nature computational science

Supplementary information https://doi.org/10.1038/s43588-024-00749-7

A programmable environment for shape optimization and shapeshifting problems

In the format provided by the authors and unedited

I. USABILITY

The *Morpho* project aims to support usability, consistent with recommendations from an evaluation of engineering software for mesh generation [1]: The code is supplied with a 132 page manual hosted on a separate GitHub repository [2] that incorporates elementary tutorials aimed at new users and a reference manual; a separate developer guide [3] provides documentation of advanced features. The main git repository [4] also includes numerous additional examples beyond those reported in the present paper, but fully described in the manual. Additional support mechanisms include a Slack channel[5] and a Youtube channel [6].

Morpho can be installed in two lines from the terminal via the *homebrew* package manager

```
brew tap Morpho-lang/Morpho
brew install Morpho Morpho-cli
    Morpho-Morphoview
```

Installation procedures with different package managers are in development. *Morpho* also aims to integrate into the surrounding environment; it can be run from the command line and includes an interactive mode similar to Surface Evolver and languages like Perl or Python; the reference manual is available and searchable in interactive mode as online help.

Advanced users who have created useful *Morpho* programs can package these for distribution, and are supported in coding for *Morpho* by a separate developer manual that is being continuously developed [3]. *Morpho* can be extended both through packages written in *Morpho* and external libraries written in C or C++. These facilities enhance the attractiveness of the package by promoting reusability of the code. External packages are provided to promote interoperability with relevant scientific software, including through VTK and JSON files, and to work with software like `Paraview`, `Gmsh`, etc. For those interested in contributing to *Morpho*, a contributor's guide is provided in the git repository [4]. A full unit-testing suite is also provided, incorporating more than 830 separate tests to enable the user to test the correctness of an installation or to assist developers interested in modifying the source.

II. PERFORMANCE COMPARISON WITH SURFACE EVOLVER

We briefly compare performance on a set of problems that can be solved both in *Morpho* and Surface Evolver 2.70 using the same optimization algorithm, either gradient descent with line search or conjugate gradient, in both codes. In Supplementary Table I we display runtimes for these examples using the same testing methodology and machine as above; note that we recompiled Surface Evolver for the ARM architecture (on the M2 Pro chip of the Macbook Pro) as the most recent binary provided is for the x86 architecture. The results demonstrate that our implementation is competitive with that of Surface Evolver. Note that these examples do not necessarily represent the *fastest* way to solve these problems in either code but are intended to test the overall implementation, e.g. mesh updates, force and energy calculations, etc. In Surface Evolver, for example, a specialized version of the Newton algorithm is available that often provides quadratic convergence when the solution is close to optimal. *Morpho*, on the other hand, provides explicit convergence tests that often lead to faster runtimes. We also have recently demonstrated that quasi-Newton algorithms lead to an order-of-magnitude speedup in performance on some problems [7].

	Time (s)	
Problem	Morpho	Evolver
Cube	1.34	2.61
Catenoid	0.37	0.46
Mound	0.96	0.68

Supplementary Table I: Comparison of runtimes for selected problems in *Morpho* and *Surface Evolver*.

III. STEP-BY-STEP EXAMPLES

We present three step-by-step examples with the program listing included. The goal of these is to quickly give the reader a sense of how *Morpho* works and what a typical program looks like; complete Supplementary Listings for all these examples are provided in the associated morpho-paper git repository[8, 9]. Further examples are to be found in the main github repository and in the manual[2], which contains extensive tutorials.

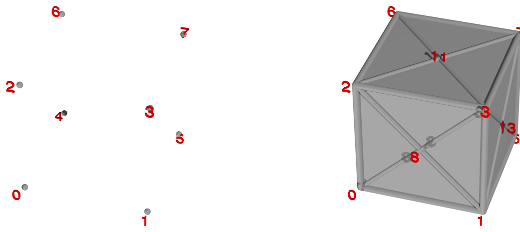
All of these programs follow a five step sequence which is very common in *Morpho* programs:—

1. Create a Mesh object, which is a discrete representation of the system.
2. Create subsidiary quantities associated with the mesh such as Fields and Selections.
3. Set up the optimization problem to be solved.
4. Perform the optimization.
5. Visualize the results.

A. Minimal Surfaces

Our first example is a canonical example of shape optimization as discussed in the main text. A minimal surface is one that minimizes its area, i.e. the functional,

$$F = \int_{\partial C} dA.$$



Supplementary Figure 1: Initial cube. (Left) Vertices specified by the user, labelled by their position in the list. (Right) Complete mesh. PolyhedronMesh adds additional vertices so that each face is represented by a sequence of triangles.

In the absence of constraints, the surface would collapse to a point, and so here we will consider a closed surface and fix the volume enclosed,

$$\int_C dV = V_0.$$

Here we will solve the problem in a way that closely follows an introductory example from the *Surface Evolver* software[10]. We start by creating a very coarse initial mesh, then set up the optimization problem and finally solve it with a gradient descent method. The complete Supplementary Listing to do all this is displayed in Supplementary Listing 1; we now discuss it step by step.

1. *Initialization*

Morpho is a modular piece of software, and hence most programs begin by "importing" modules or libraries that provide relevant functionality. It's necessary to import a module before using any of the functions or classes defined in it. In Lines 1-3, therefore, we import:

- `meshtools`, which provides some tools to help create and work with a Mesh.
- `plot`, which provides visualization capabilities.
- `optimize`, which contains the optimization package.

The beginning is also a good place to define variables that control the overall behavior of the program. In lines 5-6 we create two variables; `Nlevels`, which controls the number of levels of refinement the program will perform, and `Nsteps`, which sets the maximum number of optimization steps per refinement level.

2. *Create initial mesh*

We next create the initial Mesh in Supplementary Listing 1, lines 8-22 using a function from `meshtools`, PolyhedronMesh, that creates Meshes corresponding to polyhedra. To make a cube, for example, we make a list of

```
import meshtools
import plot
import optimize

var Nlevels = 4 // Levels of refinement
var Nsteps = 1000 // Maximum number of steps
    per refinement level

// Create an initial cube
var vertices = [[-0.5, -0.5, -0.5], // Vertex 0
                [ 0.5, -0.5, -0.5], // Vertex 1
                [-0.5,  0.5, -0.5], // Vertex 2
                [ 0.5,  0.5, -0.5], // ...
                [-0.5, -0.5,  0.5],
                [ 0.5, -0.5,  0.5],
                [-0.5,  0.5,  0.5],
                [ 0.5,  0.5,  0.5]]

var faces = [ [0,1,3,2], [4,5,7,6],
              [0,1,5,4], [3,2,6,7],
              [0,2,6,4], [1,3,7,5] ]

var m = PolyhedronMesh(vertices, faces)

// Set up the problem
var problem = OptimizationProblem(m)
var la = Area()
problem.addenergy(la)

var lv = VolumeEnclosed()
problem.addconstraint(lv)

// Create the optimizer
var opt = ShapeOptimizer(problem, m)

// Perform the optimization
for (i in 1..Nlevels) {
  opt.conjugategradient(Nsteps)
  if (i==Nlevels) break
// Refine
  var mr=MeshRefiner([m])
  var refmap = mr.refine()
  for (el in [problem, opt]) el.update(refmap)
  m = refmap[m]
}

// Visualize the result and save
Show(plotmesh(m, grade=2))
m.save("end.mesh")
```

Supplementary Listing 1: Minimal surface enclosing a fixed volume.

the eight vertices in lines 9-16 (see Supplementary Fig. 1, left). From these, we then create a list defining the six faces in lines 18-20, where the integers refer to the respective vertices in the list. Note that the vertices must be given *in order* going around each face. Once the vertices and faces are defined, we actually create the mesh in line 22. PolyhedronMesh automatically creates additional vertices and generates triangles to complete the

mesh, which is shown in Supplementary Fig. 1, right.

This is just one way to create a Mesh in *Morpho*; many other ways as described in the "Working with Meshes" chapter in the manual.

3. *Set up the optimization problem*

Now that the mesh is defined, we define the optimization problem in Supplementary Listing 1, lines 25-30. The overall problem is contained in an OptimizationProblem object, which is created in Supplementary Listing 1, line 26. We specify the target of the optimization—the Mesh contained in the variable `m`—as a parameter to the OptimizationProblem constructor function. OptimizationProblem is part of the optimize module that we imported in line 3.

Each term in the optimization problem is represented by a Functional object. These must be created and then added to the OptimizationProblem. Some Functionals are quite general and permit, for example, integration of arbitrary quantities; others are special purpose or optimized for speed.

Here, we create an Area functional in Supplementary Listing 1, line 26 and then add it to the OptimizationProblem in line 27. The method `addenergy` specifies that the functional concerned is to be included in the objective function of the optimization. Because we also want to preserve the volume enclosed by the mesh, we create a VolumeEnclosed functional in line 29. We add the resulting functional to the OptimizationProblem in line 30 using the `addconstraint` method. Area and VolumeEnclosed are just two examples of functional objects that are provided by *Morpho* for ease of use, with many more available; further, new functionals can be written in *Morpho* or added through an extension. Any functional object can equally be added as either to the objective function or used as a constraint. Local constraints are also available; see the second example.

4. *Perform optimization*

To perform the optimization, we create a ShapeOptimizer object in Supplementary Listing 1 line 33, passing it the OptimizationProblem just defined, and the Mesh object that will be the optimization target. ShapeOptimizers provide a number of methods that implement different algorithms; here we will use the conjugate gradient method[11].

The actual optimization step is performed in Supplementary Listing 1 line 37 by calling the `conjugategradient` method of the ShapeOptimizer. You'll notice that this is wrapped in a loop (lines 36-44) which handles the successive refinement as will be discussed in the next subsection. When called, optimizer then performs conjugate gradient steps until one of the following tests is satisfied:

1. The change in energy ΔE in the iteration satisfies,
$$|E| < \epsilon,$$
which checks for the (surprisingly common) case where the optimal value of the objective function is zero.
2. The change in energy ΔE in the iteration satisfies,
$$\left|\frac{\Delta E}{E}\right| < \epsilon,$$
where the value of ϵ is by default 1×10^{-8}, but can be changed by setting the `etol` property of the Optimizer object like so:
```
opt.etol = 1e-7
```
3. The number of iterations exceeds the value passed to the optimization algorithm. Here we passed the (intentionally large) value of `Nsteps` defined at the start of the program in line 5 to the `conjugategradient` method of our ShapeOptimizer.

You can determine whether the convergence check was passed by calling the method `hasconverged` on the ShapeOptimizer.

When developing a program to solve a new problem, it's often a good idea to start with one or just a few optimization steps to check that the optimization is proceeding as expected—this helps identify bugs in the specification of the problem—and then to crank up the maximum number of iterations until convergence is achieved.

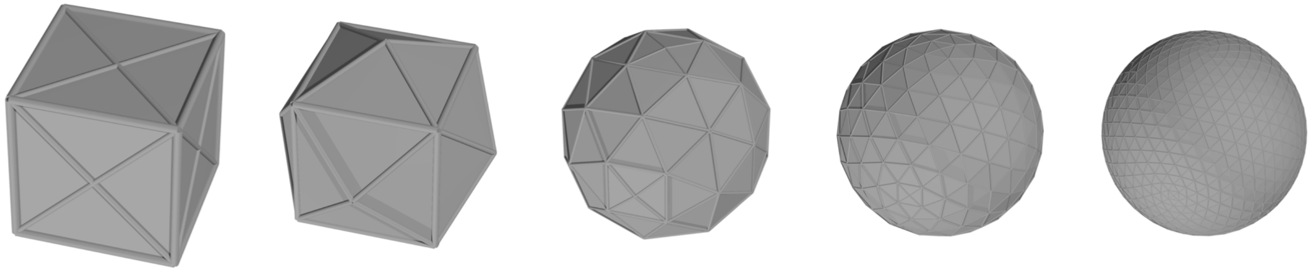
5. *Refinement*

In Supplementary Listing 1 lines 36 to 44, we optimize successive Meshes of increasing *refinement*. Refinement can mean a number of things, but in this context, it refers to creating a new mesh by replacing the triangles with smaller ones, and adding new vertices as appropriate. Refinement can be *adaptive*, localized to particular triangles where the shape is poorly resolved, or *global*, affecting all triangles equally. Here we will simply perform global refinement; the Mesh at each stage of optimization is shown in Supplementary Fig. 2.

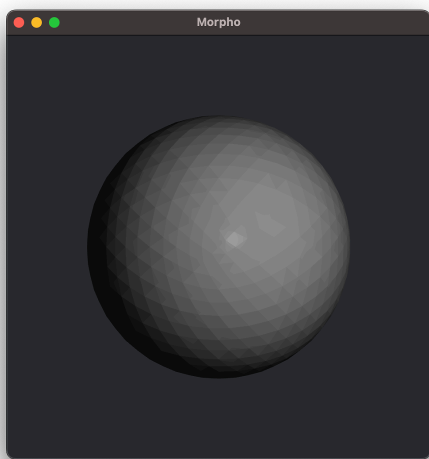
To do so, after optimizing the Mesh, we create a MeshRefiner object (from the `meshtools` module) in line 40. We give it the Mesh we want to refine.

Refinement happens by calling the `refine` method on our MeshRefiner in line 41, which returns a Dictionary object. The Dictionary maps the old, unrefined, objects—these are the Dictionary's keys—to the new, refined, objects created during the refinement process.

Having obtained the refined Mesh, we now need to update various other data structures. The OptimizationProblem and ShapeOptimizer are updated in line 42 by



Supplementary Figure 2: Successively refined meshes by splitting each triangle into four similar triangles shown after performing optimization at each stage.



Supplementary Figure 3: Optimized minimal surface enclosing a fixed volume after four levels of refinement, visualized in `Morphoview`.

calling their `update` method. These now refer to the updated Mesh. Finally, we update the variable m in line 43 to refer to the new Mesh by looking it up in the dictionary.

Refinement may look quite complicated, but quite commonly we will want to refine multiple inter-related objects at once; the MeshRefiner can take care of all of this for us. We simply have to update our data structures accordingly.

6. *Visualization*

Morpho provides a rich visualization capability. In Supplementary Listing 1, line 47 we use a function from the plot package, `plotmesh`, that draws the Mesh for us, creating a Graphics object. We use Show to display this in the Morphoview visualization program, which is a separate application installed along with *Morpho*. We also save the final result of the optimization by using the `save` method of the Mesh data structure in line 48.

Upon running this code, you should see a window like that in Supplementary Fig. 3 pop up containing the fully refined and optimized sphere. It's often a good idea to make visualizations and save results at intermediate steps; we could insert a call to plotmesh after line 38, for example, to see the optimized mesh at every stage.

B. Filaments

This example, inspired by experiments and calculations performed by a Prasath *et al.*[12], concerns an elastic filament strongly confined to the surface of a soap bubble under gravity. The filament is held fixed at the top and resists elastic deformation, but is forced to deform from its equilibrium configuration—a straight line—by the curvature.

The nondimensionalized energy of the filament on the unit sphere is, as described in the main text,

$$\tilde{F} = \int_0^{\Phi} [\Omega_g \chi^2(\tilde{s}) + \mathbf{S}(\tilde{s}) \cdot \hat{\mathbf{z}}] \, \mathrm{d}s, \tag{1}$$

where χ^2 is the curvature squared, the gravity term acts along the z direction and the integral is to be taken over the filament, which has arclength Φ. The quantity Ω_g quantifies the relative influence of curvature and gravity.

As for the first example, we proceed to solve this problem step by step. The Supplementary Listing is split into sections for clarity.

1. *Initialization*

In Supplementary Listing 2, we begin by importing packages in lines 1-3, and define the parameters in lines 5-10. We also define some metaparameters for the mesh and optimization process in lines 13-16.

The initial Mesh is created in line 19. The function LineMesh creates a Mesh given a one parameter function and a range of values. Hence, we supply an anonymous

```
1  import meshtools
2  import optimize
3  import plot
4
5  var R = 1.0 // Radius of sphere
6
7  // Initialize the filament with a given value
       of OmegaInv and Phi,
8  // the two dimensionless parameters explored
       in the paper.
9  var OmegaInv = 0.1 // Strength of bending
       energy relative to gravity
10 var Phi = 5 // Angular arclength of filament
       (2*Pi is a complete circle)
11
12 // Other parameters
13 var regulator = 5 // Strength of mesh
       regularizer (set emperically)
14 var N = 20 // Number of elements to discretize
       the filament
15 var Nsteps = 10000 // Maximum number of
       optimization steps
16 var initialStepSize = 0.002 // Initial
       stepsize for the optimizer
17
18 // Generate initial mesh
19 var m = LineMesh(fn (t) [R*sin(t), 0,
       R*cos(t)], 0..Phi:Pi/N)
20 var m0 = m.clone() // Keep a copy for
       visualization
```

Supplementary Listing 2: Setup and initial Mesh for filament example.

function that describes the geodesic,

$$\mathbf{X}(t) = \begin{pmatrix} R\sin t \\ 0 \\ R\cos t \end{pmatrix},$$

with limits set by the parameters `Phi` and `N` defined in lines 10 and 14 respectively.

We want to display the initial Mesh for visualization purposes later, so we make a copy by calling the `clone` method in line 20.

2. *Set up the optimization problem*

Having performed initialization, we now set up the optimization problem in Supplementary Listing 3. The OptimizationProblem object that defines the objective function is created in line 2. As in the previous example, each term in the objective functional is represented by a specific object. We include the curvature squared term in lines 5-6; note that the perfacter Ω_g is specified as the LineCurvatureSq object is added to the optimization problem in line 6.

The gravity term is implemented using a LineIntegral object in Supplementary Listing 3, line 10, which com-

```
1  // Setup optimization problem
2  var problem = OptimizationProblem(m)
3
4  // Bending energy
5  var linecurv = LineCurvatureSq()
6  problem.addenergy(linecurv, prefactor =
       OmegaInv)
7
8  // Gravitational potential energy
9  var up = Matrix([0,0,1])
10 var gravity = LineIntegral(fn (x) x.inner(up))
11 problem.addenergy(gravity) // Default
       prefactor is 1
12
13 // Constrain the total length
14 problem.addconstraint(Length())
15
16 // Constraint for the filament to lie on the
       sphere
17 var lc = ScalarPotential(fn (x,y,z)
       x^2+y^2+z^2-R^2)
18 problem.addlocalconstraint(lc)
19
20 // Regularization penalty function to ensure
       similar-sized elements
21 var eq = EquiElement()
22 problem.addenergy(eq, prefactor=regulator)
```

Supplementary Listing 3: Set up the optimization problem for the filament example.

putes the integral of a given function—here we supply an anonymous function—over line elements in a Mesh. We define a vector giving the "upward" direction in line 9, and the line integral computed is the dot product of the position with the up vector.

The length of the element is constrained in Supplementary Listing 3, line 14. We also want to constrain the filament to lie on the surface of a sphere, which is a local constraint. We implement this via a *level set* constraint in lines 17-18. A ScalarPotential object is created in line 17 with the scalar function,

$$f(x, y, z) = x^2 + y^2 + z^2 - R^2$$

which is zero on the desired feasible region. Once created, the ScalarPotential is added as a local constraint, i.e. one that applies to every Mesh degree of freedom.

We will find that it is necessary to add an additional term in the problem to ensure that line elements remain similar in size; this is a common strategy to regularize shape optimization problems and often improves convergence. An EquiElement object is created in Supplementary Listing 3, line 21 and added to the problem in line 22 to do this.

3. *Perform the optimization*

As before, the optimization is carried out by a

```
1  // Set up the optimizer
2  var opt = ShapeOptimizer(problem, m)
3
4  // Fix the top end of the filament
5  opt.fix(Selection(m, fn(x,y,z)
       x^2+y^2+(z-R)^2<0.0001))
6  opt.stepsize = initialStepSize
7
8  // Function to jiggle vertex positions by a
       small amount
9  fn jiggle(m, noise) {
10     for (id in 1...m.count()) { // Skip vertex
           0
11         var x = m.vertexposition(id)
12         x += noise*Matrix([randomnormal(),
13                            randomnormal(),
14                            randomnormal()])
15         m.setvertexposition(id, x)
16     }
17 }
18
19 // Jiggle filament a little to kick it out off
       the saddle point
20 jiggle(m, 0.005)
21 opt.conjugategradient(Nsteps)
```

Supplementary Listing 4: Optimizing the filament example. Note that since the starting point is a saddle point, we apply a small amount of gaussian noise to the initial configuration.

ShapeOptimizer object, which we create in line 2 of Supplementary Listing 4. To fix the top vertex in the filament, we call the fix method of the ShapeOptimizer in line 5; this takes a Selection, a collection of elements to keep fixed.

To create a Selection containing just the top vertex, we must provide the Selection constructor with the Mesh of interest, and a function that returns true within the region of interest. Since the top vertex is at coordinates $(0, 0, R)$, the inequality $x^2 + y^2 + (z - R)^2 < \epsilon^2$ will locate it if ϵ is chosen to be smaller than the distance between vertices. Here we pick a value of $\epsilon^2 = 0.0001$.

The ShapeOptimizer object has a number of parameters that we can select before performing the optimization. One is the initial stepsize to use, which is a fraction in the interval $0 < stepsize \leq 1$; the upper end corresponds to taking the full gradient step. Many optimization algorithms adjust the stepsize automatically, but typically need an initial guess. We therefore provide one in Supplementary Listing 4, line 6.

Before performing the optimization, we pause to note a peculiarity of this particular problem. The initial solution we prepared, where the filament lies exactly along a great circle is actually *already* an extremum of the functional (1); it's just not the one we're interested in (nor is it stable). If we run the optimizer from this initial guess, it simply converges on the initial solution. What we need to do is to perturb the initial guess a little bit.

```
1  // Visualize the filament and the covering
       sphere
2  fn visualize(m, m0) {
3      var gball=Graphics()
4      gball.display(Sphere([0,0,0], R-0.05))
5      gball.display(Arrow([0,0,R], [0,0,R+0.2]))
6      gball.display(Arrow([0,0,-R-0.4],
           [0,0,-R-0.2]))
7      return plotmesh(m, grade=[0,1]) + gball +
           plotmesh(m0, color=Blue)
8  }
9
10 Show(visualize(m, m0))
```

Supplementary Listing 5: Visualization code for filament example.

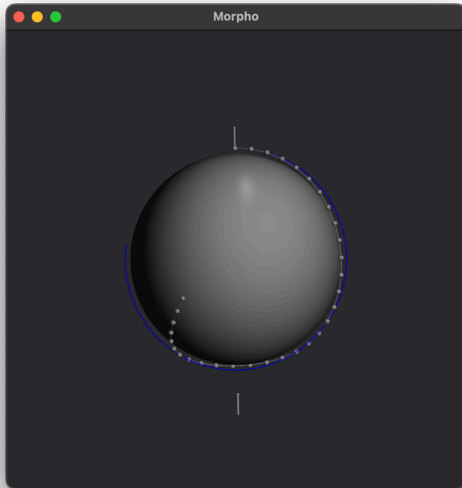
A function to do so is called `jiggle` and is defined in lines 9-17. It loops over the vertex ids in the Mesh (the number is obtained using the `count` method in line 10). In each iteration, the loop gets the position of a particular vertex in line 11, and then adds a random gaussian motion in lines 12-14 with standard deviation controlled by the `noise` parameter. Having calculated the new position, it is stored in the Mesh using the `setvertexposition` method in line 15. Note we skip the top vertex by starting the loop from id 1 in line 10.

Having defined the jiggle function, we use it in Supplementary Listing 4, line 20 to jiggle the initial guess, before performing the optimization using `conjugategradient` in line 21. The optimizer then performs conjugate gradient steps until the convergence criteria are satisfied or the maximum number of iterations is reached as in the previous example.

4. Visualization

Morpho provides rich visualization capabilities that can be readily customized to a particular problem. Users can draw Meshes as well as many other graphical elements. The filament example contains a good example of a custom visualization shown in Supplementary Listing 5. For convenience, in Supplementary Listing 5, lines 2-8 we create a function `visualize` that makes a visualization given two Mesh objects `m` (a configuration of interest) and `m0` (a reference configuration).

We first create a Graphics object in Supplementary Listing 5, line 3, which is an abstract container for graphical elements; this object doesn't actually draw anything on the screen or paper, but simply contains all the objects necessary to describe the scene we want to draw. The various items to be drawn are added to the Graphics object using the `display` method. We draw the sphere on which the filament is embedded on line 3—notice we use a slightly smaller radius to make the filament more visible—and then two arrows to indicate the upward direction in lines 5 and 6.



Supplementary Figure 4: Optimized filament embedded on a sphere, visualized in `Morphoview`. The initial configuration is displayed in blue; the final state is in grey.

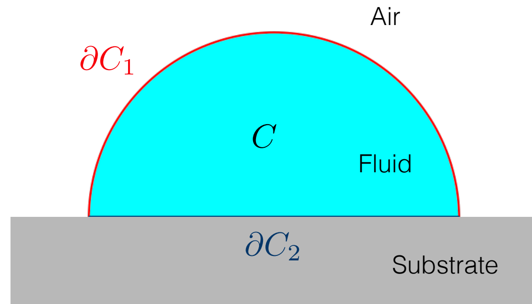
To display a Mesh object, we must use the plot package, which contains a number of functions that help visualize Meshes and associated data. Here we plot both meshes passed to the function as arguments using the `plotmesh` function in line 7; the reference Mesh is colored blue using an optional argument passed to `plotmesh`. Each plotmesh creates its own separate Graphics object; these are combined into a single Graphics object using the + operator and returned from the function in line 7.

In the main code, to visualize a configuration we simply call the visualize function with the Mesh of interest as well as a reference mesh to display. The `Show` function sends the returned Graphics object to the *Morphoview* utility, and the visualization appears in a separate window as shown in Supplementary Fig. 4.

C. Isotropic contact mechanics

The final two examples are closely related to the tactoid example presented in the main text, and inspired by a recent paper by Cousins *et al.*[13]. The code for the tactoid example is extensively discussed in the manual, and hence here we present something of an extension; how to determine the configuration of a liquid crystal droplet in contact with a boundary. This problem will also allow us to demonstrate some more advanced features of *Morpho*. We'll begin with a simpler problem, finding the shape of a 2D droplet made from isotropic fluid, and then extend this to the full nematic case.

A schematic of the domain of interest is displayed in Supplementary Fig. 5. The droplet occupies the domain C and is in contact with a substrate on its lower boundary ∂C_2 and the air on the upper boundary ∂C_1.



Supplementary Figure 5: Fluid droplet in contact with a surface.

The energy functional for the droplet is,

$$\sigma \int_{\partial C_1} dl + \Delta\sigma \int_{\partial C_2} dl,$$

where σ is the surface tension of the air-fluid interface and $\Delta\sigma$ is the difference between the surface tensions of the fluid-substrate and air-substrate interface. If $\Delta\sigma$ is positive, the droplet is *hydrophobic* and the fluid-substrate contact line will tend to vanish; if negative the droplet is *hydrophilic* and the droplet will tend to wet the substrate.

The optimization problem is subject to two constraints: first, there is a global constraint on the physical volume of the droplet,

$$\int_C dA = A_0,$$

which in 2D is really an area. Further, the lower boundary must lie on the substrate, which is a local constraint.

We pause to note that the formulation of this scenario as an optimization problem differs from typical presentations using the Young-Dupré equation (which balances forces at the contact point). The two formulations are equivalent for isotropic fluids, and the additional complexity of optimization is not required. However, by solving a simpler problem first, we can use the solution as a good starting point for the anisotropic problem later.

We will also structure our code a little differently from the previous two examples, utilizing *Morpho's* support for object oriented programming to better structure the code. As discussed in the Introduction above, most *Morpho* shape optimization programs have a similar structure, accomplishing a defined sequence of tasks. It therefore makes sense to create a class that corresponds to a particular problem, and define methods that accomplish each task.

An outline for our program's structure is shown in Supplementary Listing 6. In Supplementary Listing 6, lines 1-3 we import necessary modules. Supplementary Listing

```
import meshgen
import plot
import optimize

class Droplet {
    init(sigma=1, deltasigma=0) {
      // ...
    }

    initialMesh() {
      // ...
    }

    initialSelections() {
      // ...
    }

    setupProblem() {
      // ...
    }

    optimize(maxiterations=500) {
      // ...
    }

    visualize() {
      // ...
    }
}

var sim = Droplet(sigma=20,deltasigma=-4)

sim.initialMesh()
sim.initialSelections()
sim.setupProblem()
sim.optimize()

Show(sim.visualize())
```

Supplementary Listing 6: Outline code for isotropic contact mechanics**.**

6 lines 5-29 define the Droplet class, which incorporates a number of methods that we will define in subsequent sections.

Supplementary Listing 6 lines 31-38 execute the sequence of activities necessary to solve the problem: we create an instance of Droplet with defined parameters in line 31, and then call appropriate methods in order. Finally, the visualization is displayed in Morphoview in line 37.

We'll now define each of the missing methods in turn:

1. *Droplet initializer*

The `init` method, shown in Supplementary Listing 7, is a special method that is always called when an object is created and should set up the object ready for use. In this example we define `init` to take two optional parameters,

```
init(sigma=1, deltasigma=0) {
    self.sigma = sigma // Isotropic surface
        tension fluid-air
    self.deltasigma = deltasigma // Difference
        between surface tension of
        air-substrate interface and
        fluid-substrate interface
}
```

Supplementary Listing 7: Initializer method for isotropic contact mechanics.

```
initialMesh() {
    var c = CircularDomain([0,0], 1)
    var hs = HalfSpaceDomain(Matrix([0,0]),
        Matrix([0,1]))
    var dom = c.difference(hs)
    var mg = MeshGen(dom, [-1..1:0.2,
        -1..1:0.2], quiet=true)
    self.mesh = mg.build()
}
```

Supplementary Listing 8: Creating an initial Mesh for isotropic contact mechanics.

the surface tensions `sigma` and `deltasigma`, which are stored in the Droplet object's properties in lines 2-3.

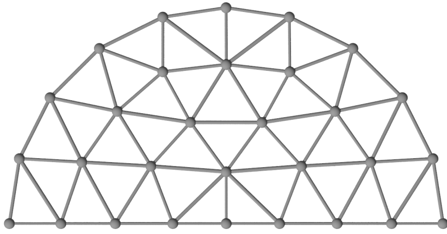
More generally, the `init` method should do the *minimum* necessary to initialize the object. It should *not* perform complex or extensive calculations; these should be reserved for other methods. If an object is complicated to create, it may be worth defining a second class to actually build it (the MeshBuilder class in the `meshtools` package is a good example).

2. *Create the initial mesh*

We want to create a Mesh corresponding to the upper half of the unit disk. To do this, we'll use the `meshgen` module, which allows us to defined a target domain using simple geometric objects—or more complex regions defined by scalar functions—combine them using set operations, and then generate a Mesh from the target domain. Many examples in both two and three dimensions are provided in the *Morpho* manual chapter "Working with Meshes".

The `initialMesh` method for the present problem is shown in Supplementary Listing 8. In line 2 we create a CircularDomain object with center $(0,0)$ and radius 1. In line 3 we define the unit half space with edge including the point $(0,0)$ and exterior normal $(0,1)$; this is equivalent to the region $y < 0$. The final domain $c\backslash hs$ is created in line 4 using the `difference` method.

Having defined the domain, we create a MeshGen object in line 5 to build the Mesh. It is necessary to provide a bounding box in the constructor by giving a Range object for each dimension; here we use the box $x, y \in [-1, 1]$.



Supplementary Figure 6: Initial Mesh created using the meshgen module.

```
1  initialSelections() {
2      self.bnd = Selection(self.mesh,
           boundary=true)
3      self.upper = Selection(self.mesh, fn (x,y)
           x^2+y^2>0.99)
4      self.lower = Selection(self.mesh, fn (x,y)
           y<0.01)
5      self.upper.addgrade(1)
6      self.lower.addgrade(1)
7  }
```

Supplementary Listing 9: Performing the optimization for isotropic contact mechanics.

The step provided in the Range object is an approximate spacing for the vertices of the Mesh. By default, MeshGen emits information about its operations and so we set `quiet=true` to suppress this output.

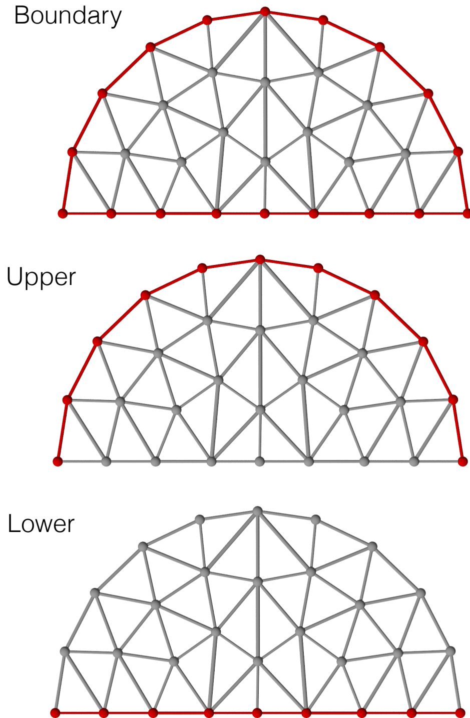
Finally, the Mesh is built by calling the `build` method in line 6 and stored in the `mesh` property of the current Droplet object (referred to using the keyword `self`). The resulting Mesh is displayed in Supplementary Fig. 6

3. Create subsidiary quantities

While this problem doesn't require any Fields, we will want to refer to the boundary of the Mesh, and indeed the upper and lower parts separately. To create the necessary Selection objects, the `initialSelections` method is defined in Supplementary Listing 9.

In Supplementary Listing 9, line 2, we create a Selection that represents the boundary of the Mesh. This selection includes both the line elements on the boundary and the vertex elements attached. In line 3 we create a Selection for the upper air-fluid boundary ∂C_1 by finding vertices on the unit circle. Notice the inequality $x^2 + y^2 > 0.99$ which is `true` for all the boundary vertices. Line 4 creates a Selection for the lower fluid-substrate boundary ∂C_2 by finding vertices below the line $y < 0.01$. Note in both cases, because the Selection constructor only finds the vertices, we need to extend each Selection to include the line elements (grade 1) using the `addgrade` method.

Creating Selections can be error prone: it's important



Supplementary Figure 7: Selections for different regions of interest on the initial Mesh.

to check that they include exactly the elements you expect, or later parts of your problem might fail. A missing component of a Selection, for example, could cause your optimization problem to be incorrectly defined and hence ill-posed, causing the Optimizer to fail to converge.

To help with this, the `plot` package provides a function, `plotselection`, that helps visualizes the selected elements. You could add the following code,

```
Show( plotselection (sim.mesh, sim.upper,
grade =[0,1])),
```

after line 33 in the main code (Supplementary Listing 6) to check that the Selection held in the upper property is correct, for example. Plots for all three Selections produced in this way are shown in Supplementary Fig. 7.

4. Set up the optimization problem

The optimization problem is set up in the `setupProblem` method reusing the same concepts from other examples: An OptimizationProblem object is created in Supplementary Listing 10 line 2 using the Droplet's mesh. We create a Length functional in line 4, which is then applied to the upper and lower boundary separately in lines 5 and 6; notice that each of these uses the optional parameters `selection` and `prefactor` to set the correct region and prefactor appropriately. Also note that only one Length object is required; the

```
1  setupProblem() {
2      var problem=OptimizationProblem(self.mesh)
3
4      var llength = Length()
5      problem.addenergy(llength,
           selection=self.upper,
           prefactor=self.sigma)
6      problem.addenergy(llength,
           selection=self.lower,
           prefactor=self.deltasigma)
7
8      var larea = Area()
9      problem.addconstraint(larea)
10
11     var lcons = ScalarPotential(fn (x,y) y)
12     problem.addlocalconstraint(lcons,
           selection=self.lower)
13
14     self.problem = problem
15 }
```

Supplementary Listing 10: Performing the optimization for isotropic contact mechanics.

```
1  optimize(maxiterations=500) {
2      var opt = ShapeOptimizer(self.problem,
           self.mesh)
3      opt.conjugategradient(maxiterations)
4  }
```

Supplementary Listing 11: Performing the optimization for isotropic contact mechanics.

optimizer uses it for each Selection separately.

The global area constraint is enforced in Supplementary Listing 10, lines 8-9 by creating an Area object and adding it to the OptimizationProblem using `addconstraint`. The local constraint that the lower boundary be confined to the line $y = 0$ is enforced in lines 11-12 using a ScalarPotential object with an appropriate anonymous function; the ScalarPotential created is added to the problem using `addlocalconstraint`.

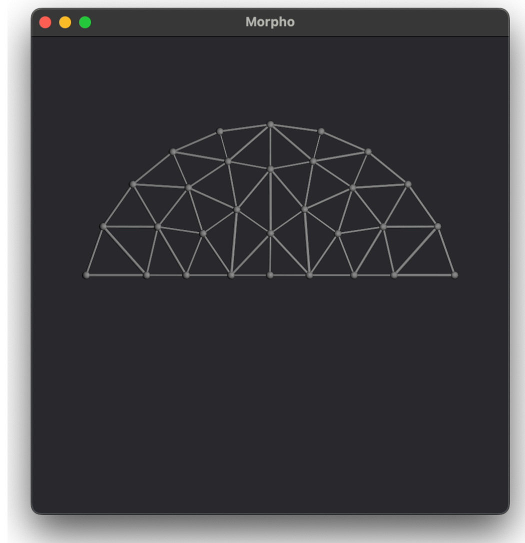
Once the OptimizationProblem has been successfully defined, it's stored in the Droplet object's `problem` property in Supplementary Listing 10, line 14.

5. *Perform the optimization*

The `optimize` method for this problem is shown in Supplementary Listing 11. We define it to take an optional parameter `maxiterations` so that the number of iterations can be easily adjusted; the default value of 500 is a reasonable start. A ShapeOptimizer is created in Supplementary Listing 11 line 2, and then conjugate gradient steps are performed in line 3 using the standard convergence criteria as used in the previous problems.

```
1  visualize() {
2      return plotmesh(self.mesh, grade=[0,1])
3  }
```

Supplementary Listing 12: Visualization method for isotropic contact mechanics.



Supplementary Figure 8: Optimized iostropic fluid droplet in contact with substrate.

6. *Visualize results*

The visualize method for the isotropic contact mechanics problem, shown in Supplementary Listing 12, is very simple; we simply use the `plotmesh` function from the plot package to display the Mesh. Here we have chosen to display the vertices and line elements (grades 0 and 1 of the Mesh) for clarity.

7. *Running the code*

With the complete code, defined by a inserting the method definitions in Supplementary Listings 7-12 into the outline Supplementary Listing 6, is complete, it can be run which will produce the final configuration shown in Supplementary Fig. 8. Try adjusting the value of $\Delta\sigma$—how does this change the resulting configuration?

D. Nematic contact mechanics

We will now extend our code to solve the analogous problem for a nematic droplet in contact with the substrate. In addition to the shape of the droplet, we must also find the associated director field $\mathbf{n}(\mathbf{x})$ that minimizes the energy functional, which is given by,

```
class NematicDroplet is Droplet {
    init(...) {}

    initialField() {}
    importData(obj) {}

    setupProblem() {}
    optimize(...) {}

    visualizeDirector() {}
    visualize() {}
}
```

Supplementary Listing 13: Class outline for nematic contact mechanics.

$$F = \frac{K}{2}\int_C (\nabla\cdot\mathbf{n})^2\, dA + \sigma\int_{\partial C_1} dl + \Delta\sigma\int_{\partial C_2} dl - w_1\int_{\partial C_1} (\nabla\cdot\mathbf{t})^2\, dl - w_2\int_{\partial C_1} (\nabla\cdot\mathbf{t})^2\, dl. \quad (2)$$

The first term is the elasticity of the liquid crystal that favors spatially uniform alignment in the so-called one-constant approximation; the second and third terms are the isotropic surface tension as in the earlier problem; the final terms are the anisotropic—i.e. orientation dependent—part of the surface tension often referred to as *anchoring* terms in the liquid crystal community. These latter terms favor alignment of the nematic with with the local tangent vector to the boundary. The functional (2) is to be minimized with respect to the droplet shape C and the configuration of the liquid crystal $\mathbf{n}$ subject to the area constraint,

$$\int_C dA = A_0,$$

and local constraint on the boundary,

$$y = 0 \forall \partial C_2,$$

together with a new local constraint on the director field,

$$\mathbf{n}\cdot\mathbf{n} = 1.$$

Because this new problem extends the isotropic case above, we would like to reuse the isotropic code already developed. A natural way to do so is to exploit *Morpho*'s support for *inheritance* by defining a new class, NematicDroplet that extends the Droplet class already defined.

An outline for the new class is displayed in Supplementary Listing 13, showing the new methods we need to define. NematicDroplet inherits all the method definitions from Droplet—they are copied into the class definition before we define new methods—and we will reuse many of these. Any new methods we define in NematicDroplet that have the same name as one in Droplet *replace* the old method, but we can still call that method using the keyword `super` as we shall see below.

```
init(initialState=nil, sigma=1, deltasigma=0,
    k=1, wupper=1, wlower=1) {
    super.init(sigma=sigma,
        deltasigma=deltasigma)

    self.k = k
    self.wupper = wupper
    self.wlower = wlower

    if (initialState)
        self.importData(initialState)
}

importData(obj) {
    self.mesh = obj.mesh
    self.upper = obj.upper
    self.lower = obj.lower
    self.bnd = obj.bnd
}

initialField() {
    self.director = Field(self.mesh,
        Matrix([1,0,0]))
}
```

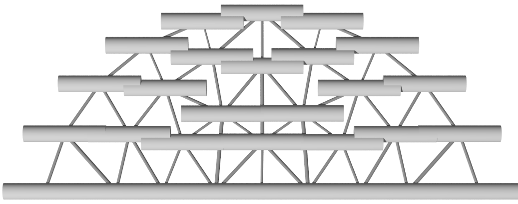
Supplementary Listing 14: Initialization for nematic contact mechanics.

1. *Initializing a NematicDroplet*

Our NematicDroplet initialization methods are shown in Supplementary Listing 14. We begin by calling Droplet's initializer in line 2. The keyword `super` is used to indicate that it's the *superclass's* method that should be called, not the method defined on the current NematicDroplet class. It's a common practice to do this in the init method of a subclass to ensure any initialization the superclass expects has been done correctly.

The `init` method is a little more complicated because we now have additional parameters. Defining these using the optional parameter syntax is helpful to the user because, first, default values can be provided and hence the user may not need to always specify them; moreover they can be specified in the method call in arbitrary order. Lines 4-6 copy the values provided by the user in the constructor, or the default values as appropriate, into the NematicDroplet's properties.

To assist the user, we also provide an optional parameter `initialState`. If this is specified by the user, in Supplementary Listing 14, line 8 we call a method `importData` defined in lines 11-15 that copies the mesh and selections from a supplied object to this one. This in-



Supplementary Figure 9: Initial configuration for nematic droplet, starting from a uniform director field.

terface allows the user to prepare and minimize a Droplet and then use it as the initial configuration for the nematic optimization. Of course, if they wish, they could still use the `initialMesh` and `initialSelections` methods that are copied from Droplet into NematicDroplet by inheritance.

Since we now have a director field to consider as well as the droplet, we will accordingly need to create an initial Field object; hence we define a method `initialField` that simply creates a uniform configuration $\mathbf{n} = (1, 0, 0)$. Even though the problem here is two dimensional—the droplet should really be viewed as a prism extruded into the paper—we define a 3D director, because we may wish to examine scenarios where $\mathbf{n}$ points out of the plane. We could constrain the director to lie in plane by creating a 2D Field, $\mathbf{n} = (1, 0)$; we would need to mildly change the visualization code below if so since this assumes a 3D director. The initial configuration of the system, starting from the a droplet shaped obtained from the isotropic problem, is displayed in Supplementary Fig. 9 using the visualization code described in subsection III D 4 below.

2. *Setting up the problem*

Supplementary Listing 15 displays the code for the new `setupProblem` method. Since the nematic problem encapsulates the isotropic droplet problem, we reuse the old `setupProblem` method by calling Droplet's `setupProblem` method in line 2 using `super`. After this line is executed, the isotropic problem has been correctly set up, and we can simply append the new Functionals to it.

We prepare and add nematic elasticity in Supplementary Listing 15, lines 4-5 using a Nematic functional. The anchoring energies are more complicated. We define a LineIntegral object in line 12 that will compute the anchoring energy, and give it the director as a parameter. We also need to provide an integrand function, which here is defined as a local function, `anchintegrand` in Supplementary Listing 15, lines 7-10. Functions defined within a method or function are only visible within the function, and hence are a good alternative where an anonymous function would be too long.

The integrand function, `anchintegrand`, obtains the local tangent vector by calling the special `tangent` function, and then computes the dot product. The local value of the director is computed from the Field by interpolation and passed to the integrand function as a parameter. Because $\mathbf{n}$ is a three dimensional vector and the tangent is two dimensional, we compute the dot product manually and return it in line Supplementary Listing 15, 9.

```
1  setupProblem() {
2      super.setupProblem() // Initialize the
           problem with the isotropic part
3
4      var lnem = Nematic(self.director) //
           Nematic elasticity
5      self.problem.addenergy(lnem, prefactor =
           self.k)
6
7      fn anchintegrand(x, n) { // Integrand for
           anchoring energy
8          var t = tangent()    // Gets the local
               tangent vector
9          return (n[0]*t[0]+n[1]*t[1])^2
10     }
11
12     var lanch = LineIntegral(anchintegrand,
           self.director) // Anisotopic surface
           tension (anchoring)
13     self.problem.addenergy(lanch,
           selection=self.lower,
           prefactor=-self.wlower)
14     self.problem.addenergy(lanch,
           selection=self.upper,
           prefactor=-self.wupper)
15
16     var lnorm = NormSq(self.director)
                                  // Unit vector
           constraint
17     self.problem.addlocalconstraint(lnorm,
           field=self.director, target=1)
18 }
```

Supplementary Listing 15: Problem setup for nematic contact mechanics.

Once this apparatus is set up, the anchoring energies are added to the problem in Supplementary Listing 15, lines 13-14. Notice that the domain of each is specified using the `selection` parameter to `addenergy`, and the appropriate coefficient is also provided.

The unit length constraint on the director is imposed in Supplementary Listing 15, lines 16-17 using the NormSq functional, and the desired target value of 1 specified as a parameter to the `addlocalconstraint` method.

3. *Performing the optimization*

Since we must optimize the energy functional both with respect to shape and the director, we must incorporate the additional optimization. We construct an Alternating Optimization scheme in the new optimize method displayed in Supplementary Listing 16. As in

```
optimize(maxiterations=500, altiterations=10) {
    var opt = ShapeOptimizer(self.problem,
        self.mesh)
    var fopt = FieldOptimizer(self.problem,
        self.director)

    for (i in 1..maxiterations) { //
        Alternating optimization scheme for
        Shape and Field
        fopt.conjugategradient(altiterations)
        opt.conjugategradient(altiterations)
        if (opt.hasconverged() &&
            fopt.hasconverged()) break
    }
}
```

Supplementary Listing 16: Optimization method for nematic contact mechanics.

the isotropic case, we create a ShapeOptimizer in Supplementary Listing 16, line 2, but now we also create a FieldOptimizer in line 3, and give it the Field to be optimized as a parameter.

Having created the optimizers, the alternate between performing conjugate gradient steps on the shape and the field respectively in Supplementary Listing 16, lines 5-9. The optional parameter altiterations specifies how many iterations of each to perform. Whether each optimizer has converged according to the standard criteria defined above in Subsection III A 4 is checked in line 8 and the loop terminated if so using the `break` keyword.

Performance of the alternating optimization scheme may depend on the relative number of optimization steps taken on the field and mesh; you are encouraged to investigate the effect of changing this. By design, Morpho gives you the ability to control how optimization is performed because this is strongly problem dependent. Nonetheless, even without tuning the optimization the present code overall runs in a fraction of a second on a macBook Pro M2.

Note that the `optimize` *package is under very active development and we expect improved optimizers to be available in future releases.*

4. Visualization

We now need to visualize the director field **n** in addition to the Mesh, so we create a method `visualizeDirector` to do so. This gives a valuable illustration of *Morpho's* custom visualization capabilities; it's shown in Supplementary Listing 17. The method generates a Graphics object by drawing a cylinder at every vertex in the Mesh oriented locally along the direction of **n**. We choose a cylinder not an arrow because the physics does not depend sign of **n**.

One question is how large should the cylinder be? We could leave this as a user specified parameter, but it's nice

```
visualizeDirector() {
    // Estimate scale from mean mesh separation
    var scale = 0.7*Length.total(self.mesh)/
                    self.mesh.count(1)

    var g = Graphics()
    for (id in 0...self.mesh.count()) {
        var x = self.mesh.vertexposition(id)
            // Get the position of vertex id
        var xx = Matrix([x[0], x[1], 0])
            // Promote it to a 3D vector
        var nn = self.director[0,id]
            // Get the corresponding director
        g.display(Cylinder(xx-scale*nn,
            xx+scale*nn, aspectratio=0.2,
            color=White))
    }
    return g
}

visualize() {
    return plotmesh(self.mesh, grade=1) +
        self.visualizeDirector()
}
```

Supplementary Listing 17: Visualization method for nematic contact mechanics.

to calculate this automatically from the Mesh itself. We compute the average length of an edge Supplementary Listing 17, lines 3-4 as follows: First, we can compute the total length of all line elements in the mesh using the `total` method of the Length functional. Note we can call this directly on the class; we don't need to instantiate a Length object. We then obtain the number of line elements using the `count` method of the Mesh. The average length is the ratio of these two quantities. The scaling of 0.7 was chosen arbitrarily to ensure the cylinders don't overlap.

Having created an empty Graphics object in line 6, we loop over all Mesh vertex ids in Supplementary Listing 17, lines 6-11. For each vertex, we obtain its position in line 8 and convert this from a 2D vector to a 3D vector in line 9. The value of **n** at the appropriate vertex is obtained in line 9 (note that the index takes two values, the first, 0, indicates grade zero or vertices, and the second is the vertex id. We then draw the cylinder from `x` − `scalen` to `x` + `scalen` with an aspect ratio of 0.2 and color set to White in line 11. The completed Graphics object is returned in line 13.

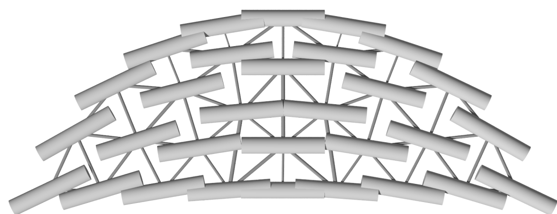
A separate method, `visualize`, displays both Mesh and Field together in Supplementary Listing 17, lines 16-18. The visualize method from the Droplet superclass was so simple it's not necessary to reuse it, and in any case we prefer not to display the vertices now that directors are being drawn. The new visualize class draws the Mesh with `plotmesh`, calls the `visualizeDirector` method and combines the two Graphics objects.

```
var nsim = NematicDroplet(sigma=20,
              deltasigma=-4, k=1,
              wupper=1, wlower=1,
              initialState=sim)

nsim.initialField()
nsim.setupProblem()
nsim.optimize()

Show(nsim.visualize())
```

Supplementary Listing 18: Using the NematicDroplet class to determine the equilibrium droplet shape and director configuration.



Supplementary Figure 10: Optimized configuration for the nematic droplet.

5. Running the code

Now that the new NematicDroplet class is defined, the code to use it is very short indeed as shown in Supplementary Listing 18. We assume the NematicDroplet code in Supplementary Listing 13 is inserted after Supplementary Listing 6, with all method definitions filled out in both classes and the minimization code for the Droplet in place. In 18, line 1-4, we create an instance of the NematicDroplet class.

To use the results of the earlier minimization, stored in the variable `sim`, as an the initial configuration for our NematicDroplet, we pass it to the NematicDroplet constructor using the `initialState` optional argument that we created especially for this task.

Having set up a NematicDroplet, to obtain the solution we need to add an initial field, construct the problem and then perform the optimization which is done in 18, lines 6-8. The resulting configuration, shown in Supplementary Fig. 10, is produced by our new `visualize` method and displayed with Show in line 10.

We pause to observe the value of using an object-oriented approach to writing a *Morpho* program for this task. We *directly* reused more than half of the code already defined in Droplet in our NematicDroplet; the resulting definition was therefore significantly shorter. Because it's typically desirable to start optimization from a good guess, we reused the *result* of a Droplet calculation for our new problem. Further, what the code is doing is much simpler and *clearer* to the reader than if we had

```
refine() {
    var mr = MeshRefiner([self.mesh,
        self.director, self.bnd, self.upper,
        self.lower])
    var refmap = mr.refine()

    // Now refinement is done update the
        problems and optimizers
    for (el in [self.problem])
        el.update(refmap)

    // Update our references
    self.mesh = refmap[self.mesh]   // There
        are tidier ways to do this!
    self.director = refmap[self.director]
    self.bnd = refmap[self.bnd]
    self.lower = refmap[self.lower]
    self.upper = refmap[self.upper]
}
```

Supplementary Listing 19: Refinement method for the NematicDroplet class.

written our program without classes. Because the methods were named to reflect the activities performed, the code is self-documenting and requires less commenting.

One can imagine further refinements. It's possible to define the Droplet and NematicDroplet classes in separate files from the "driver" code, for example, and incorporate them using `import`. This would make them reusable in other contexts: They could even be distributed as a *Morpho* package, a public git repository with a well-defined structure that can incorporate Morpho help files and other supporting material as desired. Further details are supplied as part of the *Morpho* developer's guide available from the *Morpho* Github repository.

E. Refinement

While the code is elegantly written, the solution obtained so far (Supplementary Fig. 10) can be improved upon considerably. We might like to refine the solution, to obtain a better approximation to the solution to the continuous problem. We might also be interested in changing the parameters. Both of these possibilities will be explored in this section.

1. Refinement

The mesh refinement process follows the same structure as that in Subsection III A 5 for the minimal surface example, but is a bit more complicated due to the additional Field and Selection objects that must be refined. To facilitate it, we'll add a new method to the NematicDroplet class called `refine`, displayed in Supplementary

```
var nsim = NematicDroplet(sigma=20,
            deltasigma=-4, k=1,
            wupper=1, wlower=1,
            initialState=sim)

nsim.initialField()
nsim.setupProblem()
nsim.optimize()

for (i in 1..2) {
    nsim.refine()
    nsim.optimize()
}

Show(nsim.visualize())
```

Supplementary Listing 20: Driver code for refinement following optimization.

Listing 19.

In Supplementary Listing 19, line 2, we create a MeshRefiner object, which is initialized with an list of objects that are to be refined. The actual refinement happens in line 3, which returns a Dictionary that maps the coarse Mesh to the newly refined Mesh, as well as all subsidiary objects.

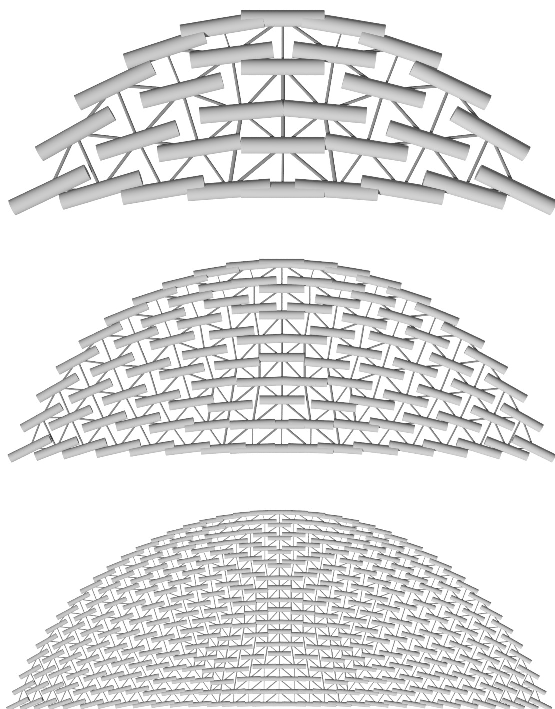
Once we have this Dictionary, we must update objects that depend on these data structures. The only one that does is the OptimizationProblem object; we therefore call the update method on this object in Supplementary Listing 19, line 6. We use a loop so that the list of objects to be updated can be readily extended if the code is modified.

Note that, in the code presented, fresh ShapeOptimizer and FieldOptimizer objects are created every time the `optimize` method is called. Creation of objects is very cheap in *Morpho*, and creating the optimizers as needed obviates the need to update them following refinement making coding less error prone. It's nonetheless quite possible to re-use optimizer objects, and if this is done they should be updated here.

Finally, in Supplementary Listing 19, lines 9-13 we update the properties of the current NematicDroplet object by looking up new values in the Dictionary.

Now the new `refine` method is implemented, we can adapt the driver code to perform refinement. The code to do so is shown in Supplementary Listing 20; this *replaces* the previous driver code from Supplementary Listing 18. The only difference is the addition of Supplementary Listing 20, lines 10-13, which is a loop that successively refines and optimizes the solution. This process is often referred to as Nested Iteration, and greatly improves the performance of the code relative to optimizing a fine solution on a fixed Mesh. The resulting set of solutions obtained are displayed in Supplementary Fig. 11.

Again, the benefits of the object oriented structure are apparent: the driver code (Supplementary Listing 20) needed to be only very mildly changed and a new method



Supplementary Figure 11: Successively refined and optimized solutions for the nematic droplet.

```
updateParameters(sigma=nil, deltasigma=nil,
    k=nil, wupper=nil, wlower=nil) {
    if (sigma) self.sigma=sigma
    if (deltasigma) self.deltasigma=deltasigma
    if (k) self.k = k
    if (wupper) self.wupper = wupper
    if (wlower) self.wlower = wlower

    self.setupProblem() // Regenerate problem
}
```

Supplementary Listing 21: Method to update system parameters for the NematicDroplet class.

defined to accommodate what is a large improvement in the program. If we wanted to further adjust how the optimization was performed, or the sequence of operations, we would only have to change a few relevant lines.

2. *Continuation*

A common goal in simulation is to explore the range of possible solutions, and how they depend on the system parameters. Continuation is an approach to explore the solution space that involves slowly changing a parameter of interest and following how the optimized solution evolves. Within the object oriented approach, it's very easy to implement continuation. To assist, we'll begin

```
1  for (w in 1..5:1) {
2      print "===Wupper=${w}"
3      nsim.updateParameters(wupper=w)
4      nsim.optimize()
5  }
```

Supplementary Listing 22: Continuation driver loop for the NematicDroplet class, which slowly increased the anchoring energy on the upper boundary.

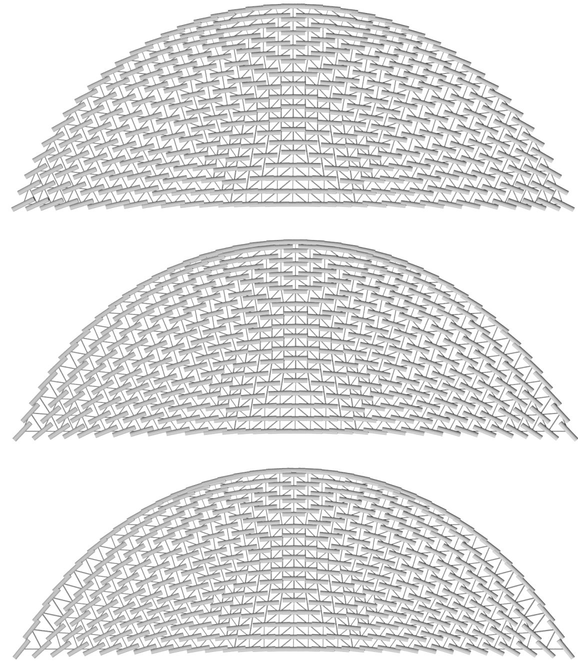
```
1  regularize(maxiterations=10) {
2      var reg=OptimizationProblem(self.mesh) //
           Create an ancillary regularization
           problem
3
4      var leq = EquiElement() // Function to
           equalize elements
5      reg.addenergy(leq)
6
7      var lcons = ScalarPotential(fn (x,y) y) //
           Level set constraint for lower boundary
8      reg.addlocalconstraint(lcons,
           selection=self.lower)
9
10     var ropt = ShapeOptimizer(reg, self.mesh)
11     ropt.stepsize = 0.001
12     ropt.fix(self.upper) // Fix upper boundary
13
14     ropt.conjugategradient(maxiterations)
15     equiangulate(self.mesh) // Edge flips
16 }
```

Supplementary Listing 23: Regularization method for the NematicDroplet class. This should be called during continuation to improve the quality of the Mesh

by adding a method to NematicDroplet that allows us to easily change parameters (see Supplementary Listing 21).

Notice the design of the optional parameters: their default value is `nil`, which is the *Morpho* keyword that indicates the *absence* of a quantity and always evaluates as `false` when used in a condition. In the method's implementation, Supplementary Listing 21 lines 2-6, parameters are only updated if they are not `nil`. Hence, the user need only provide parameters they wish to set in a call to `updateParameters`.

To perform continuation, we simply need to include a loop in the driver code that traverses the desired trajectory in parameter space. An example is shown in Supplementary Listing 22, which slowly increases the anchoring energy on the upper surface, and re-optimizes after each step. This loop should be inserted after the first loop in Supplementary Listing 20, i.e. after line 13. In this particular case, we are able to use a large stepsize in line 1, but it's quite common that the stepsize be significantly smaller for more challenging problems. Optimized configurations for selected parameters are shown in Figure 12.



Supplementary Figure 12: Nematic droplet solutions for different values of anchoring parameter $W \in 1, 3, 5$ (increasing downwards) on the upper boundary.

3. *Further improvements*

The nematic contact mechanics code presented here could be developed even further. Rather than the global refinement performed here, it may be valuable to perform adaptive refinement. Parameter studies where the droplet shape changes significantly usually require some form of *regularization*, where the internal vertices are periodically moved to render the elements equal in size, or equal in energy. An example of a method that tries to do this is shown in Supplementary Listing 23; it should be added to the NematicDroplet class definition and called in the continuation driver loop before each optimization step (e.g. between lines 1 and 2 in Supplementary Listing 22).

IV. STABILITY ANALYSIS

We close with a more advanced topic: It's often desirable to determine whether solutions obtained with *Morpho* are stable, i.e. whether we are at a minimum of the energy or, as is quite common, at a saddle point. In this section we'll re-examine the minimal surfaces from the previous section to assess their stability.

Before doing so, let us consider the problem of stability more generally. Imagine that we perturb the solution in an arbitrary feasible direction δx. If the problem were unconstrained, we can model the energy (we'll refer to this as the objective function in the remainder of this

```
// Returns the inertia (N+, N-, N0) of a
// matrix given the list of eigenvalues.
// tol is the tolerance below which an
// eigenvalue will be considered as zero.
fn inertia(ev, tol) {
    var np=0, nz=0, nn=0
    for (e in ev) {
        if (e>tol) np+=1
        else if (e<-tol) nn+=1
        else nz+=1
    }
    return (np, nn, nz)
}

var Ha = la.hessian(m) // Hessian Area
var Hv = lv.hessian(m) // Hessian VolumeEnclosed

var ga = la.gradient(m)
var gv = lv.gradient(m)

var dim = Ha.dimensions()
ga.reshape(dim[0],1)
gv.reshape(dim[0],1)

var lambda = ga.inner(gv)/gv.inner(gv)

var KKT = Matrix([[Ha - lambda*Hv, gv],
                  [gv.transpose(), 0]])

var es = Matrix(KKT).eigensystem() // Compute eigenvalues and eigenvectors
var ev = es[0]

var tol = 1e-4 // Tolerance for zero eigenvalues
print inertia(ev, tol)
```

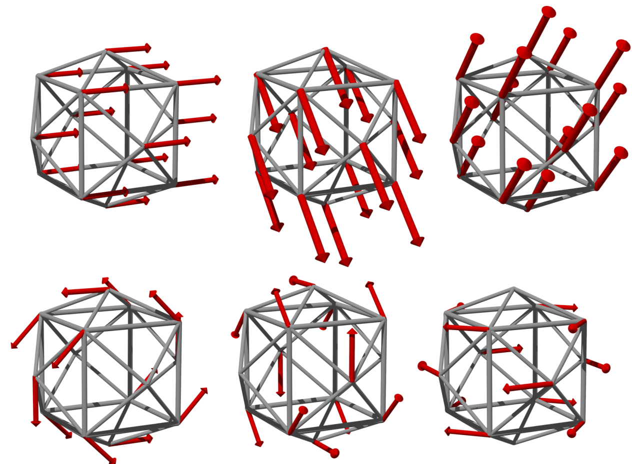
Supplementary Listing 24: Testing the stability of a minimal surface

section) using a MacLaurin series,

$$f = f_0 + \nabla f \cdot \delta x + \frac{1}{2}\delta x \cdot H \cdot \delta x + \ldots,$$

where $H_{ij} = \frac{\partial^2}{\partial x_i \partial x_j}$ is the *Hessian matrix* of the objective function. Since we are at a optimized solution, the linear term $f_1 = \nabla f \cdot \delta x$ is identically zero, and hence the leading term characterizing the change in the objective function is the quadratic term $f_2 = \frac{1}{2}\delta x \cdot H \cdot \delta x$. We'd therefore like to know if the quadratic term f_2 is positive, no matter what direction δx we pick, or whether there are any directions for which the quadratic term could be zero or negative.

A natural way to determine whether f_2 is positive is to compute the eigenvalues h_i and an orthonormal basis of eigenvectors ξ_i from the Hessian matrix. Any arbitrary direction δx can be expanded onto the basis of eigenvectors,



Supplementary Figure 13: Zero modes of the minimal surface. Each panel visualizes an eigenvector associated with a zero eigenvalue; arrows indicate the direction of motion for each vertex along the eigenvector. The top row corresponds to three independent translational motions; the bottom three are rotations about orthogonal axes.

$$\delta x = \sum_i a_i \xi_i,$$

where a_i are real coefficients, and the change in energy written,

$$f_2 = \frac{1}{2}\delta x \cdot H \cdot \delta x = \frac{1}{2}\sum_i a_i^2 h_i,$$

where we exploited the orthonormality of the eigenvectors. Hence f_2 is positive definite if all of the eigenvalues h_i are positive, and our solution is indeed a minimum. If they are not all positive, we can choose a direction that lies within the subspace spanned by the eigenvectors associated with negative eigenvalues, and moving in that direction will reduce the objective function; we have found a solution that corresponds to a saddle point of f.

In principle, we do not even need to know the values of the eigenvalues of the Hessian matrix, simply their sign. This information is encoded in a quantity called the *inertia,* which is a triplet of integers (N_+, N_-, N_0) counting the number of positive, negative and zero eigenvalues of a matrix. According to Sylvester's law of inertia, the inertia is invariant under a change of basis.

If the optimization problem is constrained, as is the case for the minimal surface example, we cannot simply examine the Hessian of the objective function, but must instead consider the Lagrangian,

$$\mathcal{L} = f - \sum_i \lambda_i c_i,$$

where $c_i = 0$ are the constraint functions and λ_i are the Lagrange multipliers associated with the corresponding

constraints. The hessian of the Lagrangian, referred to as the KKT matrix[14], has the following structure,

$$KKT = \begin{pmatrix} \nabla^2 \mathcal{L} & \nabla c \\ \nabla^T c & 0 \end{pmatrix},$$

where the notation ∇ and ∇^2 denote the gradient and Hessian with respect to the original variables of the problem (i.e. excluding the Lagrange multipliers). The stability of the solution can be determined by computing the inertia of the KKT matrix. In contrast to the constrained case, we should expect negative eigenvalues, but the system is stable if the number of negative eigenvalues $N_- \leq N_c$ the number of constraints.

Returning to the minimal surfaces produced in Section III A, the code to perform stability analysis is shown in Supplementary Listing 24 and can be inserted directly after Supplementary Listing 1. Supplementary Listing 24, lines 1-13 define a function, `inertia`, that calculates the inertia from list of eigenvalues. Numerical diagonalization does not yield exact values, so a tolerance must be given. If an eigenvalue $e_i < tol$, then the eigenvalue is considered zero. In lines 15-19 we compute the gradient and hessian of the Area and VolumeEnclosed functionals. The gradient is returned as a $dim \times N_v$ matrix, where dim is the dimension of the space (here 3) and N_v is the number of vertices. Hence in lines 21-23 we reshape these matrices into a column vector.

Since we didn't explicitly use Lagrange multipliers to find the optimized surface, we have to calculate them. To do so, we use the first order optimality criterion,

$$\nabla \mathcal{L} = \nabla f - \sum_i \lambda_i \nabla c_i = 0,$$

and, by taking inner products of this equation with ∇c_j arrive at the linear system,

$$\sum_i \left(\nabla c_i \cdot \nabla c_j\right) \lambda_i = \left(\nabla f \cdot \nabla c_j\right),$$

which can be solved to find the Lagrange multipliers λ_i. Since only one constraint is present, the calculation to do so is particularly simple and occurs in Supplementary Listing 24, line 25.

In Supplementary Listing 24, line 27 we build the KKT matrix, noting that the upper left block $\nabla^2 \mathcal{L} = \nabla^2 f - \sum_i \lambda_i \nabla^2 c_i$. Line 30 computes the eigenvalues and eigenvectors of the KKT matrix, which are returned in a List object: the first component is a List of eigenvalues (we extract this in line 31) and the second is a dense Matrix whose columns are the eigenvectors. In lines 33-34 we then compute the inertia from the list of eigenvalues.

If we apply this analysis to the first unrefined cube, we get the result $(36, 1, 6)$. There are 14 vertices, for a total of 42 degrees of freedom. Since the one constraint is balanced by one negative eigenvalue of the KKT matrix, the system is stable. But what are the six zero eigenvalues? In Supplementary Fig. 13 we visualize the eigenvectors associated with these modes by placing an arrow at each vertex pointing in the direction along the eigenvector[15]. As can be seen, three of these correspond to translating the mesh in 3D in any of three orthogonal directions. The second trio correspond to rotations about three perpendicular axes. These zero modes arise because we can arbitrarily translate and rotate the mesh without changing either its surface area or its enclosed volume—they don't affect the stability.

Refining the mesh successively, we obtain the inertia $(144, 1, 6)$ after refining once, and $(576, 1, 6)$ after refining twice. In each case, the single negative eigenvalue is balanced by the one constraint, and the six residual degrees of freedom of 3D space yield six zero eigenvalues.

Stability analysis of other, more complicated problems can be performed in *Morpho* analogously. While here we have computed the necessary quantities manually, future releases of *Morpho* will automate this process. There are other optimizations that could be performed: Here we computed the inertia from the eigenvalues, but this scales poorly with problem size. Alternative approaches include calculating the LDL^T factorization or another suitable matrix decomposition of the Hessian of the Lagrangian.

---

[1] Smith, W. S., Lazzarato, D. A. & Carette, J. State of the practice for mesh generation and mesh processing software. *Advances in Engineering Software* **100**, 53–71 (2016).

[2] Morpho manual. `https://github.com/morpho-lang/morpho-manual`. Accessed: 4-23-2024.

[3] Morpho developer guide. `https://github.com/Morpho-lang/morpho-devguide`. Accessed: 4-23-2024.

[4] Morpho code. https://github.com/Morpho-lang/morpho.

[5] To join the slack channel, please follow the link provided on the project github. `https://github.com/morpho-lang/morpho`.

[6] Morpho youtube channel. `https://www.youtube.com/@Morpho-lang`. Accessed: 6-26-2024.

[7] Adler, J. H., Andrei, A. S. & Atherton, T. J. Nonlinear Methods for Shape Optimization Problems in Liquid Crystal Tactoids. *arXiv:2310.04022 [Mathematics - Numerical Analysis]* (2023). arXiv:2310.04022.

[8] Example code. https://github.com/Morpho-lang/morpho-paper.

[9] Example code. https://doi.org/10.5281/zenodo.14193814.

[10] For those familiar with Surface Evolver, the relevant file is `cube.fe`.

[11] For a detailed discussion see W. H. Press et al. *Numerical Recipes, 3rd Ed.* (Cambridge University Press) Section

10.6.

[12] S.G. Prasath *et al. "Shapes of a filament on the surface of a bubble"* Proc. R. Soc. A 477 20210353 (2021).

[13] Cousins, J. R., Duffy, B. R., Wilson, S. K. & Mottram, N. J. Young and young–laplace equations for a static ridge of nematic liquid crystal, and transitions between equilibrium states. *Proceedings of the Royal Society A* **478**, 20210849 (2022).

[14] The KKT matrix is sometimes referred to as the *bordered hessian* in the context of stability analysis.

[15] Code to do this is included in the provided listing in [8].