arg      The lazy region (ie no further remodelling)
                              centre, denoted S in the literature, typically
                              0.3
  -z [ --lazypop ] arg        When this fraction of the SED data is within the
                              lazy region, remodelling will cease. Enter value
                              between 0.0-1.0
  -d [ --displ ] arg          The name of the solver generated displacement
                              file
  -p [ --petsc ] arg (=Y)     Use the the PETSC-based solver, or the older
                              solver (PARA_BMU). Enter Y/N
  -f [ --fullconnec ] arg (=N) Use full/26-neighbour connectivity to remove
                              disconnected elements (aka islands), or the less
                              drastic 6-neighbour connectivity. Enter Y for
                              26- or N for 6-connectivity
```

Note that whilst all the options presented can be given in a configuration file, particular options can be overwritten on the command line using the given tags. Thus to remind the user, the actual chosen command line options are stored in a generated file: `Remodelling.args.txt`.

4.2 Adaptive Remodelling

Using fixed thresholds has several drawbacks: finding good initial thresholds requires many runs to determine and often those starting values will give poor results after a handful of iterations. One factor, however, made the search for a more automatic threshold more awkward - namely that the range of Strain Energy Density (SED) values generated

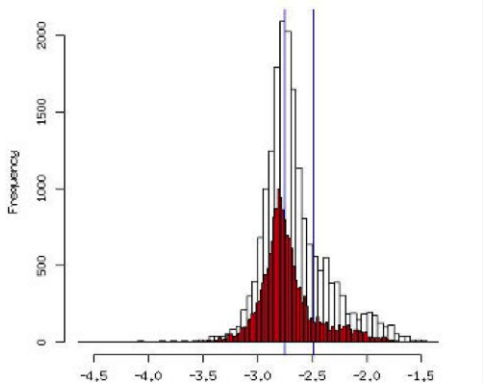


Figure 12: Example strain histogram of the number of voxels with a given SED, showing the effect of log scaling.

at the surface of models is neither normal nor half-normal. In fact, plotting histograms shows that the majority of the data lie in the first or the first few bins.

The solution adopted in VOX-FE is to use a log scale for the histogram bins. This shows a much more 'normal' shape, which is more amenable to the selection of upper and lower populations (see Figure 12). If the user selects adaptive thresholding (see Section 4.1), then two other parameters must be provided giving the fraction of the population below which voxels are removed and that fraction of the population above which voxels are added. Since more voxels tend to be added than removed, it is advisable to have a larger removal threshold than the addition threshold e.g. 0.05 and 0.995.

Weinans *et al* [8] and several others have proposed that a 'lazy zone' can be defined in which no further growth should occur. A zone (centre [-k] and width [-s]) can be set in the configuration file which will terminate the remodelling process if a sufficient percentage of the population should be found to lie within this zone.

4.3 Discussion

The process outlined above could be streamlined, particularly with regard to (re-)reading input files. Ideally, the graph control would remain resident throughout, checking periodically for the generation of appropriate files, perhaps, via notifications from the solver

through a socket interface. The current scheme, however, has the advantage of simplicity and ease of debugging as all intermediate files are available.

Although use of the connected components algorithm is advised to remove separated bone 'islands', the solver has so far proved to be robust to models with small numbers of disconnected elements. Possibly this is because such regions are generally well away from regions of high strain, so that the impact is slight. The more computationally demanding 26-neighbour check is used currently, since it should be noted that cheaper 6-face connectivity option removes all weakly linked (i.e. node or edge only) element connections.

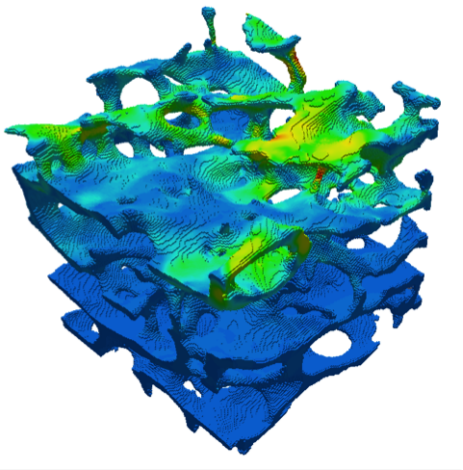


Figure 13: Strain energy density visualisation of a trabecular bone model. Yellow and red indicate regions of high strain, where new voxels will be added prior to the next remodelling iteration.

Due to some memory issues with the 2.0 solver, only models of around 2 million elements have been tested within the above scheme. The wall-clock time for solving the model, shown in Figure 13, lies between 280-370 seconds (on 16 cores, the variation arising from the fact that the model geometry and number of elements varies during each remodelling cycle), with the remodelling step itself requiring less than 10 seconds.

Setting the 'lazy zone' to terminate the remodelling process, which should lie around 0.0025-0.0052 J/g (±30%) according to the literature, has not proved to be very workable so far. It is unclear if these values may be too restrictive or whether these values may

vary for different bone types. Given that the zone is now highly configurable in VOX-FE, however, it should be possible, with further experimentation, to define a likely termination zones.

5 Conclusion

In summary, we have developed several new features in VOX-FE - Muscle wrapping boundary conditions and a flexible intensity mapping model are now implemented in the GUI, the solver has improved scaling for load-imbalanced geometries and parallel input reading, allowing more simulations of up to 200M elements at least, and the existing remodelling tools have been extended to automate the process. This code will shortly be released as VOX-FE 3.0, and is already freely available from the VOX-FE project SVN (on https://sourceforge.net/projects/vox-fe/). This new functionality makes VOX-FE more widely applicable, and as well as applications within the Hull group to dragonfly and mammalian skulls, there has already been interest in applying VOX-FE in other fields including fusion materials and soft matter.

Acknowledgements

This work was funded under the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk)

An earlier version of this document was reviewed by Mark Bull and Kevin Stratford.

References

[1] DR Carter and WC Hayes. The compressive behavior of bone as a two-phase porous structure. *J Bone Joint Surg Am.*, 7:954–962, 1977.

[2] Benedikt Helgason, Egon Perilli, Enrico Schileo, Fulvia Taddei, Sigurur Brynjlfsson, and Marco Viceconti. Mathematical relationships between bone density and mechanical properties: A literature review. *Clinical Biomechanics*, 23(2):135 – 146, 2008.

ISSN 0268-0033. doi: http://dx.doi.org/10.1016/j.clinbiomech.2007.08.024. URL http://www.sciencedirect.com/science/article/pii/S0268003307001866.

[3] Ian R. Grosse, Elizabeth R. Dumont, Chris Coletta, and Alex Tolleson. Techniques for modeling muscle-induced forces in finite element models of skeletal structures. *The Anatomical Record: Advances in Integrative Anatomy and Evolutionary Biology*, 290(9):1069–1088, 2007. ISSN 1932-8494. doi: 10.1002/ar.20568. URL http://dx.doi.org/10.1002/ar.20568.

[4] Jia Liu, Junfen Shi, Laura C Fitton, Roger Phillips, Paul OHiggins, and Michael J Fagan. The application of muscle wrapping to voxel-based finite element models of skeletal structures. *Biomechanics and modeling in mechanobiology*, 11(1-2):35–47, 2012.

[5] David Doria. Point Set Processing for VTK - Outlier Removal, Curvature Estimation, Normal Estimation, Normal Orientation. *The VTK Journal*, 2010. URL http://www.vtkjournal.org/browse/publication/708.

[6] N. Banglawala, I. Bethune, R. Holbrey, and M. J. Fagan. Voxel-based finite element modelling with VOX-FE2. *ARCHER Whitepaper*, May 2015.

[7] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. PETSc Web page. http://www.mcs.anl.gov/petsc, 2016. URL http://www.mcs.anl.gov/petsc.

[8] Harrie Weinans, Rik Huiskes, and HJ Grootenboer. The behavior of adaptive bone-remodeling simulation models. *Journal of biomechanics*, 25(12):1425–1441, 1992.