MorphoShell: Simple Simulation Software for Shape-Shifting Shells

https://github.com/Daniel-Duffy/MorphoShell
Daniel Duffy, daniellouisduffy[at]gmail[dot]com

1 Introduction

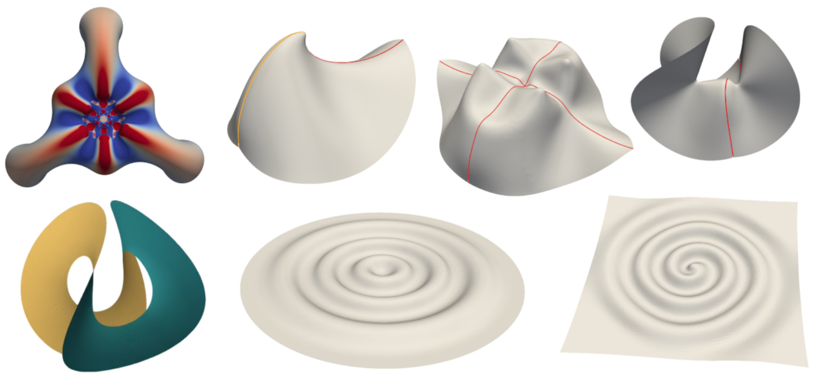
Figure 1: Elastic shell configurations calculated by MorphoShell. Bottom-middle and bottom-right arise from squashing passive cones [1]; the rest [2–4] show active 'shape-programmed' shells that morphed from flat to reach these states.

MorphoShell simulates elastic shells as they deform. It calculates forces from an elastic stretch+bend energy, and evolves a triangulated mesh accordingly, via damped Newtonian dynamics. Such dynamics are more directly physical than those in many other codes, which is good for both understanding your results and making nice videos of them. It also makes it much easier to capture mechanical instabilities. While the code can handle simulations of traditional passive shells perfectly well, it is especially aimed at active 'shape-programmed' shells, which start flat but then develop a preferred metric and/or curvature as they are 'activated'; a growing leaf being a familiar example. A less familiar example is a liquid crystal elastomer sheet, which can be experimentally 'programmed' with an azimuthal pattern of contraction, and will then morph from flat into a cone upon heating (Fig. 2), lifting thousands of times its own weight in the process [5, 6].

MorphoShell first appeared alongside a 2020 paper (*Defective Nematogenesis* [2]) by John Biggins and myself, but the version maintained at `https://github.com/Daniel-Duffy/MorphoShell` is more up-to-date. It's written in

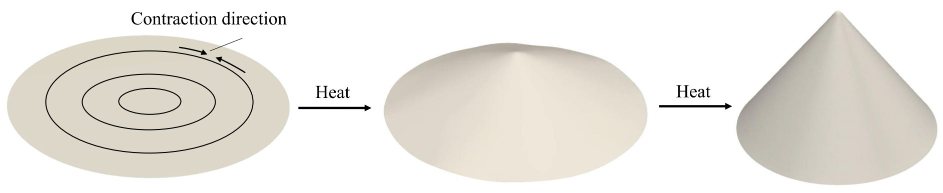

Figure 2

c++, but is designed to be pretty friendly to use and modify, even for inexperienced coders. John and I wrote it because we couldn't find any good, easy-to-use software for shape-programmed shells. The philosophy is to be as simple as possible while getting the job done; the code's not supposed to be a black box, nor extremely general, nor impressively fast, nor particularly feature rich. It's served us extremely well in our research [1–4, 7–9] by being well-suited to the task at hand, easy to understand, and straightforward to modify. The target audience is broad, but I'm particularly aiming at physicists who may be intimidated by the idea of running simulations themselves, or just don't want to spend loads of time coding. You should be able to use MorphoShell even if you have almost no experience writing or running code.

My c++ source files are distributed at the Github link above under the Cambridge Academic Software License (CASL), which very roughly means that if you're an academic the license is a lot like GPL, while if you're at a company or in any commercial setting, you'll need to come to another arrangement with me. For meshing, I use a Python implementation of the DistMesh algorithm [10]; I provide the necessary scripts in the Github repo under a GPL license. For visualisation I recommend ParaView, which is excellent and free.

This handbook gives detailed instructions on how the code works and how to use it, but please also see this instructional video: `https://danielduffy.org`: COMING SOON! Examples: LCE cone, some other LCE example, a programmed bend example, an example where only bbar is dialled (to 0), a final example squashing the LCE cone.

2 How MorphoShell works

Let's start with geometry (for a more complete version see my thesis at `https://doi.org/10.17863/CAM.105659`). Here and henceforth we will use the Einstein summation convention. Greek indices will run over {1, 2}, Latin indices will run

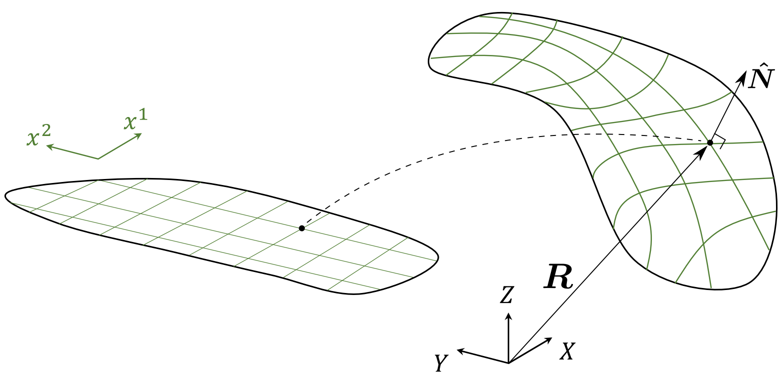

Figure 3: Left: A reference configuration of a shell, flat in this case, with coordinates x^α inscribed. Right: The realised shell described by $\boldsymbol{R}(x^\alpha)$.

over $\{1, 2, 3\}$, and $\partial_\alpha \equiv \partial/\partial x^\alpha$. We define fixed a reference state/configuration for a shell's 2D midsurface, and inscribe it with some coordinates x^1, x^2. These coordinates are best thought of as labels for 'material points': each tiny physical piece of the midsurface is assigned a unique label, which is just its reference-state coordinates (one could imagine these being physically painted onto each infinitesimal element of midsurface). Any realised/deformed state of the midsurface, can be described by specifying the location in 3D of each material point. We will be considering only Euclidean 3D space, so we can use Cartesian 3D coordinates and interpret them as components of a position vector $\boldsymbol{R} = (X, Y, Z)$. We thus describe any realised midsurface with a vector function of the reference coordinates: $\boldsymbol{R}(x^\alpha)$. See Fig. 3. We then have the unit normal vector field

$$\hat{\boldsymbol{N}} \equiv \frac{\partial_1 \boldsymbol{R} \times \partial_2 \boldsymbol{R}}{|\partial_1 \boldsymbol{R} \times \partial_2 \boldsymbol{R}|}, \tag{1}$$

the metric tensor $a_{\alpha\beta} = \partial_\alpha \boldsymbol{R} \cdot \partial_\beta \boldsymbol{R}$, and the 'second fundamental form' or 'curvature tensor'[1] $b_{\alpha\beta} = -\hat{\boldsymbol{N}} \cdot \partial_\alpha \partial_\beta \boldsymbol{R}$. The symmetric tensor fields a and b completely describe the realised midsurface up to rigid body motions. (Although they describe the geometry of the realised midsurface, we can take them to 'live' on the reference midsurface.) Defining $[a]$ and $[b]$ to be the respective matrices of downstairs components, we can form the shape operator matrix $S = [a]^{-1}[b]$. The eigenvalues of S are the principal curvatures, and the mean and Gauss curvatures are given by $H = \mathrm{tr}(S)/2$ and $K = \det(S)$ respectively.

Now some mechanics: The shells we consider are hyperelastic [11]: their elastic

[1]Warning: much of the terminology commonly draped around this tensor is an outdated mess.

energy $E = \int W(\boldsymbol{R}(x^\alpha))\mathrm{d}A$, where W is some energy density and integration is over the reference area. The standard W is often attributed to Koiter [12], but has also been re-derived in the context of shape programming [13]. It takes the form

$$W = \frac{Y}{2(1-\nu^2)}\,Q\Big([\bar{a}]^{-1}([a]-[\bar{a}])\Big) + \frac{D}{2}\,Q\Big([\bar{a}]^{-1}([b]-[\bar{b}])\Big)\,, \tag{2}$$

where the quadratic operator $Q(\tau) \equiv \nu\mathrm{tr}(\tau)^2 + (1-\nu)\mathrm{tr}(\tau\cdot\tau)$. Here ν is Poisson ratio, while $Y \equiv \mathtt{E}t$ and $D \equiv \mathtt{E}t^3/(12(1-\nu^2))$ are respectively the stretching modulus and bending modulus ('flexural rigidity') for realised Young's modulus $\mathtt{E}$ and realised thickness t. The thickness t and the 'programmed' tensors $[\bar{a}]$ and $[\bar{b}]$ determine the energetically preferred geometry, and in shape programming they vary with the activation stimulus, e.g. temperature.

We use a slightly modified W, containing an extra 'bend stiffening' term, i.e. a term that effectively increases the bending modulus as curvature increases, which one physically expects once curvatures reach scales $\sim 1/t$ (representing the breakdown of shell theory). Denoting the bend term in (2) by W_b, we instead take the bend contribution to W to be $W_b' = W_b + W_b^2/(\gamma\mu t)$ where μ is the realised shear modulus and γ is a dimensionless number chosen in the settings file. The factor μt is exactly the scale of W_b when curvatures are $\sim 1/t$. In almost all simulations, γ can be chosen to be some huge number, completely removing the stiffening effect. However a moderate value (e.g. $\gamma \sim 1$) can be desired for some simulations. For example, shape-programming a steep cone via azimuthal contraction, such a γ can prevent the tip (whose curvatures $\sim 1/t$) from buckling into something pathological.

In a shape programming problem, we want to minimise the energy E, for some given $[\bar{a}]$, $[\bar{b}]$, and t. This happens naturally in damped Newtonian dynamics, which is after all the way real physical shells reach their energy minima! Furthermore, the Newtonian dynamics of a system with energy E may well be of interest in their own right. Thus motivated, MorphoShell approximates those dynamics, implementing a rather simple algorithm: We discretise, approximating the energy E by a function of finitely many degrees of freedom (mesh node positions). Then, at each time step, the elastic forces on the degrees of freedom are just minus the gradient of the discretised energy, which is taken analytically. Next we add simple linear viscous damping, i.e. damping force $\propto$ speed. Finally, we advance the degrees of freedom accordingly using Semi-Implicit Euler time integration.

These dynamics run in a loop, while $\bar{a}$ and $\bar{b}$ are gradually 'dialled' dynamically from values matching the reference state ($[\bar{a}] = \mathrm{diag}(1, 1)$, $[\bar{b}] = 0$) to their final target values (which MorphoShell takes as input along with the reference mesh). Then the dynamics continue removing kinetic energy until equilibrium is 'reached', which we take to be when all node speeds and forces drop below two respective small tolerances.

Now I'll explain the energy discretisation, which amounts to a finite-element approach for stretching and a finite-difference approach for bending. The deform-

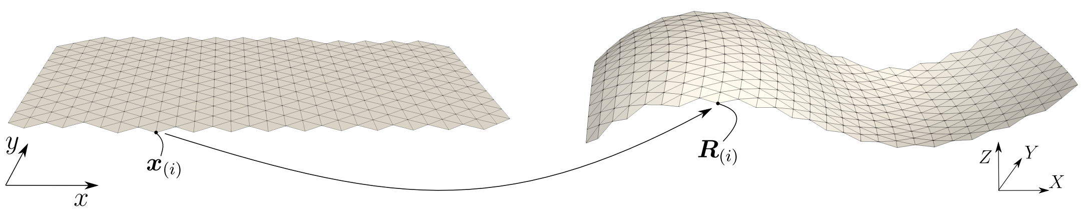

Figure 4: The reference and deformed states of a small piece of mesh.

ing shell is represented by an evolving unstructured triangulated mesh (which I generate using a Python implementation of DistMesh [10]). Mirroring continuum theory, each node has a fixed 2D position vector $\boldsymbol{x}_{(i)}$ in the initial/reference state, and a dynamical 3D position vector $\boldsymbol{R}_{(i)}$ in the realised/deformed/'current' state (Fig. 4).

The realised metric for each triangle, $[a]$, is simply estimated from the unique linear deformation that maps the triangle's reference-state vertex positions onto their current-state positions: a standard constant-strain finite-element approach. Algebraically, for each triangle T we define the 3×2 'deformation gradient' matrix F via $F^i{}_\alpha = \partial_\alpha R^i$, and require that F maps two of the triangle's reference-state side vectors onto their current-state values. This quickly yields a simple expression for F in terms of reference and current side vectors of T, and then the downstairs components $[a] = F^{\mathrm{T}}F$.

We use a 'patch-fitting' approach to estimate the realised curvature tensor $[b]$, as is common in computer graphics. In contrast to some others [14], our estimate contains only node-position degrees of freedom. For a given triangle T, six 'patch nodes' are determined: T's three vertices, and three further nodes selected from nearby triangles. The deformed surface in the vicinity of the triangle is approximated by a 'patch' surface $\boldsymbol{P}(\boldsymbol{x})$ that is a quadratic function of Cartesian reference coordinates (x, y) and interpolates the six patch node positions (Fig. 5).

Algebraically:

$$\boldsymbol{P} = \begin{pmatrix} A_1 & B_1 & C_1 & D_1 & E_1 & F_1 \\ A_2 & B_2 & C_2 & D_2 & E_2 & F_2 \\ A_3 & B_3 & C_3 & D_3 & E_3 & F_3 \end{pmatrix} \begin{pmatrix} 1 \\ \Delta x \\ \Delta y \\ \Delta x^2 \\ \Delta x \Delta y \\ \Delta y^2 \end{pmatrix} \equiv C \begin{pmatrix} 1 \\ \Delta x \\ \Delta y \\ \Delta x^2 \\ \Delta x \Delta y \\ \Delta y^2 \end{pmatrix}, \tag{3}$$

where $\Delta x \equiv x - \mathring{x}$, $(\mathring{x}, \mathring{y})$ are the reference-state Cartesian coordinates of T's centroid, and the 18 elements of C are all constants. Eqn (3) is nothing more than a truncated Taylor expansion for each component of $\boldsymbol{P}(\boldsymbol{x})$, valid as long as the surface's smallest local radius of curvature is significantly larger than the triangle size. Requiring the patch surface $\boldsymbol{P}$ to pass through the current coordinates of the six patch nodes (assembled in a matrix $\tilde{P}$) then provides six constraints that can be used to find the elements of C. Taking, for example, T's patch nodes to have labels $i = 1, 2, 3, 4, 5, 6$:

$$\begin{aligned} MC^{\mathrm{T}} &\equiv \begin{pmatrix} 1 & \Delta x_{(1)} & \Delta y_{(1)} & \Delta x^2_{(1)} & \Delta x_{(1)}\Delta y_{(1)} & \Delta y^2_{(1)} \\ 1 & \Delta x_{(2)} & \Delta y_{(2)} & \Delta x^2_{(2)} & \Delta x_{(2)}\Delta y_{(2)} & \Delta y^2_{(2)} \\ 1 & \Delta x_{(3)} & \Delta y_{(3)} & \Delta x^2_{(3)} & \Delta x_{(3)}\Delta y_{(3)} & \Delta y^2_{(3)} \\ 1 & \Delta x_{(4)} & \Delta y_{(4)} & \Delta x^2_{(4)} & \Delta x_{(4)}\Delta y_{(4)} & \Delta y^2_{(4)} \\ 1 & \Delta x_{(5)} & \Delta y_{(5)} & \Delta x^2_{(5)} & \Delta x_{(5)}\Delta y_{(5)} & \Delta y^2_{(5)} \\ 1 & \Delta x_{(6)} & \Delta y_{(6)} & \Delta x^2_{(6)} & \Delta x_{(6)}\Delta y_{(6)} & \Delta y^2_{(6)} \end{pmatrix} C^{\mathrm{T}} \\ &= \begin{pmatrix} X_{(1)} & Y_{(1)} & Z_{(1)} \\ X_{(2)} & Y_{(2)} & Z_{(2)} \\ X_{(3)} & Y_{(3)} & Z_{(3)} \\ X_{(4)} & Y_{(4)} & Z_{(4)} \\ X_{(5)} & Y_{(5)} & Z_{(5)} \\ X_{(6)} & Y_{(6)} & Z_{(6)} \end{pmatrix} \equiv \tilde{P}, \end{aligned} \tag{4}$$

where for example $Z_{(5)}$ is the current-state Z-coordinate of this triangle's fifth patch node. Thus

$$C = \tilde{P}^{\mathrm{T}} M^{-\mathrm{T}}. \tag{5}$$

Note that $M^{-\mathrm{T}}$ is a reference-state quantity that can be computed just once and re-used repeatedly, while $\tilde{P}$ is updated with the new node positions at each step. The three non-vertex patch nodes were simply chosen to be those closest to T's centroid in the reference state, subject to $M^{-\mathrm{T}}$ being suitably well conditioned. Using $b_{\alpha\beta} = -(\partial_\alpha \partial_\beta \boldsymbol{R}) \cdot \hat{\boldsymbol{N}}$, we combine C with the triangle face normal $\hat{\boldsymbol{N}}$ to

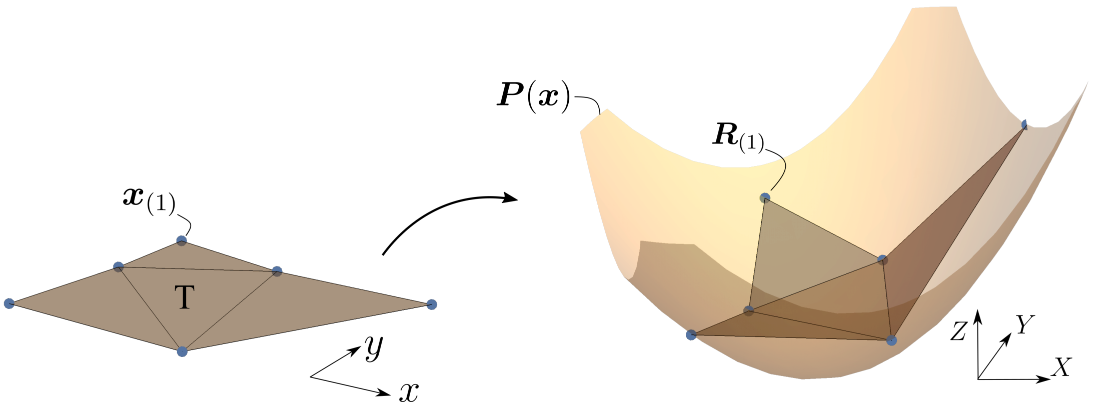

Figure 5: Sketch of the reference and deformed state for a triangle T and its six patch nodes, and the fitted patch surface $\boldsymbol{P}(\boldsymbol{x})$.

give an estimate of $[b]$ for triangle T:

$$[b] = -\sum_{j=1}^{3} \begin{pmatrix} 2\,C_{j4}\;\hat{N}_j & C_{j6}\;\hat{N}_j \\ C_{j6}\;\hat{N}_j & 2\,C_{j5}\;\hat{N}_j \end{pmatrix}. \tag{6}$$

Taking $[a]$ and $[b]$ calculated as described, our continuum shell energy integral becomes a discrete sum over triangles, which is straightforward to differentiate analytically with respect to the simulation degrees of freedom (node positions), given the simplicity of the $[a]$ and $[b]$ approximations. In practice in the code, the $[a]$ and $[b]$ estimates aren't explicitly calculated in the dynamics loop — instead the forces on dofs are expressed in big messy expressions from Mathematica involving quantities like C directly — but what I've described above is what's happening implicitly.

3 Preliminaries to running MorphoShell

Linux or Mac are recommended and assumed by these instructions. The `sudo` commands may require you to enter a password, which should just be whatever password you used to log in to your computer. Don't worry that you don't see the password appearing; just type it and hit enter.

I'm sure if you know what you're doing all the same steps are possible on Windows and the code should run, but I expect it'll be more painful getting there.

3.1 Python stuff

To run the meshing scripts, you'll need to have Python and a bunch of its standard packages. My recommendation: Install Anaconda (Python 3 version), following the online instructions on the Anaconda website. Then open a terminal, type

```
conda update conda
```

on a single line and hit enter (we call this 'running' a command). Then create a new 'environment' by running

```
conda create --name my_shiny_new_env
```

then install some necessary Python packages by activating your environment and then running

```
conda install numpy scipy matplotlib
```

Now every time you want to use Python, you just open a terminal and run

```
conda activate my_shiny_new_env
```

before, say,

```
python3 make_mesh.py
```

3.2 C++ stuff

The compiler I recommend is g++14.

To install that compiler on Linux you should open a terminal and run

```
sudo apt update
```

followed by

```
sudo apt install gcc-14 g++-14
```

Later you'll also need the `make` utility, but on Linux you probably already have it automatically; if not you can get it via `sudo apt`.

On Mac you should first install the widely used Homebrew, then open a terminal and run

```
brew install gcc@14
```

On a Mac you may also need to install `make` if you don't have it already, in which case run

```
brew install make
```

The installer may tell you that `make` has been 'installed as `gmake`'. This just means that the name of the `make` program has been given a different name, so later, when I say things like ‘run `make morphoshell.exe` in a command line’ you will need to type `gmake morphoshell.exe` instead.

You’ll need to install the Eigen library that is commonly used for linear algebra. Here’s the route I recommend on Linux: Go to `eigen.tuxfamily.org`, download the latest stable release from the front page, and unzip it. In my case this produces a folder called `eigen-3.4.0`, which contains some files and roughly 15 other folders. One of those folders should just be called `Eigen`. We’ll now copy-paste the `Eigen` folder into an appropriate location that the compiler will search for it automatically. On Linux, the correct location is usually the `/usr/local/include` folder on your computer[2]. To accomplish the copy-paste, open a terminal, type `sudo cp -R` but don’t hit enter, hit space, drag-and-drop the `Eigen` folder onto the terminal (which will copy its full name into the terminal), then hit space again in the terminal, then type `/usr/local/include`, then hit enter.

On Mac, a similar approach to the above is probably the way to go (though others also exist). As above, download and unzip the latest stable Eigen release. Inside your Downloads folder you should now have a folder called something like `eigen-3.4.0`. Right click on it and select `New Terminal At Folder` to open a terminal. We need to now find a suitable location to copy the Eigen files to, so the compiler can find them. Run `g++-14 -E -x c++ - -v < /dev/null` in the terminal that opened. The output should contain a line `#include <...> search starts here` followed by a list of locations (paths). Highlight one of those locations from the list, and copy it to the clipboard (the first one is probably fine). For me that location is

```
/opt/homebrew/Cellar/gcc/14.2.0_1/bin/../lib/gcc/current/gcc/aarch64-apple-darwin23/14/../../../../../../include/c++/14
```

Then, in the terminal you have open, type `cp -R Eigen`, type a space, then paste the location from the clipboard, then hit enter.

3.3 ParaView

Download and extract the latest stable release from the ParaView website. I just keep the folder on my desktop. The executable (`paraview`) that actually runs ParaView is then inside the subfolder ‘`bin`’ inside the overall ParaView directory you copied. To run ParaView you just double-click on that `paraview` executable.

[2]Other possible gcc ‘include’ directories where you could put Eigen can be listed by running `g++-14 -E -x c++ - -v < /dev/null`

4 Meshing and python

To demonstrate the meshing workflow, open a terminal in the MorphoShell folder, then run

```
conda activate duffyenv
```

(replacing `duffyenv` with the conda environment you set up), and then

```
python3 gauss-codazzi-mainardi.py
```

After the script has finished running you should find that a file called `gauss-codazzi-mainardi` has appeared in the `files_from_python` folder in the MorphoShell folder.

Make sure your triangles are sufficiently small to *resolve out* the phenomena you expect to see by setting `approx_shortest_edge_length` to be smaller than whatever characteristic length scale you think is relevant for your problem. Even you don't know what it is, it's very unlikely to be smaller than the thickness, so the thickness is often a sensible edge length to try. *Most* of the simulations I've done have had a thickness $\sim 1\%$ of the sample size, and have used a mesh with between 20,000 and 80,000 triangles. A good idea when investigating some phenomenon is always to try re-running a simulation with a finer mesh, to check that the answer doesn't change to an extent that you care about; otherwise you should be using finer meshes! In problems that are susceptible to membrane locking you should expect to have to use triangles of size $\sim$ thickness.

Meshes with a uniform triangle size tend to come out very regular from DistMesh (which is great), and these always give better results than irregular meshes, especially for curvature estimates.

5 How to run MorphoShell

To compile MorphoShell, first open a terminal in the MorphoShell directory. You can usually do so by right-clicking inside the directory and selecting 'Open in terminal' or similar. Then run

```
make clean
```

followed by

```
make morphoshell.exe
```

(As mentioned earlier, on a Mac you may have to type `gmake` instead of `make` in the above commands). A bunch of text should then be spat out in the terminal, reporting on the compilation progress. Once it's finished, an executable called `morphoshell.exe` should have appeared in the directory.

Now you can run
```
OMP_PLACES=cores OMP_PROC_BIND=spread OMP_NUM_THREADS=1 ./morphoshell.exe
settings.txt gauss-codazzi-mainardi.vtk
```
That command will run in serial, on a single core. To instead run in parallel using 5 cores, say, change that '`=1`' to '=5'.

6 A brief tour of the source code

The actual MorphoShell code is contained in text (`.cpp` and `.hpp`) files in the `source_files` folder that lives inside the `MorphoShell` folder. These text files are where any desired modifications should be made — I recommend the Visual Studio Code text editor for doing so.

The 'core' of the code is `main.cpp`. Opening it in a text editor, we see that the first function call is `preliminary_setup(...)`. The details of what `preliminary_setup(...)` does can be read in the `preliminary_setup.cpp` file, but basically it creates folders and files on your computer that the code's output will go into.

The next function call is `read_settings(...)`, which reads all the settings values from the settings file, and stores them in a bundle of values called `stuff` (embarrassing but I really couldn't think of a good name!). The value of, say, the `ref_density` setting is stored as `stuff.ref_density` which can then be used and/or modified elsewhere in the code. Various other miscellaneous important quantities (stuff!) other than settings are also stored in `stuff`, e.g. `stuff.timestep`.

Next, the (one or two) input mesh files are read by `read_vtk_data(...)`, with the information being split up and stored in various places like `stuff`, `node_positions` (a matrix with `stuff.num_nodes` rows), `triangles` (a vector, each element of which is a `Triangle_Class`) etc.

Note switch from node positions being stored in `node_positions` to `dofs`.

7 Directory handling

Explain directory handling.

8 Time scales

Explain long time scale.

9 Tags

Explain tags. Easy to add more tags if desired, though remember to also change `num_tri_quants` or `num_node_quants` as appropriate in my `write_VTK` python function.

10 Ansatzing

Remember to explain that you may have to change `dial_factor` in the header of your ansatz vtk file.

Vanilla ansatzing first, then dialling from ansatzes. Combining the latter with remeshing requires gradient descent (useless for full simulations as requires orders of magnitude more timesteps for the slow modes to do anything). Explain this workflow. Mention analogy to Leslie Greengard's k-sweeping idea.

11 How to control the dialling procedure

12 External forces

We can also go beyond free sheets to simulate squashing between two 'glass slides', by adding a simple condition in the algorithm that adds extra vertical stiff-spring forces to any node with a Z coordinate above/below the dynamically evolving upper/lower slide Z-coordinates. These slide coordinates are themselves either directly specified, or evolved via overdamped dynamics based on the forces applied by (and hence on) the slide, the same linear viscous damping, and a sensible chosen slide weight. In simulations with slides, MorphoShell doesn't terminate until you kill it, even if equilibrium is reached as far as the shell is concerned, but that behaviour could easily be changed; you'd simply modify the `if` statement near the end of `main.cpp` just before the `break` command that terminates the dynamics loop.

There is actually a simple friction model for slides in the code, which you can examine in `calc_non_deformation_forces.cpp`, though I usually switch friction off via `slide_friction_coefficient=0` in the settings file.

The effect of gravity on the shell itself is of course also easy to add if desired (I've already played with it; see the relevant commented-out lines in `Stuff_Class.hpp` and `calc_non_deformation_forces.cpp`).

13 How to change the elastic energy functional

14 How to add your own auxiliary output

15 How to add your own settings

Order of settings in settings file does matter and has to be exactly correct.

16 How to add more information to the input or output vtk files

17 Timesteps

In general try running with a timestep prefactor of 1.4 or so first. If you have a very nice mesh and a rather mild deformation you may well get away with this! If it's a long way off working, usually the code will just crash within 30 real human seconds or so of running. Then you can decrease timestep prefactor by a bit and try again. Once you're making it past the first 30 seconds or so, I'd recommend decreasing by another factor of 0.7 or so, because it's good to have some safety, plus triangles' aspect ratios often get worse as the simulation progresses due to anisotropic deformation, and this lowers the stable timestep. I've done many

many simulations using 0.8. I've had to use values as low as 0.2 in the past for extreme deformations.

For a discussion of the physics behind the code's timestep, see Appendix. A.

18 Data output

Explain Cauchy stress format.

We sometimes want to calculate the Gauss curvature of a mesh integrated over some sub-area. To do so we borrow the well-known 'angle deficit' method from discrete differential geometry. This method is motivated by an exact discrete analogue of the Gauss-Bonnet theorem for a triangulated mesh surface S [15]:

$$\sum_{i \in interior} (2\pi - \Xi_i) \; + \sum_{i \in boundary} (\pi - \Xi_i) \; = \; 2\pi\chi(S), \tag{7}$$

where the sums are over the non-boundary and boundary nodes respectively, χ is the mesh's topological Euler characteristic, and

$$\Xi_i = \sum_{\Delta} \beta_{\Delta} \tag{8}$$

is the sum of mesh-triangle interior angles $\{\beta_\Delta\}$ around the node i. Comparing with the continuum Gauss-Bonnet theorem (see Chapter **??**), this provides a natural and widely used definition for the integrated Gauss curvature of an interior node in a triangulated mesh [16]: the 'angle deficit'

$$d_i = 2\pi - \Xi_i. \tag{9}$$

We can also obtain a consistent estimate of local Gauss curvature by simply dividing each node's angle deficit by an area assigned to that node:

$$K_i = \frac{2\pi - \Xi_i}{A_i}. \tag{10}$$

We take $A_i = \sum_\Delta A_\Delta/3$, so that each triangle contributes 1/3 of its area to each vertex. This and similar estimates for K are not locally convergent in full generality [17]. However they are unproblematic for highly regular meshes so we take care to use such meshes whenever (10) is required, and the estimate K_i then performs well. This estimate is used in Fig. **??**.

Elsewhere, no particular consistency with (9) is important, so we instead just use (6) to calculate Gauss and mean curvatures. This is the method used to curvature-color all simulation pictures, unless otherwise noted.

19 Tips

19.1 Workflow

- Whenever anyone says to 'run' something in a terminal, they mean you have to open a terminal, type the relevant command into the terminal, and then hit enter on your keyboard.
- When working with terminals, spaces or other funny characters in file names are very annoying to deal with, so it's better to use only letters and numbers, and replace spaces with underscores, e.g. `my_file.txt` instead of `my file.txt`
- When you want to run `somecommand` in a terminal, you can always instead run

  ```
  somecommand &
  ```

  instead, and then run `exit` to close the terminal while the `somecommand` program will keep running in the background. To kill the `somecommand` program: You can open a new terminal, run `pgrep -a somecommand` to see all instances of `somecommand` that are currently running. You should see an output like `375 somecommand`, where I made the number up. Then run `kill 375`.
- If you right click inside a folder, there's usually a helpful 'open terminal here' option.
- Before you start typing into a terminal, you can push the ↑ key on your keyboard repeatedly to scroll through recent previous terminal commands you've run, which you can then modify before hitting enter. This saves a lot of re-typing long commands!
- You can copy from text terminal using `ctrl+shift+c` instead of the usual `ctrl+c`. Similarly use `ctrl+shift+v` to paste into a terminal. Also, if you single click on a file or a folder, and do `ctrl+c` (as if you were about to paste

it to another location), then you can do `ctrl+shift+v` in terminal to paste the full path to that file instead of typing it, which comes in very handy, e.g. if you want to run using an ansatz file that is stored many directories down and don't want to type the whole path by hand.

- To find and list all instances of some-word in the files contained in a directory, open a terminal in that directory and run the following, including the quotes and the final full stop/period (which means 'the current directory' in terminal language): `grep -nr "some-word" .`
- Here's something useful if your hard drive is filling up and you want to delete some stuff. To find and list the largest files/folders inside the current working directory (recursively), and print out the 10 biggest for you, run the following in a terminal:

  ```
  du -a -h | sort -h -r | head -n 10
  ```
- For exploring folders containing large numbers of files, the Thunar file manager is much faster than the Ubuntu default Nautilus. To run the former, just run `thunar` in a terminal (you may have to install it first).
- To see current usage of cores/cpu/memory, run `top` in a terminal and then press `1`.
- Can check current CPU speeds with `cat /proc/cpuinfo | grep "cpu MHz"`, and compare with the model specifications you see with `cat /proc/cpuinfo | grep "model"`. You may actually find the former are larger than the latter due to various overclocking/turbo abilities your cores might have, which I think is ok. If the former are much smaller than the latter though, that's probably a sign that your cores are being throttled and hence slowing things down, and you'd be better off using few cores to avoid overheating.
- It's good to keep packages up to date by regularly running `sudo apt update` and then `sudo apt full-upgrade`. Similarly, you should regularly do the following: Run `conda update conda` in a plain terminal without first activating any conda environment, then run `conda activate my_shiny_new_env` (or whatever environment you're currently working with), and then run `conda update --all` to update all the packages in that environment.
- PROVIDE FILES: FFMPEG, PIC COMBINER.

- SHOW PARAVIEW TOUR IN TUTORIAL VIDEO, HIGHLIGHTING COLORING OPTIONS, MESH EDGES OR NOT, BACKGROUND COLOR, LIGHTING, BOUNDING RULER FILTER, SELECTING AND EXTRACTING SUBSET OF TRIANGLES OR NODES. SPREADSHEET VIEW. SAVING SCREENSHOT, AND SAVING ANIMATION.

19.2 Coding

- *"Premature optimization is the root of all evil"* – Donald Knuth. Optimised code is usually harder to read and debug, and also takes more time to write, so only optimize things that are worth optimizing (bottlenecks)!

- With OpenMP parallelization, basically if you have different threads trying to *write* to the same location in memory at the same time, things will often break; so *don't* do that. An example would be something like this:

  ```
  double v = 0;
  for(int j=0; j < 100; ++j){
  v += some_func(j);}.
  ```
  Bad! Instead, different threads should only *read* from the same locations in memory. Different threads writing to different parts of the same matrix or vector should be ok (though not a `std::vector<bool>`, which you should never use for anything).

20 Troubleshooting

- If you changed the c++ code, then the first thing to try should *always* be to recompile from scratch by just running `make` before you run `make morphoshell`.

- If there's an error message, read it carefully! These often contain the answer even if they're dense and tedious.

- If you added a function or changed the name of any functions, remember you need to modify/add the function name in the makefile accordingly.

- When it comes to debugging, adding lots of simple print statements is almost always all you need (`std::cout << thing_that_might_help_with_diagnosing << std::endl;`). I've never used a debugger and never needed to; I gather they are quite hard to use correctly and most people misuse them (ditto profilers).

- Check your settings file has the correct format, with all semicolons included etc.
- Try decreasing the `timestep_prefactor` setting. Also check other settings aren't set to extreme or crazy values.
- Examine your mesh. The code tells you which triangle has the worst aspect ratio when it starts up, and if it crashes because a force was too high it tells you which was the dof in question; these are good places to start looking!
- Looking visually at what's going on in Paraview may reveal the problem. Remember you can always set `outputs_per_char_long_time` to a huge number to ensure that you get an output file for every single timestep.
- Think about the physics of your problem: *should* it have sensible dynamics or is it pathological in some way?
- Contact me if you've tried the above and are still stuck!

A Dispersion Relations and Time Steps

A 'toy' model makes it easier to analyse the dispersion relations arising in the code's dynamics: We suppose the energy-minimising sheet is flat, and we perturb around it, assuming the perturbative deformations are translationally invariant in the y-direction, and that the length in that direction is L. We denote by u and w the displacements in the x and z directions respectively. Using these variables, a toy action can be constructed, with kinetic, stretch, and bend energy densities analogous to those used in the code:

$$S = L \int\int \left(\frac{t\rho}{2}(\dot{u}^2 + \dot{w}^2) - \frac{\mu t}{2}(u')^2 - \frac{D}{2}(w'')^2 \right) \mathrm{d}x\, \mathrm{d}\tau$$

where dots represent derivatives with respect to time τ and primes represent derivatives with respect to x, and t, ρ, μ have their usual meanings and dimensions, and $D = \frac{\mu t^3}{6(1-\nu)}$ is the flexural rigidity.

Following the principle of stationary action, we extremise S with respect to u and w to find the equations of motion. Adding in linear viscous damping 'by hand', these equations become

$$t\rho\ddot{u} - \mu t u'' + \eta\dot{u}/t = 0,$$
$$t\rho\ddot{w} + D w'''' + \eta\dot{w}/t = 0$$

where η is a damping constant with dimensions of [force]/([velocity] [length]) (the same as dynamic viscosity, so you can think of η as the viscosity of the surrounding medium). Substituting ansatz solutions $\propto e^{i(kx-\omega t)}$ then give the following dispersion relations for the stretching and bending modes respectively:

$$\omega_s = \frac{-i\eta \pm \sqrt{4\rho\mu t^4 k_s^2 - \eta^2}}{2\rho t^2}, \tag{11}$$
$$\omega_b = \frac{-i\eta \pm \sqrt{4D\rho t^3 k_b^4 - \eta^2}}{2\rho t^2}. \tag{12}$$

The most time-efficient way to run the simulation is to roughly critically damp the longest-wavelength bending mode, which will almost always be the slowest (smallest ω) mode, whilst also usually being the most important mode for large shape changes. This approach leaves all other modes underdamped. Letting the approximate element and sample linear sizes be a and L respectively, the smallest wavevector in the system will be $k \approx 2\pi/L$, so by setting the square root in ω_b to zero, we find that

$$\eta_{\text{crit}} = 2\sqrt{D\rho t^3}\,(2\pi/L)^2 \tag{13}$$

is the desired value of η. In MorphoShell this value is taken as the natural scale of η, and you just tune it via a dimensionless prefactor in the settings file.

With the above value of η, calculating $2\pi/\omega_b$ gives the approximate longest and shortest bending time scales in the simulation as $T_b^{\text{long}} \sim \sqrt{\rho t/D}\,L^2/(2\pi)$ and (for any reasonable $a \ll L$) $T_b^{\text{short}} \sim \sqrt{\rho t/D}\,a^2/(2\pi)$. The corresponding stretching time scales are $T_s^{\text{long}} \sim \sqrt{\rho/\mu}\,L$ and $T_s^{\text{short}} \sim \sqrt{\rho/\mu}\,a$. Usually $t \sim a$ and therefore $T_b^{short} \sim T_s^{short} \ll T_s^{long} \ll T_b^{long}$. Thus, the time step must be chosen to be $\Delta T \sim T^{short}$, and the number of time steps required for the simulation to run should be $\sim T_b^{long}/T^{short} \sim (L/a)^2$ which is often of order 10^4; agreeing pleasingly well with what is observed.

Looking back at our toy equations of motion, we see that the damping force on a little element of area dA moving at velocity $\boldsymbol{v}$ is $-(\eta\boldsymbol{v}/t)\mathrm{d}A$. But the area element has mass $m = \rho t \mathrm{d}A$, so we can write the damping force in a form that we can easily apply to our discrete simulation degrees of freedom: $f = -\eta m \boldsymbol{v}/(\rho t^2)$.

Another simple energy minimization scheme is gradient descent with a fixed step size. We can think of this as the overdamped limit of our damped Newtonian dynamics: deleting the inertial terms from out equations of motion, we find that the velocity is just proportional to the non-damping force (think of pushing a ball through honey). The dispersion relations become[3]

$$\omega_s = \frac{\mu t^2 k^2}{i\eta}\,, \tag{14}$$

$$\omega_b = \frac{Dtkt}{i\eta}\,, \tag{15}$$

yielding the long and short times

$$T_s^{\text{long}} \sim \frac{\eta}{\mu t^2}\frac{L^2}{2\pi}\,, \tag{16}$$

$$T_s^{\text{short}} \sim \frac{\eta}{\mu t^2}\frac{a^2}{2\pi}\,, \tag{17}$$

$$T_b^{\text{long}} \sim \frac{\eta}{Dt}\frac{L^4}{(2\pi)^3}\,, \tag{18}$$

$$T_b^{\text{short}} \sim \frac{\eta}{Dt}\frac{a^4}{(2\pi)^3}\,, \tag{19}$$

which we can use to set the gradient descent timestep ('step size'). We see that now the number of time steps required for a full simulation to run $\sim T_b^{long}/T^{short} \sim (L/a)^4 \sim 10^8$ usually, which is impractically large.[4] Gradient descent isn't completely useless however: it's very good at getting rid of very short-wavelength noise that occurs when you re-mesh the domain, map the new mesh onto a previous output's discrete surface as an initialization, and then begin simulating. This is what I use gradient descent for, as discussed in the main text.

[3]A good question is 'what happened to the other roots of the ω quadratics in this limit?' They get very large and imaginary, corresponding to divergently fast decay of any initial transient in the overdamped limit. By discarding inertia from the get-go, we have essentially already cooked in this infinitely fast decay, so the other roots don't appear.

[4]Intuitively, the problem is that the timestep is set by the fastest modes, so the slow modes take many timesteps to do anything interesting, but unlike Newtonian dynamics they don't get to use their inertia to go quicker — for many timesteps in a row the slow mode velocities will point in the same direction, but those velocities don't compound like they do in Newtonian dynamics; they're 'wasted' instead!

Throughout this appendix we've written down various symbols which are really realised-state (not reference-state) quantities, e.g. ρ, μ, t, a. When the time step is actually calculated in MorphoShell, it's reference-state quantities that are used instead. This is basically ok, since are results are essentially scaling results to be fine-tuned by hand anyway, and the reference-state versions should usually be the same kind of scale as the realised-state versions. However, there are a couple of things to bear in mind: The true stable time step (based on realised-state quantities) will be changing in a simulation as dialling occurs, so the time step we pre-compute just once based on reference-state quantities is just a decent guess really. Hence it's often good to make the time step prefactor in the settings a bit smaller than it absolutely needs to be at the start of the simulation, to give a bit of leeway/headroom for later. E.g. if running a simulation with a timestep prefactor of p seems to begin ok, while using $1.1p$ causes a crash within a few hundred time steps of starting, I'll often choose a prefactor of around $0.9p$ to run the simulation properly, but I've had to go as low as $0.2p$ sometimes.

Sometimes adjustments aren't needed in a way that seems surprising: For example, suppose you run a simulation for which the stretch time step T_s^{short} is the limiting one, and in which you don't dial density, modulus or thickness at all; you just take their dialled values to be their reference values. Now re-run the simulation with the same settings but a (dial factor = 1) $\bar{a}$ that is scaled by a factor of 1/8. Ignoring bend effects, you're going to arrive at basically the same final shape, except with 1/4 of the first area. You might think that this second simulation would require a much smaller time step prefactor than the first, since the eventual element size will be half the size of the first run's, which isn't accounted for in our time step calculation that uses reference-state quantities. Happily this isn't the case: the same prefactor can be used because the fact that we use a reference-state density to calculate the time step exactly compensates for the fact that we use a reference-state element size.

Finally, remember that if a simulation crashes before it finishes, you can probably re-start it from pretty much where it left off, using its penultimate output as an initialization, and using a slightly smaller time step prefactor, and maybe changing some other settings related to dialling, depending on your situation.

References

[1] Daniel Duffy, Joselle M. McCracken, Tayler S. Hebner, Timothy J. White, and John S. Biggins. Lifting, loading, and buckling in conical shells. *Phys. Rev. Lett.*, 131:148202, Oct 2023. doi: 10.1103/PhysRevLett.131.148202.

[2] Daniel Duffy and John S Biggins. Defective nematogenesis: Gauss curvature in programmable shape-responsive sheets with topological defects. *Soft Matter*, 16:10935–10945, 2020.

[3] Daniel Duffy, M Javed, MK Abdelrahman, TH Ware, Mark Warner, and JS Biggins. Metric mechanics with nontrivial topology: Actuating irises, cylinders, and evertors. *Physical Review E*, 104(6):065004, 2021.

[4] Fan Feng, Daniel Duffy, Mark Warner, and John Simeon Biggins. Interfacial metric mechanics: stitching patterns of shape change in active sheets. *Proc. R. Soc. A.*, 478(20220230), 2022. doi: 10.1098/rspa.2022.0230.

[5] Carl D Modes, Kaushik Bhattacharya, and Mark Warner. Gaussian curvature from flat elastica sheets. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 467(2128):1121–1140, 2011.

[6] Tyler Guin, Michael J Settle, Benjamin A Kowalski, Anesia D Auguste, Richard V Beblo, Gregory W Reich, and Timothy J White. Layered liquid crystal elastomer actuators. *Nature communications*, 9:2531, 2018. doi: 10.1038/s41467-018-04911-4. http://creativecommons.org/licenses/by/4.0/.

[7] D Duffy, Luka Cmok, JS Biggins, A Krishna, CD Modes, MK Abdelrahman, M Javed, TH Ware, Fan Feng, and Mark Warner. Shape programming lines of concentrated gaussian curvature. *J. Appl. Phys.*, 129(22):224701, 2021.

[8] A Giudici, A Clement, DL Duffy, M Ravi Shankar, and JS Biggins. Multiple shapes from a single nematic elastomer sheet activated via patterned illumination. *Europhysics Letters*, 140(3):36003, 2022.

[9] Tayler S Hebner, Riley GA Bowman, Daniel Duffy, Cyrus Mostajeran, Itay Griniasty, Itai Cohen, Mark Warner, Christopher N Bowman, and Timothy J White. Discontinuous metric programming in liquid crystalline elastomers. *ACS Applied Materials & Interfaces*, 15(8):11092–11098, 2023.

[10] Per-Olof Persson and Gilbert Strang. A simple mesh generator in matlab. *SIAM Review*, 46(2):329–345, 2004.

[11] Jerrold E Marsden and Thomas JR Hughes. *Mathematical foundations of elasticity*. Courier Corporation, 1994.

[12] Warner T Koiter. On the nonlinear theory of thin elastic shells. *Proc. Koninkl. Ned. Akad. van Wetenschappen, Series B*, 69:1–54, 1966.

[13] E. Efrati, E. Sharon, and R. Kupferman. The metric description of elasticity in residually stressed soft materials. *Soft Matter*, 9:8187–8197, 2013.

[14] Eitan Grinspun, Yotam Gingold, Jason Reisman, and Denis Zorin. Computing discrete shape operators on general meshes. *Computer Graphics Forum*, 25(3):547–556, 2006.

[15] Konrad Polthier and Markus Schmies. Straightest geodesics on polyhedral surfaces. SIGGRAPH 2006, page 30–38, 2006.

[16] J.-M. Morvan and D. Cohen-Steiner. Restricted delaunay triangulations and normal cycle. In *Symposium on Computational Geometry*, 2003.

[17] Xu Zhiqiang and Xu Guoliang. Discrete schemes for gaussian curvature and their convergence. *Computers & Mathematics with Applications*, 57(7):1187–1195, 2009.