Morpho
0.5

October 19, 2021

Contents

1 Overview **2**

2 Installing *Morpho* **3**

3 Using *Morpho* **5**

4 Tutorial **7**

Reference **16**

5 Language **17**

6 Data Types **28**

7 Computational Geometry **33**

8 I/O **40**

9 Modules **42**

Chapter 1

Overview

Morpho aims to solve the following class of problems. Consider a functional,

$$F = \int_C f(q, \nabla q, \nabla^2 q, ...) d^n x + \int_{\partial C} g(q, \nabla q, \nabla^2 q, ...) d^{n-1} x,$$

where q represents a set of fields defined on a manifold C that could include scalar, vector, tensor or other quantities and their derivatives $\nabla^n q$. The functional includes terms in the bulk and on the boundary ∂C and might also include geometric properties of the manifold such as local curvatures. This functional is to be minimized from an initial guess $\{C_0, q_0\}$ with respect to the fields q and the shape of the manifold C. Global and local constraints may be imposed both on C and q.

Morpho is an object-oriented environment: all components of the problem, including the computational domain, fields, functionals etc. are all represented as objects that interact with one another. Much of the effort in writing a *morpho* program involves creating and manipulating these objects. The environment is flexible, modular, and users can easily create new kinds of object, or entirely change how *morpho* works.

Chapter 2

Installing *Morpho*

Morpho is hosted on a publicly available github repository `https://github.com/Morpho-lang/morpho`. *Morpho* is presently installed from source; streamlined installation will be provided in future releases. Instructions for different platforms are provided below.

2.1 Dependencies

Morpho leverages a few libraries to provide certain functionality:

glfw is used to provide gui functionality for an interactive visualization application, `morphoview`.

blas/lapack are used for dense linear algebra.

suitesparse is used for sparse linear algebra.

2.2 MacOS

1. Install the Homebrew package manager, following instructions on the homebrew site.
2. Install dependencies. Open the Terminal application and type:

   ```
   brew update
   brew install glfw suite-sparse
   ```

3. Obtain the source by cloning the github public repository:

   ```
   git clone https://github.com/Morpho-lang/morpho.git
   ```

4. Navigate to the `morpho5` folder within the downloaded repository and build the application[1]

   ```
   cd morpho/morpho5
   make install
   ```

5. Navigate to the `morphoview` folder and build the viewer application

   ```
   cd morpho/morpho5
   make install
   ```

[1]Some users may need to use `sudo make install`

6. Check that the application works by typing

```
morpho5
```

2.3 Linux

Note that the build script places `morpho5` and `morphoview` in the `/usr/local` file structure; this can easily be changed if a different location is preferred.

1. Install morpho's dependencies using your distribution's package manager (or manually if you prefer). For example, on Ubuntu you would type:

```
sudo apt install libglfw3
sudo apt install libsuitesparse-dev
sudo apt install liblapacke
```

2. Obtain the source by cloning the github public repository:

```
git clone https://github.com/Morpho-lang/morpho.git
```

3. Navigate to the `morpho5` folder within the downloaded repository and build the application:

```
cd morpho/morpho5
sudo make -f Makefile.linux install
```

4. Navigate to the `morphoview` folder and build the viewer application:

```
cd ../morphoview
sudo make -f Makefile.linux install
```

5. Check that the application works by typing

```
morpho5
```

Chapter 3

Using *Morpho*

Morpho is a command line application, like `python` or `lua`. It can be used to run scripts or programs, which are generally given the *.morpho* file extension, or run interactively responding to user commands.

3.1 Running a program

To run a program, simply run morpho with the name of the file,

```
morpho5 script.morpho
```

3.2 Interactive mode

To use *morpho* interactively, simply load the *Terminal* application (or equivalent on your system) and type

```
morpho5
```

As shown in Fig. 3.2.1, you'll be greeted by a brief welcome and a prompt > inviting you to enter *morpho* commands. For now, try a classic:

```
print "Hello World"
```

which will display `Hello World` as output. More information about the *morpho* language is provided in the Reference section, especially chapter 5; if you're familiar with C-like languages such as C, C++, Java, Javascript, etc. things should be quite familiar.

To assist the user, the contents of the reference manual are available to the user in interactive mode as online help. To get help, simply type:

```
help
```

or even more briefly,

```
?
```

to see the list of main topics. To find help on a particular topic, for example `for` loops, simply type the topic name afterwards:

```
? for
```

Once you're done using *morpho*, simply type

```
quit
```

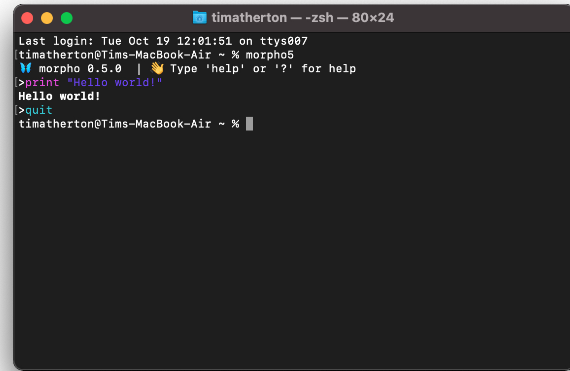

Figure 3.2.1: Using *morpho* interactively from the command line.

to exit the program and return to the shell.

The interactive environment has a few other useful features to assist the user:

- **Autocomplete.** As you type, *morpho* will show you any suggested commands that it thinks you're trying to enter. For example, if you type `v` the command line will show the `var` keyword. To accept the suggestion, press the tab key. Multiple suggestions may be available; use the up and down arrow keys to rotate through them.

- **Command history.** Use the arrow keys to retrieve previously entered commands. You may then edit them before running them.

- **Line editing.** As you're typing a command, use the left and right arrows to move the cursor around; you can insert new characters at the cursor just by typing them or delete characters with the `delete` key. Hold down the `shift` key as you use the left and right arrow keys to select text; you can then use `Ctrl-C` to copy and `Ctrl-V` to paste. `Ctrl-A` moves to the start of the line and `Ctrl-E` the end.

Chapter 4

Tutorial

To illustrate how to use *morpho*, we will solve a problem involving nematic liquid crystals (NLCs), fluids composed of long, rigid molecules that possess a local average molecular orientation described by a unit vector field $\hat{\mathbf{n}}$. Droplets of NLC immersed in a host isotropic fluid such as water are called *tactoids* and, unlike droplets of, say, oil in water that form spheres, tactoids can adopt elongated shapes.

The functional to be minimized, the free energy of the system, is quite complex,

$$F = \underbrace{\frac{1}{2}\int_C K_{11}\left(\nabla\cdot\mathbf{n}\right)^2 + K_{22}(\mathbf{n}\cdot\nabla\times\mathbf{n})^2 + K_{33}\left|\mathbf{n}\times\nabla\times\mathbf{n}\right|^2 dA}_{\text{Liquid crystal elastic energy}} + \underbrace{\sigma\int dl}_{s.t.} - \underbrace{\frac{W}{2}\int(\mathbf{n}\cdot\mathbf{t})^2\,dl}_{\text{anchoring}} \tag{4.0.1}$$

where the three terms include **liquid crystal elasticity** that drives elongation of the droplet, **surface tension** (s.t.) that opposes lengthening of the boundary and an **anchoring term** that imposes a preferred orientation at the boundary. We need a local constraint, $\mathbf{n}\cdot\mathbf{n} = 1$, and will also impose a constraint on the volume of the droplet. For simplicity, we'll solve this problem in 2D. The complete code for this tutorial example is contained in the `examples/tactoid` folder in the repository.

4.1 Importing modules

Morpho is a modular system and hence we typically begin our program by telling *morpho* the modules we need so that they're available for us to use. To do so, we use the `import` keyword followed by the name of the module:

```
import meshtools
import optimize
import plot
```

We can also use the `import` keyword to import additional program files to assist in modularizing large programs. These are the modules we'll use for this example:

Module	Purpose
meshtools	Utility code to create and refine meshes
optimize	Perform optimization
plot	Visualize results

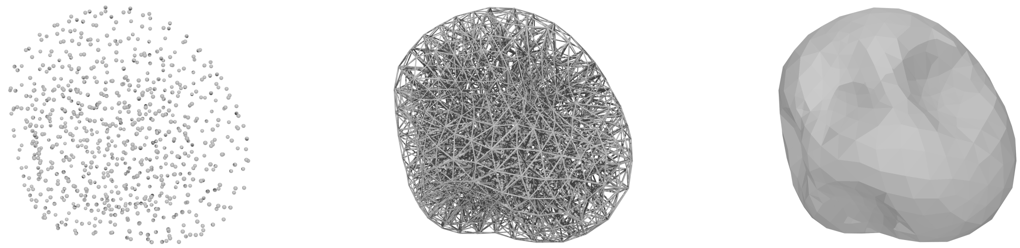

Figure 4.2.1: A *Mesh* object contains different kinds of element. In this example, the mesh contains points, lines and area elements referred to by their *grade*.

4.2 Creating the initial mesh

Meshes are discretized regions of space. The very simplest region we can imagine is a *point* or *vertex* described by a set of coordinates $(x_1, x_2, ..., x_D)$ where the number of coordinates D defines the dimensionality of the space that the manifold is said to be *embedded* in. From more than one point, we can start constructing more complex regions. First, between two points we can imagine fixing an imaginary ruler and drawing a straight line or *edge* between them. Three points define a plane, and also a triangle; we can therefore identify the two dimensional area of the plane bounded by the triangle as a *face*, as in the face of a polyhedron. Using four points, we can define the volume bounded by a tetrahedron. Each of these **elements** has a different dimensionality—called a *grade*—and a complete `Mesh` may contain elements of many different grades as shown in Fig. 4.2.1.

Morpho provides a number of ways of creating a mesh. One can load a mesh from a file, build one manually from a set of points, create one from a polyhedron, or from the level set (contours) of a function.

For this example, we'll use a predefined mesh file `disk.mesh`. To create a Mesh object from this file, we call the *Mesh* function with the file name:

```
var m = Mesh("disk.mesh")
```

Here, the **var** keyword tells morpho to create a new variable *m*, which now refers to the newly created *Mesh* object. The initial mesh is depicted in Fig. 4.2.2; we'll provide the code to perform the visualization in section 4.7.

If you open the file `disk.mesh`, which you can find in the same folder as `tactoid.morpho`, you'll find it has a simple human readable format:

```
vertices

1 -1. 0. 0
2 -0.951057 -0.309017 0
...

edges
1 8 2
2 2 4
...

faces
1 8 2 4
```

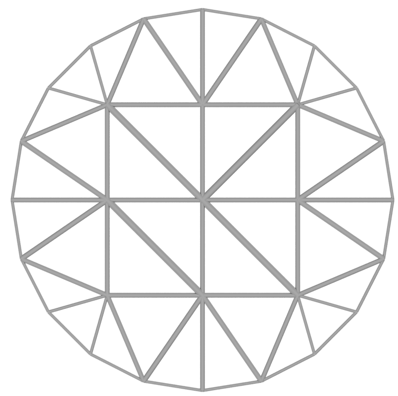

Figure 4.2.2: The initial mesh, loaded from `disk.mesh`.

```
2 8 4 6
...
```

The file is broken into sections, each describing elements of a different grade. Each line begins either with a section delimiter such as *vertices*, *edges* or *faces*, or with an id. Vertices are then defined by a set of coordinates; edges and faces are defined by providing the respective vertex ids.

4.3 Selections

Sometimes, we want to refer to specific parts of a `Mesh` object: elements that match some criterion, for example. `Selection` objects enable us to do this. Because selecting the boundary is a very common activity, the `Selection` constructor function takes an optional argument to do this:

```
// Initial boundary selection
var bnd=Selection(m, boundary=true)
```

By default, only the boundary elements are included in the `Selection`. For a mesh with at most grade 2 elements (facets), the boundaries are grade 1 elements (lines); for a mesh with grade 3 elements (volumes), the boundaries are grade 2 elements (facets). Quite often we want the vertices themselves as well, so we can call a method to achieve that:

```
bnd.addgrade(0)
```

Once a `Selection` has been created, it can be helpful to visualize it to ensure the correct elements are selected. We'll talk more about visualization in section 4.7, but for now the line

```
Show(plotselection(m, bnd, grade=1))
```

shows a visualization of the mesh with the selected grade 1 elements shaded red as displayed in Fig. 4.3.1.

4.4 Fields

Having created our initial the computational domain, we will now create a `Field` object representing the director field $\mathbf{n}$:

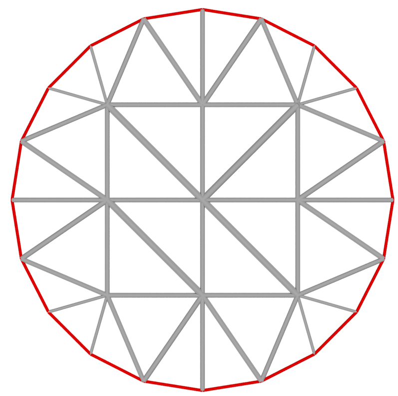

Figure 4.3.1: Selecting the boundary of the mesh.

```
var nn = Field(m, Matrix([1,0,0]))
```

As with the `Mesh` object earlier, we declare a variable, *nn*, to refer to the `Field` object. We have to provide two arguments to `Field`: the `Mesh` object on which the `Field` is defined, and something to initialize it. Here, we want the initial director to have a spatially uniform value, so we can just provide `Field` a constant `Matrix` object. By default, *morpho* stores a copy of this matrix on each vertex in the mesh; Fields can however store information on elements of any grade (and store both more than one quantity per grade and information on multiple grades at the same time).

It's possible to initialize a `Field` with spatially varying values by providing an *anonymous function* to `Field` like this:

```
var phi = Field(m, fn (x,y,z) x^2+y^2)
```

Here, *phi* is a scalar field that takes on the value $x^2 + y^2$. The **fn** keyword is used to define functions.

4.5 Defining the problem

We now turn to setting up the problem. Each term in the energy functional (4.0.1) is represented by a corresponding *functional* object, which acts on a `Mesh` (and possibly a `Field`) to calculate an integral quantity such as an energy; Functional objects are also responsible for calculating gradients of the energy with respect to vertex positions and components of Fields.

Let's take the terms in (4.0.1) one by one: To represent the nematic elasticity we create a `Nematic` object:

```
var lf=Nematic(nn)
```

The surface tension term involves the length of the boundary, so we need a `Length` object:

```
var lt=Length()
```

The anchoring term doesn't have a simple built in object type, but we can use a general `LineIntegral` object to achieve the correct result.

```
var la=LineIntegral(fn (x, n) n.inner(tangent())^2, nn)
```

Notice that we have to supply a function—the integrand—which will be called by `LineIntegral` when it evaluates the integral. Integrand functions are called with the local coordinates first (as a `Matrix` object representing a column vector) and then the local interpolated value of any number of `Fields`. We also make use of the special function `tangent()` that locally returns a local tangent to the line.

We also need to impose constraints. Any *functional* object can be used equally well as an energy or a constraint, and hence we create a `NormSq` (norm-squared) object that will be used to implement the local unit vector constraint on the director field:

```
var ln=NormSq(nn)
```

and an `Area` object for the global constraint:

```
var laa=Area()
```

Now we have a collection of functional objects that we can use to define the problem. So far, we haven't specified which functionals are energies and which are constraints; nor have we specified which parts of the mesh the functionals are to be evaluated over. All that information is collected in an `OptimizationProblem` object, which we will now create:

```
// Set up the optimization problem
var W = 1
var sigma = 1

var problem = OptimizationProblem(m)
problem.addenergy(lf)
problem.addenergy(la, selection=bnd, prefactor=-W/2)
problem.addenergy(lt, selection=bnd, prefactor=sigma)
problem.addconstraint(laa)
problem.addlocalconstraint(ln, field=nn, target=1)
```

Notice that some of these functionals only act on a selection such as the boundary and hence we use the optional *selection* parameter to specify this. We can also specify the prefactor of the functional.

4.6 Performing the optimization

We're now ready to perform the optimization, for which we need an `Optimizer` object. These come in two flavors: a `ShapeOptimizer` and a `FieldOptimizer` that respectively act on the shape and a field. We create them with the problem and quantity they're supposed to act on:

```
// Create shape and field optimizers
var sopt = ShapeOptimizer(problem, m)
var fopt = FieldOptimizer(problem, nn)
```

Having created these, we can perform the optimizion by calling the `linesearch` method with a specified number of iterations for each:

```
// Optimization loop
for (i in 1..100) {
        fopt.linesearch(20)
        sopt.linesearch(20)
}
```

Each iteration of a linesearch evolves the field (or shape) down the gradient of the target functional, subject to constraints, and finds an optimal stepsize to reduce the value of the functional. Here, we alternate between optimizing the field and optimizing the shape, performing twenty iterations of each, and overall do this one hundred times. These numbers have been chosen rather arbitrarily, and if you look at the output you will notice that *morpho* doesn't always execute twenty iterations of each. Rather, at each iteration it checks to see if the change in energy satisfies,

$$|E| < \epsilon,$$

or,

$$\left|\frac{\Delta E}{E}\right| < \epsilon$$

where the value of ϵ, the convergence tolerance can be changed by setting the `etol` property of the Optimizer object:

```
sopt.etol = 1e-7 // default value is 1e-8
```

Some other properties of an Optimizer that may be useful for the user to adjust are as follows:

Property	Default value	Purpose
`etol`	1×10^{-8}	Energy tolerance (relative error)
`ctol`	1×10^{-10}	Constraint tolerance (how well are constraints satisfied)
`stepsize`	0.1	Stepsize for relax (changed by linesearch)
`steplimit`	0.5	Largest stepsize a linesearch can take
`maxconstraintsteps`	20	Number of steps the optimizer may take to ensure constraints are satisfied
`quiet`	false	Whether to print output as the optimization happens

4.7 Visualizing results

Morpho provides a highly flexible graphics system, with an external viewer application *morphoview*, to enable rich visualizations of results. Visualizations typically involve one or more Graphics objects, which act as a container for graphical elements to be displayed. Various *graphics primitives*, such as spheres, cylinders, arrows, tubes, etc. can be added to a Graphics object to make a drawing.

We are now ready to visualize the results of the optimization. First, we'll draw the mesh. Because we're interested in seeing the mesh structure, we'll draw the edges (i.e. the grade 1 elements). The function to do this is provided as part of the plot module that we imported in section 4.1:

```
var g=plotmesh(m, grade=1)
```

Next, we'll create a separate Graphics object that contains the director. Since the director $\mathbf{n}$ is a unit vector field, and the sign is not significant (the nematic elastic energy is actually invariant under $\mathbf{n} \to -\mathbf{n}$), an appropriate way to display a single director is as a cylinder oriented along $\mathbf{n}$. We will therefore make a helper function that creates a Graphics object and draws such a cylinder at every mesh point:

```
// Function to visualize a director field
// m - the mesh
// nn - the director Field to visualize
// dl - scale the director
fn visualize(m, nn, dl) {
  var v = m.vertexmatrix()
  var nv = m.count() // Number of vertices
```

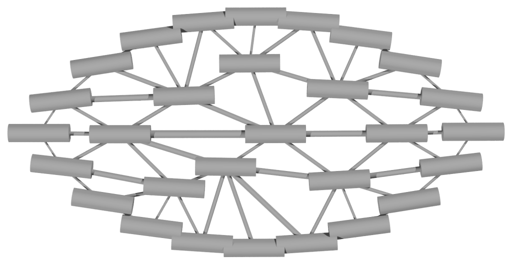

Figure 4.7.1: Optimized mesh and director field.

```
  var g = Graphics() // Create a graphics object
  for (i in 0...nv) {
    var x = v.column(i) // Get the ith vertex
        // Draw a cylinder aligned with nn at this vertex
    g.display(Cylinder(x-nn[i]*dl, x+nn[i]*dl, aspectratio=0.3))
  }
  return g
}
```

Once we've defined this function, we can use it:

```
var gnn=visualize(m, nn, 0.2)
```

The variables g and gnn now refer to two separate Graphics objects. We can combine them using the $+$ operator, and display them with *morphoview* like so:

```
var gdisp = g+gnn
Show(gdisp)
```

The resulting visualization is shown in Fig. 4.7.1.

4.8 Refinement

We have now solved our first shape optimization problem, and the complete problem script is provided in the `examples/tutorial` folder inside the git repository as `tutorial1.morpho`. The result we have obtained in Fig. 4.7.1 is, however, a very coarse, low resolution solution comprising only a relatively small number of elements. To gain an improved solution, we need to *refine* our mesh. Because modifying the mesh also requires us to update other data structures like fields and selections, a special `MeshRefiner` object is used to perform the refinement.

To perform refinement we:

1. Create a MeshRefiner object, providing it a list of all the `Mesh`, `Field` and `Selection` objects (i.e. the mesh and objects that directly depend on it) that need to be updated:

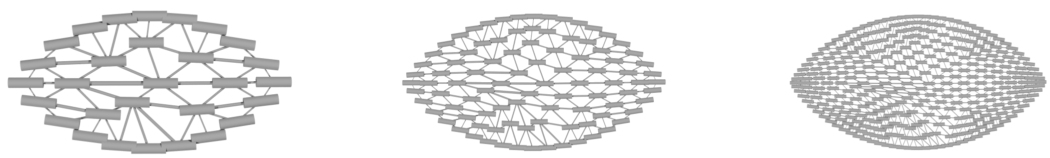

Figure 4.8.1: Optimized mesh and director field at three successive levels of refinement.

```
var mr=MeshRefiner([m, nn, bnd]) // Set the refiner up
```

2. Call the `refine` method on the `MeshRefiner` object to actually perform the refinement. This method returns a `Dictionary` object that maps the old objects to potentially newly created ones.

   ```
   var refmap=mr.refine() // Perform the refinement
   ```

3. Tell any other objects that refer to the mesh, fields or selections to update their references using `refmap`. For example, `OptimizationProblem` and `Optimizer` objects are typically updated at this step.

   ```
   for (el in [problem, sopt, fopt]) el.update(refmap) // Update the problem
   ```

4. Update our own references

   ```
   m=refmap[m]; nn=refmap[nn]; bnd=refmap[bnd] // Update variables
   ```

We insert this code after our optimization section, which causes *morpho* to successively optimize and refine[1]. The resulting optimized shapes are displayed in Fig. 4.8.1.

```
// Optimization loop
var refmax = 3
for (refiter in 1..refmax) {
  print "===Refinement level ${refiter}==="
  for (i in 1..100) {
    fopt.linesearch(20)
    sopt.linesearch(20)
  }

  if (refiter==refmax) break

  // Refinement
  var mr=MeshRefiner([m, nn, bnd]) // Set the refiner up
  var refmap=mr.refine() // Perform the refinement
  for (el in [problem, sopt, fopt])
  el.update(refmap) // Update the problem
  m=refmap[m]; nn=refmap[nn]; bnd=refmap[bnd] // Update variables
}
```

[1]The complete code including refinement is in `examples/tutorial` folder inside the git repository as `tutorial2.morpho`

4.9 Next steps

Having completed this tutorial, you may wish to explore the effect of changing some of the parameters in the file. What happens if you change `sigma` and W, the coefficients in front of the terms in the energy? What happens if you take a different number of steps? Or change properties of the Optimizers like `stepsize` and `steplimit`?

You should look at other example files provided in the `examples` folder of the git repository. The remainder of the manual provides a detailed reference for *morpho* functionality, and a complete description of the scripting language.

Reference

Chapter 5

Language

5.1 Syntax

Morpho provides a flexible object oriented language similar to other languages in the C family (like C++, Java and Javascript) with a simplified syntax.

Morpho programs are stored as plain text with the .morpho file extension. A program can be run from the command line by typing

```
morpho program.morpho
```

5.2 Comments

Two types of comment are available. The first type is called a 'line comment' whereby text after `//` on the same line is ignored by the interpreter.

```
a.dosomething() // A comment
```

Longer 'block' comments can be created by placing text between `/*` and `*/`. Newlines are ignored

```
/* This
   is
   a longer comment */
```

In contrast to C, these comments can be nested

```
/* A nested /* comment */ */
```

enabling the programmer to quickly comment out a section of code.

5.3 Symbols

Symbols are used to refer to named entities, including variables, classes, functions etc. Symbols must begin with a letter or underscore _ as the first character and may include letters or numbers as the remainder. Symbols are case sensitive.

```
asymbol
_alsoasymbol
another_symbol
EvenThis123
```

```
YET_ANOTHER_SYMBOL
```

Classes are typically given names with an initial capital letter. Variable names are usually all lower case.

5.4 Newlines

Strictly, morpho ends statements with semicolons like C, but in practice these are usually optional and you can just start a new line instead. For example, instead of

```
var a = 1; // The ; is optional
```

you can simply use

```
var a = 1
```

If you want to put several statements on the same line, you can separate them with semicolons:

```
var a = 1; print a
```

There are a few edge cases to be aware of: The morpho parser works by accepting a newline anywhere it expects to find a semicolon. To split a statement over multiple lines, signal to morpho that you plan to continue by leaving the statement unfinished. Hence, do this:

```
print a +
      1
```

rather than this:

```
print a   // < Morpho thinks this is a complete statement
      + 1 // < and so this line will cause a syntax error
```

5.5 Booleans

Comparison operations like ==, < and >= return `true` or `false` depending on the result of the comparison. For example,

```
print 1==2
```

prints `false`. The constants `true` or `false` are provided for you to use in your own code:

```
return true
```

5.6 Nil

The keyword `nil` is used to represent the absence of an object or value.

Note that in `if` statements, a value of `nil` is treated like `false`.

```
if (nil) {
    // Never executed.
}
```

5.7 Blocks

Code is divided into *blocks*, which are delimited by curly brackets like this:

```
{
  var a = "Hello"
  print a
}
```

This syntax is used in function declarations, loops and conditional statements.

Any variables declared within a block become *local* to that block, and cannot be seen outside of it. For example,

```
var a = "Foo"
{
  var a = "Bar"
  print a
}
print a
```

would print “Bar” then “Foo”; the version of `a` inside the code block is said to *shadow* the outer version.

5.8 Precedence

Precedence refers to the order in which morpho evaluates operations. For example,

```
print 1+2*3
```

prints `7` because `2*3` is evaluated before the addition; the operator `*` is said to have higher precedence than `+`.

You can always modify the order of evaluation by using brackets:

```
print (1+2)*3 // prints 9
```

5.9 Print

The `print` keyword is used to print information to the console. It can be followed by any value, e.g.

```
print 1
print true
print a
print "Hello"
```

5.10 Variables

Variables are defined using the `var` keyword followed by the variable name:

```
var a
```

Optionally, an initial assignment may be given:

```
var a = 1
```

Variables defined in a block of code are visible only within that block, so

```
var greeting = "Hello"
{
    var greeting = "Goodbye"
    print greeting
}
print greeting
```

will print
Goodbye Hello
Multiple variables can be defined at once by separating them with commas

```
var a, b=2, c[2]=[1,2]
```

where each can have its own initializer (or not).

5.11 Indexing

Morpho provides a number of collection objects, such as `List`, `Range`, `Array`, `Dictionary`, `Matrix` and `Sparse`, that can contain more than one value. Index notation (sometimes called subscript notation) is used to access elements of these objects.

To retrieve an item from a collection, you use the `[` and `]` brackets like this:

```
var a = List("Apple", "Bag", "Cat")
print a[0]
```

which prints *Apple*. Note that the first element is accessed with `0` not `1`.
Similarly, to set an entry in a collection, use:

```
a[0]="Adder"
```

which would replaces the first element in `a` with `"Adder"`.
Some collection objects need more than one index,

```
var a = Matrix([[1,0],[0,1]])
print a[0,0]
```

and others such as `Dictionary` use non-numerical indices,

```
var b = Dictionary()
b["Massachusetts"]="Boston"
b["California"]="Sacramento"
```

as in this dictionary of state capitals.

5.12 Control Flow

Control flow statements are used to determine whether and how many times a selected piece of code is executed. These include:

- `if` - Selectively execute a piece of code if a condition is met.
- `else` - Execute a different block of code if the test in an `if` statement fails.

- `for` - Repeatedly execute a section of code with a counter
- `while` - Repeatedly execute a section of code while a condition is true.

5.13 If

If allows you to selectively execute a section of code depending on whether a condition is met. The simplest version looks like this:

```
if (x<1) print x
```

where the body of the loop, `print x`, is only executed if x is less than 1. The body can be a code block to accommodate longer sections of code:

```
if (x<1) {
    ... // do something
}
```

If you want to choose between two alternatives, use `else`:

```
if (a==b) {
    // do something
} else {
    // this code is executed only if the condition is false
}
```

You can even chain multiple tests together like this:

```
if (a==b) {
    // option 1
} else if (a==c) {
    // option 2
} else {
    // something else
}
```

5.14 While

While loops repeat a section of code while a condition is true. For example,

```
var k=1
while (k <= 4) { print k; k+=1 }
       ^cond    ^body
```

prints the numbers 1 to 4. The loop has two sections: `cond` is the condition to be executed and `body` is the section of code to be repeated.

Simple loops like the above example, especially those that involve counting out a sequence of numbers, are more conveniently written using a `for` loop,

```
for (k in 1..4) print k
```

Where `while` loops can be very useful is where the state of an object is being changed in the loop, e.g.

```
var a = List(1,2,3,4)
while (a.count()>0) print a.pop()
```

which prints 4,3,2,1.

5.15 Do

A `do`...`while` loop repeats code while a condition is true—similar to a `while` loop—but the test happens at the end:

```
var k=1
do {
  print k;
  k+=1
} while (k<5)
```

which prints 1,2,3,4
Hence this type of loop executes at least one interation

5.16 For

For loops allow you to repeatedly execute a section of code. They come in two versions: the simpler version looks like this,

```
for (var i in 1..5) print i
```

which prints the numbers 1 to 5 in turn. The variable `i` is the *loop variable*, which takes on a different value each iteration. `1..5` is a range, which denotes a sequence of numbers. The *body* of the loop, `print i`, is the code to be repeatedly executed.

Morpho will implicitly insert a `var` before the loop variable if it's missing, so this works too:

```
for (i in 1..5) print i
```

If you want your loop variable to count in increments other than 1, you can specify a stepsize in the range:

```
for (i in 1..5:2) print i
               ^step
```

Ranges need not be integer:

```
for (i in 0.1..0.5:0.1) print i
```

You can also replace the range with other kinds of collection object to loop over their contents:

```
var a = Matrix([1,2,3,4])
for (x in a) print x
```

Morpho iterates over the collection object using an integer *counter variable* that's normally hidden. If you want to know the current value of the counter (e.g. to get the index of an element as well as its value), you can use the following:

```
var a = [1, 2, 3]
for (x, i in a) print "${i}: ${x}"
```

Morpho also provides a second form of `for` loop similar to that in C:

```
for (var i=0; i<5; i+=1) { print i }
     ^start   ^test ^inc.  ^body
```

which is executed as follows: start: the variable `i` is declared and initially set to zero. test: before each iteration, the test is evaluated. If the test is `false`, the loop terminates. body: the body of the loop is executed. inc: the variable `i` is increased by 1.

You can include any code that you like in each of the sections.

5.17 Break

Break is used inside loops to finish the loop early. For example

```
for (i in 1..5) {
    if (i>3) break // --.
    print i        //   | (Once i>3)
}                  //   |
...                // <-'
```

would only print 1,2 and 3. Once the condition `i>3` is true, the `break` statement causes execution to continue after the loop body.

Both `for` and `while` loops support break.

5.18 Continue

Continue is used inside loops to skip over the rest of an iteration. For example

```
for (i in 1..5) {     // <-.
    print "Hello"           |
    if (i>3) continue // --'
    print i
}
```

prints "Hello" five times but only prints 1,2 and 3. Once the condition `i>3` is true, the `continue` statement causes execution to transfer to the start of the loop body.

Traditional `for` loops also support `continue`:

```
                // v increment
for (var i=0; i<5; i+=1) {
    if (i==2) continue
    print i
}
```

Since `continue` causes control to be transferred *to the increment section* in this kind of loop, here the program prints 0..4 but the number 2 is skipped.

Use of `continue` with `while` loops is possible but isn't recommended as it can easily produce an infinite loop!

```
var i=0
while (i<5) {
    if (i==2) continue
    print i
    i+=1
```

```
}
```

In this example, when the condition `i==2` is `true`, execution skips back to the start, but `i` *isn't* incremented. The loop gets stuck in the iteration `i==2`.

5.19 Functions

A function in morpho is defined with the `fn` keyword, followed by the function's name, a list of parameters enclosed in parentheses, and the body of the function in curly braces. This example computes the square of a number:

```
fn sqr(x) {
  return x*x
}
```

Once a function has been defined you can evaluate it like any other morpho function.

```
print sqr(2)
```

5.20 Return

The `return` keyword is used to exit from a function, optionally passing a given value back to the caller. `return` can be used anywhere within a function. The below example calculates the `n` th Fibonacci number,

```
fn fib(n) {
  if (n<2) return n
  return fib(n-1) + fib(n-2)
}
```

by returning early if `n<2`, otherwise returning the result by recursively calling itself.

5.21 Classes

Classes are defined using the `class` keyword followed by the name of the class. The definition includes methods that the class responds to. The special `init` method is called whenever an object is created.

```
class Cake {
    init(type) {
        self.type = type
    }

    eat() {
        print "A delicious "+type+" cake"
    }
}
```

Objects are created by calling the class as if it was a function:

```
var c = Cake("carrot")
```

Methods are called using the `.` operator:

```
c.eat()
```

5.22 Self

The `self` keyword is used to access an object's properties and methods from within its definition.

```
class Vehicle {
  init (type) { self.type = type }
}
```

5.23 Super

The keyword `super` allows you to access methods provided by an object's superclass rather than its own. This is particularly useful when the programmer wants a class to extend the functionality of a parent class, but needs to make sure the old behavior is still maintained.

For example, consider the following pair of classes:

```
class Lunch {
    init(type) { self.type=type }
}

class Soup < Lunch {
    init(type) {
        print "Delicious soup!"
        super.init(type)
    }
}
```

The subclass Soup uses `super` to call the original initializer.

5.24 Modules

Morpho is extensible and provides a convenient module system that works like standadrd libraries in other languages. Modules may define useful variables, functions and classes, and can be made available using the `import` keyword. For example,

```
import color
```

loads the `color` module that provides functionality related to color.

You can create your own modules; they're just regular morpho files that are stored in a standard place. On UNIX platforms, this is `/usr/local/share/morpho/modules`.

5.25 Import

Import provides access to the module system and including code from multiple source files.

To import code from another file, use import with the filename:

```
import "file.morpho"
```

which immediately includes all the contents of `"file.morpho"`. Any classes, functions or variables defined in that file can now be used, which allows you to divide your program into multiple source files.

Morpho provides a number of built in modules–and you can write your own–which can be loaded like this:

```
import color
```

which imports the `color` module.

You can selectively import symbols from a modules by using the `for` keyword:

```
import color for HueMap, Red
```

which imports only the `HueMap` class and the `Red` variable.

5.26 Help

Morpho provides an online help system. To get help about a topic called `topicname`, type

```
help topicname
```

A list of available topics is provided below and includes language keywords like `class`, `fn` and `for`, built in classes like `Matrix` and `File` or information about functions like `exp` and `random`.

Some topics have additional subtopics: to access these type

```
help topic subtopic
```

For example, to get help on a method for a particular class, you could type

```
help Classname.methodname
```

Note that `help` ignores all punctuation.

You can also use `?` as a shorthand synonym for `help`

```
? topic
```

A useful feature is that, if an error occurs, simply type `help` to get more information about the error.

5.27 Builtin functions

Morpho provides a number of built-in functions.

5.28 arctan

Returns the arctangent of an input angle. You can use one argument:

```
print arctan(0) // expect: 0
```

or to get the angle in the correct quadrant, use two arguments:

```
print arctan(x, y)
```

Note the order `x`, `y` differs from some other languages.

5.29 isnil

Returns `true` if a value is `nil` or `false` otherwise.

5.30 isint

Returns `true` if a value is an integer or `false` otherwise.

5.31 isfloat

Returns `true` if a value is a floating point number or `false` otherwise.

5.32 isbool

Returns `true` if a value is a boolean or `false` otherwise.

5.33 isobject

Returns `true` if a value is an object or `false` otherwise.

5.34 isstring

Returns `true` if a value is a string or `false` otherwise.

5.35 isclass

Returns `true` if a value is a class or `false` otherwise.

5.36 isrange

Returns `true` if a value is a range or `false` otherwise.

5.37 isdictionary

Returns `true` if a value is a dictionary or `false` otherwise.

5.38 islist

Returns `true` if a value is a list or `false` otherwise.

5.39 isarray

Returns `true` if a value is an array or `false` otherwise.

5.40 ismatrix

Returns `true` if a value is a matrix or `false` otherwise.

5.41 issparse

Returns `true` if a value is a sparse matrix or `false` otherwise.

Chapter 6

Data Types

6.1 Array

Arrays are collection objects that can have any number of indices. Their size is set when they are created:

```
var a[5]
var b[2,2]
var c[nv,nv,nv]
```

Values can be retrieved with appropriate indices:

```
print a[0,0]
```

Any morpho value can be stored in an array element

```
a[0,0] = [1,2,3]
```

6.2 Dimensions

Get the dimensions of an Array object:

```
var a[2,2]
print a.dimensions() // expect: [ 2, 2 ]
```

6.3 Dictionary

Dictionaries are collection objects that associate a unique *key* with a particular *value*. Keys can be any kind of morpho value, including numbers, strings and objects.

An example dictionary mapping states to capitals:

```
var dict = { "Massachusetts" : "Boston",
             "New York" : "Albany",
             "Vermont" : "Montpelier" }
```

Lookup values by a given key with index notation:

```
print dict["Vermont"]
```

You can change the value associated with a key, or add new elements to the dictionary like this:

```
dict["Maine"]="Augusta"
```

Create an empty dictionary using the `Dictionary` constructor function:

```
var d = Dictionary()
```

Loop over keys in a dictionary:

```
for (k in dict) print k
```

6.4 List

Lists are collection objects that contain a sequence of values each associated with an integer index.

Create a list like this:

```
var list = [1, 2, 3]
```

Lookup values using index notation:

```
list[0]
```

You can change list entries like this:

```
list[0] = "Hello"
```

Create an empty list:

```
var list = []
```

Loop over elements of a list:

```
for (i in list) print i
```

6.5 Append

Adds an element to the end of a list:

```
list = []
list.append("Foo")
```

6.6 Insert

Inserts an element into a list at a specified index:

```
list = [1,2,3]
list.insert(1, "Foo")
print list // prints [ 1, Foo, 2, 3 ]
```

6.7 Pop

Remove the last element from a list, returning the element removed:

```
print list.pop()
```

If an integer argument is supplied, returns and removes that element:

```
var a = [1,2,3]
print a.pop(1) // prints '2'
print a        // prints [ 1, 3 ]
```

6.8 Sort

Sorts a list:

```
list.sort()
```

You can provide your own function to use to compare values in the list

```
list.sort(fn (a, b) a-b)
```

This function should return a negative value if `a<b`, a positive value if `a>b` and `0` if `a` and `b` are equal.

6.9 Order

Returns a list of indices that would, if used in order, would sort a list. For example

```
var list = [2,3,1]
print list.order() // expect: [2,0,1]
```

would produce `[2,0,1]`

6.10 Remove

Remove any occurrences of a value from a list:

```
var list = [1,2,3]
list.remove(1)
```

6.11 ismember

Tests if a value is a member of a list:

```
var list = [1,2,3]
print list.ismember(1) // expect: true
```

6.12 Add

Join two lists together:

```
var l1 = [1,2,3], l2 = [4, 5, 6]
print l1+l2 // expect: [1,2,3,4,5,6]
```

6.13 Matrix

The Matrix class provides support for matrices. A matrix can be initialized with a given size,

```
var a = Matrix(nrows,ncols)
```

where all elements are initially set to zero. Alternatively, a matrix can be created from an array,

```
var a = Matrix([[1,2], [3,4]])
```

You can create a column vector like this,

```
var v = Matrix([1,2])
```

Once a matrix is created, you can use all the regular arithmetic operators with matrix operands, e.g.

```
a+b
a*b
```

The division operator is used to solve a linear system, e.g.

```
var a = Matrix([[1,2],[3,4]])
var b = Matrix([1,2])

print b/a
```

yields the solution to the system a*x = b.

6.14 Range

Ranges represent a sequence of numerical values. There are two ways to create them depending on whether the upper value is included or not:

```
var a = 1..5  // inclusive version, i.e. [1,2,3,4,5]
var b = 1...5 // exclusive version, i.e. [1,2,3,4]
```

By default, the increment between values is 1, but you can use a different value like this:

```
var a = 1..5:0.5 // 1 - 5 with an increment of 0.5.
```

You can also create Range objects using the appropriate constructor function:

```
var a = Range(1,5,0.5)
```

Ranges are particularly useful in writing loops:

```
for (i in 1..5) print i
```

They can easily be converted to a list of values:

```
var c = List(1..5)
```

To find the number of elements in a Range, use the `count` method

```
print (1..5).count()
```

6.15 Sparse

The Sparse class provides support for sparse matrices. An empty sparse matrix can be initialized with a given size,

```
var a = Sparse(nrows,ncols)
```

Alternatively, a matrix can be created from an array of triplets,

```
var a = Sparse([[row, col, value] ...])
```

For example,

```
var a = Sparse([[0,0,2], [1,1,-2]])
```

creates the matrix

```
[ 2 0 ]
[ 0 -2 ]
```

Once a sparse matrix is created, you can use all the regular arithmetic operators with matrix operands, e.g.

```
a+b
a*b
```

Chapter 7

Computational Geometry

7.1 Field

Fields are used to store information, including numbers or matrices, associated with the elements of a `Mesh` object.

You can create a `Field` by applying a function to each of the vertices,

```
var f = Field(mesh, fn (x, y, z) x+y+z)
```

or by supplying a single constant value,

```
var f = Field(mesh, Matrix([1,0,0]))
```

Fields can then be added and subtracted using the + and - operators.

To access elements of a `Field`, use index notation:

```
print f[grade, element, index]
```

where * `grade` is the grade to select * `element` is the element id * `index` is the element index

As a shorthand, it's possible to omit the grade and index; these are then both assumed to be `0`:

```
print f[2]
```

7.2 Grade

To create fields that include grades other than just vertices, use the `grade` option to `Field`. This can be just a grade index,

```
var f = Field(mesh, 0, grade=2)
```

which creates an empty field with `0` for each of the facets of the mesh `mesh`.

You can store more than one item per element by supplying a list to the `grade` option indicating how many items you want to store on each grade. For example,

```
var f = Field(mesh, 1.0, grade=[0,2,1])
```

stores two numbers on the line (grade 1) elements and one number on the facets (grade 2 elements). Each number in the field is initialized to the value `1.0`.

7.3 Shape

The `shape` method returns a list indicating the number of items stored on each element of a particular grade. This has the same format as the list you supply to the `grade` option of the `Field` constructor. For example,

```
[1,0,2]
```

would indicate one item stored on each vertex and two items stored on each facet.

7.4 Op

The `op` method applies a function to every item stored in a `Field`. For example,

```
f.op(fn (x) x.norm())
```

calls the `norm` method on each element stored in `f`.

Additional `Field` objects may supplied as extra arguments to `op`. These must have the same shape (the same number of items stored on each grade). The function supplied to `op` will now be called with the corresponding element from each field as arguments. For example,

```
f.op(fn (x,y) x.inner(y), g)
```

calculates an elementwise inner product between the elements of Fields `f` and `g`, returning the result as elements of a new `Field` object.

7.5 Functionals

A number of `functionals` are available in Morpho. Each of these represents an integral over some `Mesh` and `Field` objects (on a particular `Selection`) and are used to define energies and constraints in an `OptimizationProblem` provided by the `optimize` module.

Many functionals are built in. Additional functionals are available by importing the `functionals` module:

```
import ‘functionals‘
```

Functionals provide a number of standard methods:

- `total`(mesh) - returns the value of the integral with a provided mesh, selection and fields
- `integrand`(mesh) - returns the contribution to the integral from each element
- `gradient`(mesh) - returns the gradient of the functional with respect to vertex motions.
- `fieldgradient`(mesh, field) - returns the gradient of the functional with respect to components of the field

Each of these may be called with a mesh, a field and a selection.

7.6 Length

A `Length` functional calculates the length of a line elements in a mesh.

Evaluate the length of a circular loop: import constants import meshtools var m = LineMesh(fn (t) [cos(t), sin(t), 0], 0. . . 2*Pi:Pi/20, closed=true) var le = Length() print le.total(m)

7.7 AreaEnclosed

An `AreaEnclosed` functional calculates the area enclosed by a loop of line elements.

```
var la = AreaEnclosed()
```

7.8 Area

An `Area` functional calculates the area of a area elements in a mesh:

```
var la = Area()
print la.total(mesh)
```

7.9 VolumeEnclosed

An `VolumeEnclosed` functional calculates the volume enclosed by a surface of line elements. Note that this estimate may be inaccurate for highly deformed surfaces.

```
var lv = VolumeEnclosed()
```

7.10 Volume

An `Volume` functional calculates the volume of volume elements.

```
var lv = Volume()
```

7.11 ScalarPotential

The `ScalarPotential` functional is applied to point elements.

```
var ls = ScalarPotential(potential, gradient)
```

You must supply two functions (which may be anonymous) that return the potential and gradient respectively.

This functional is often used to implement level set constraints. For example, to confine a set of points to a sphere:

```
import optimize
fn sphere(x,y,z) { return x^2+y^2+z^2-1 }
fn grad(x,y,z) { return Matrix([2*x, 2*y, 2*z]) }
var lsph = ScalarPotential(sphere, grad)
problem.addlocalconstraint(lsph)
```

See the thomson example.

7.12 LinearElasticity

The `LinearElasticity` functional measures the linear elastic energy away from a reference state.

You must initialize with a reference mesh: var le = LinearElasticity(mref)

Manually set the poisson's ratio and grade to operate on: le.poissonratio = 0.2 le.grade = 2

7.13 EquiElement

The `EquiElement` functional measures the discrepency between the size of elements adjacent to each vertex. It can be used to equalize elements for regularization purposes.

7.14 LineCurvatureSq

The `LineCurvatureSq` functional measures the integrated curvature squared of a sequence of line elements.

7.15 LineTorsionSq

The `LineTorsionSq` functional measures the integrated torsion squared of a sequence of line elements.

7.16 MeanCurvatureSq

The `MeanCurvatureSq` functional computes the integrated mean curvature over a surface.

7.17 GaussCurvature

The `GaussCurvature` computes the integrated gaussian curvature over a surface.

7.18 GradSq

The `GradSq` functional measures the integral of the gradient squared of a field. The field can be a scalar, vector or matrix function.

Initialize with the required field: var le=GradSq(phi)

7.19 Nematic

The `Nematic` functional measures the elastic energy of a nematic liquid crystal.

```
var lf=Nematic(nn)
```

There are a number of optional parameters that can be used to set the splay, twist and bend constants:

```
var lf=Nematic(nn, ksplay=1, ktwist=0.5, kbend=1.5, pitch=0.1)
```

These are stored as properties of the object and can be retrieved as follows: print lf.ksplay

7.20 NematicElectric

The `NematicElectric` functional measures the integral of a nematic and electric coupling term integral((n.E)^2) where the electric field E may be computed from a scalar potential or supplied as a vector.

Initialize with a director field `nn` and a scalar potential `phi`: var lne = NematicElectric(nn, phi)

7.21 NormSq

The `NormSq` functional measures the elementwise L2 norm squared of a field.

7.22 LineIntegral

The `LineIntegral` functional computes the line integral of a function. You supply an integrand function that takes a position matrix as an argument.

To compute integral(x^{2+y}2) over a line element:

```
var la=LineIntegral(fn (x) x[0]^2+x[1]^2)
```

The function `tangent()` returns a unit vector tangent to the current element:

```
var la=LineIntegral(fn (x) x.inner(tangent()))
```

You can also integrate functions that involve fields:

```
var la=LineIntegral(fn (x, n) n.inner(tangent()), n)
```

where `n` is a vector field. The local interpolated value of this field is passed to your integrand function. More than one field can be used; they are passed as arguments to the integrand function in the order you supply them to `LineIntegrand`.

7.23 AreaIntegral

The `AreaIntegral` functional computes the area integral of a function. You supply an integrand function that takes a position matrix as an argument.

To compute integral(x*y) over an area element:

```
var la=AreaIntegral(fn (x) x[0]*x[1])
```

You can also integrate functions that involve fields:

```
var la=AreaIntegral(fn (x, phi) phi^2, phi)
```

More than one field can be used; they are passed as arguments to the integrand function in the order you supply them to `AreaIntegrand`.

7.24 Mesh

The `Mesh` class provides support for meshes. Meshes may consist of different kinds of element, including vertices, line elements, facets or area elements, tetrahedra or volume elements.

To create a mesh, you can import it from a file:

```
var m = Mesh("sphere.mesh")
```

or use one of the functions available in `meshtools` or `implicitmesh` packages.

Each type of element is referred to as belonging to a different `grade`. Point-like elements (vertices) are *grade 0*; line-like elements (edges) are *grade 1*; area-like elements (facets; triangles) are *grade 2* etc.

The `plot` package includes functions to visualize meshes.

7.25 Save

Saves a mesh as a .mesh file.

```
m.save("new.mesh")
```

7.26 Vertexposition

Retrieves the position of a vertex given an id:

```
print m.vertexposition(id)
```

7.27 Setvertexposition

Sets the position of a vertex given an id and a position vector:

```
print m.setvertexposition(1, Matrix([0,0,0]))
```

7.28 Addgrade

Adds a new grade to a mesh. This is commonly used when, for example, a mesh file includes facets but not edges. To add the missing edges:

```
m.addgrade(1)
```

7.29 Addsymmetry

Adds a symmetry to a mesh. Experimental in version 0.5.

7.30 Maxgrade

Returns the highest grade element present:

```
print m.maxgrade()
```

7.31 Count

Counts the number of elements. If no argument is provided, returns the number of vertices. Otherwise, returns the number of elements present of a given grade:

```
print m.count(2) // Returns the number of area-like elements.
```

7.32 Selection

The Selection class enables you to select components of a mesh for later use. You can supply a function that is applied to the coordinates of every vertex in the mesh, or select components like boundaries.

Create an empty selection: var s = Selection(mesh)

Select vertices above the z=0 plane using an anonymous function: var s = Selection(mesh, fn (x,y,z) z>0)

Select the boundary of a mesh: var s = Selection(mesh, boundary=true)

Selection objects can be composed using set operations: var s = s1.union(s2)

or var s = s1.intersection(s2)

To add additional grades, use the addgrade method. For example, to add areas: s.addgrade(2)

7.33 addgrade

Adds elements of the specified grade to a Selection. For example, to add edges to an existing selection, use

```
s.addgrade(1)
```

By default, this only adds an element if *all* vertices in the element are currently selected. Sometimes, it's useful to be able to add elements for which only some vertices are selected. The optional argument `partials` allows you to do this:

```
s.addgrade(1, partials=true)
```

Note that this method modifies the existing selection, and does not generate a new Selection object.

7.34 removegrade

Removes elements of the specified grade from a Selection. For example, to remove edges from an existing selection, use

```
s.removegrade(1)
```

Note that this method modifies the existing selection, and does not generate a new Selection object.

7.35 idlistforgrade

Returns a list of element ids included in the selection.

To find out which edges are selected: var edges = s.idlistforgrade(1)

7.36 isselected

Checks if an element id is selected, returning `true` or `false` accordingly.

To check if edge number 5 is selected: var f = s.isselected(1, 5))

Chapter 8

I/O

8.1 File

The `File` class provides the capability to read from and write to files, or to obtain the contents of a file in convenient formats.

To open a file, create a File object with the filename as the argument

```
var f = File("myfile.txt")
```

which opens `"myfile.txt"` for *reading*. To open a file for writing or appending, you need to provide a mode selector

```
var g = File("myfile.txt", "write")
```

or

```
var g = File("myfile.txt", "append")
```

Once the file is open, you can then read or write by calling appropriate methods:

```
f.lines()            // reads the contents of the file into an array of lines.
f.readline()         // reads a single line
f.readchar()         // reads a single character.
f.write(string)      // writes the arguments to the file.
```

After you're done with the file, close it with

```
f.close()
```

8.2 lines

Returns the contents of a file as an array of strings; each element corresponds to a single line.

Read in the contents of a file and print line by line:

```
var f = File("input.txt")
var s = f.lines()
for (i in s) print i
f.close()
```

8.3 readline

Reads a single line from a file; returns the result as a string.

Read in the contents of a file and print each line:

```
var f = File("input.txt")
while (!f.eof()) {
  print f.readline()
}
f.close()
```

8.4 readchar

Reads a single character from a file; returns the result as a string.

8.5 write

Writes to a file.

Write the contents of a list to a file:

```
var f = File("output.txt", "w")
for (k, i in list) f.write("${i}: ${k}")
f.close()
```

8.6 close

Closes an open file.

Chapter 9

Modules

9.1 Color

The `color` module provides support for working with color. Colors are represented in morpho by `Color` objects. The module predefines some colors including `Red`, `Green`, `Blue`, `Black`, `White`.

To use the module, use import as usual:

```
import color
```

Create a Color object from an RGB pair:

```
var col = Color(0.5,0.5,0.5) // A 50% gray
```

The module also provides `ColorMap`s which are subclasses of color that map a single parameter in the range 0..1 onto a continuum of colors. These include `GradientMap`, `GrayMap` and `HueMap`. `Color`s and `Colormap`s have the same interface.

Get the red, green or blue components of a color or colormap:

```
var col = HueMap()
print col.red(0.5) // argument can be in range 0..1
```

Get all three components as a list:

```
col.rgb(0)
```

Create a grayscale:

```
var c = Gray(0.2) // 20% gray
```

9.2 rgb

Gets the rgb components of a `Color` or `ColorMap` object as a list. Takes a single argument in the range 0..1, although the result will only depend on this argument if the object is a `ColorMap`.

```
var col = Color(0.1,0.5,0.7)
print col.rgb(0)
```

9.3 Constants

The constants module contains a number of useful mathematical and physical constants. Import it like any other module:

```
import constants
```

Available constants:

- `E` the base of natural logarithms.
- `Pi` ratio of the perimeter of a circle to its diameter.

9.4 Graphics

The `graphics` module provides a number of classes to provide simple visualization capabilities. To use it, you first need to import the module:

```
import graphics
```

The `Graphics` class acts as an abstract container for graphical information; to actually launch the display see the `Show` class. You can create an empty scene like this,

```
var g = Graphics()
```

Additional elements can be added using the `display` method.

```
g.display(element)
```

Morpho provides the following fundamental Graphical element classes:

```
TriangleComplex
```

You can also use functions like `Arrow`, `Tube` and `Cylinder` to create these elements conveniently.

To combine graphics objects, use the add operator:

```
var g1 = Graphics(), g2 = Graphics()
// ...
Show(g1+g2)
```

9.5 Show

`Show` is used to launch an interactive graphical display using the external `morphoview` application. `Show` takes a `Graphics` object as an argument:

```
var g = Graphics()
Show(g)
```

9.6 TriangleComplex

A `TriangleComplex` is a graphical element that can be used as part of a graphical display. It consists of a list of vertices and a connectivity matrix that selects which vertices are used in each triangle.

To create one, call the constructor with the following arguments:

```
TriangleComplex(position, normals, colors, connectivity)
```

- `position` is a `Matrix` containing vertex positions as *columns*.
- `normals` is a `Matrix` with a normal for each vertex.
- `colors` is the color of the object.
- `connectivity` is a `Sparse` matrix where each column represents a triangle and rows correspond to vertices.

Add to a `Graphics` object using the `display` method.

9.7 Arrow

The `Arrow` function creates an arrow. It takes two arguments:

```
arrow(start, end)
```

- `start` and `end` are the two vertices. The arrow points `start` -> `end`.

You can also provide optional arguments:

- `aspectratio` controls the width of the arrow relative to its length
- `n` is an integer that controls the quality of the display. Higher `n` leads to a rounder arrow.
- `color` is the color of the arrow. This can be a list of RGB values or a `Color` object

Display an arrow:

```
var g = Graphics([])
g.display(Arrow([-1/2,-1/2,-1/2], [1/2,1/2,1/2], aspectratio=0.05, n=10))
Show(g)
```

9.8 Cylinder

The `Cylinder` function creates a cylinder. It takes two required arguments:

```
cylinder(start, end)
```

- `start` and `end` are the two vertices.

You can also provide optional arguments:

- `aspectratio` controls the width of the cylinder relative to its length.
- `n` is an integer that controls the quality of the display. Higher `n` leads to a rounder cylinder.
- `color` is the color of the cylinder. This can be a list of RGB values or a `Color` object.

Display an cylinder:

```
var g = Graphics()
g.display(Cylinder([-1/2,-1/2,-1/2], [1/2,1/2,1/2], aspectratio=0.1, n=10))
Show(g)
```

9.9 Tube

The `Tube` function connects a sequence of points to form a tube.

```
Tube(points, radius)
```

- `points` is a list of points; this can be a list of lists or a `Matrix` with the positions as columns.
- `radius` is the radius of the tube.

You can also provide optional arguments:

- `n` is an integer that controls the quality of the display. Higher `n` leads to a rounder tube.
- `color` is the color of the tube. This can be a list of RGB values or a `Color` object.
- `closed` is a `bool` that indicates whether the tube should be closed to form a loop.

Draw a square:

```
var a = Tube([[-1/2,-1/2,0],[1/2,-1/2,0],[1/2,1/2,0],[-1/2,1/2,0]], 0.1, closed=true)
var g = Graphics()
g.display(a)
```

9.10 Sphere

The `Sphere` function creates a sphere.

```
Sphere(center, radius)
```

- `center` is the position of the center of the sphere; this can be a list or column `Matrix`.
- `radius` is the radius of the sphere

You can also provide an optional argument:

- `color` is the color of the sphere. This can be a list of RGB values or a `Color` object.

Draw some randomly sized spheres:

```
var g = Graphics()
for (i in 0...10) {
  g.display(Sphere([random()-1/2, random()-1/2, random()-1/2], 0.1*(1+random()),
color=Gray(random())))
}
Show(g)
```

9.11 ImplicitMesh

The `implicitmesh` module allows you to build meshes from implicit functions. For example, the unit sphere could be specified using the function x^{2+y}2+zˆ2-1 == 0.

To use the module, first import it:

```
import implicitmesh
```

To create a sphere, first create an ImplicitMeshBuilder object with the implict function you'd like to use:

```
var impl = ImplicitMeshBuilder(fn (x,y,z) x^2+y^2+z^2-1)
```

You can use an existing function (or method) as well as an anonymous function as above.

Then build the mesh,

```
var mesh = impl.build(stepsize=0.25)
```

The `build` method takes a number of optional arguments:

- `start` - the starting point. If not provided, the value Matrix([1,1,1]) is used.
- `stepsize` - approximate lengthscale to use.
- `maxiterations` - maximum number of iterations to use. If this limit is exceeded, a partially built mesh will be returned.

9.12 KDTree

The `kdtree` module implements a k-dimensional tree, a space partitioning data structure that can be used to accelerate computational geometry calculations.

To use the module, first import it:

```
import kdtree
```

To create a tree from a list of points:

```
var pts = []
for (i in 0...100) pts.append(Matrix([random(), random(), random()]))
var tree=KDTree(pts)
```

Add further points:

```
tree.insert(Matrix([0,0,0]))
```

Test whether a given point is present in the tree:

```
tree.ismember(Matrix([1,0,0]))
```

Find all points within a given bounding box:

```
var pts = tree.search([[-1,1], [-1,1], [-1,1]])
for (x in pts) print x.location
```

Find the nearest point to a given point:

```
var pt = tree.nearest(Matrix([0.1, 0.1, 0.5]))
print pt.location
```

9.13 Insert

Inserts a new point into a k-d tree. Returns a KDTreeNode object.

```
var node = tree.insert(Matrix([0,0,0]))
```

Note that, for performance reasons, if the set of points is known ahead of time, it is generally better to build the tree using the constructor function KDTree rather than one-by-one with insert.

9.14 Ismember

Checks if a point is a member of a k-d tree. Returns `true` or `false`.

```
print tree.ismember(Matrix([0,0,0]))
```

9.15 Nearest

Finds the point in a k-d tree nearest to a point of interest. Returns a KDTreeNode object.

```
var pt = tree.nearest(Matrix([0.1, 0.1, 0.5]))
```

To get the location of this nearest point, access the location property:

```
print pt.location
```

9.16 Search

Finds all points in a k-d tree that lie within a cuboidal bounding box. Returns a list of KDTreeNode objects.

Find and display all points that lie in a cuboid 0<=x<=1, 0<=y<=2, 1<=z<=2:

```
var result = tree.search([[0,1], [0,2], [1,2]])
for (x in result) print x.location
```

9.17 KDTreeNode

An object corresponding to a single node in a k-d tree. To get the location of the node, access the `location` property:

```
print node.location
```

9.18 Meshtools

The Meshtools package contains a number of functions and classes to assist with creating and manipulating meshes.

9.19 AreaMesh

This function creates a mesh composed of triangles from a parametric function. To use it:

```
var m = LineMesh(function, range1, range2, closed=boolean)
```

where

- `function` is a parametric function that has one parameter. It should return a list of coordinates or a column matrix corresponding to this parameter.
- `range1` is the Range to use for the first parameter of the parametric function.
- `range2` is the Range to use for the second parameter of the parametric function.
- `closed` is an optional parameter indicating whether to create a closed loop or not. You can supply a list where each element indicates whether the relevant parameter is closed or not.

To use `AreaMesh`, import the `meshtools` module:

```
import meshtools
```

Create a square:

```
var m = LineMesh(fn (u,v) [u, v, 0], 0..1:0.1, 0..1:0.1)
```

Create a tube:

```
var m = AreaMesh(fn (u, v) [v, cos(u), sin(u)], -Pi...Pi:Pi/4, -1..1:0.1, closed=[true, fa
```

Create a torus: var c=0.5, a=0.2 var m = AreaMesh(fn (u, v) [(c + a*cos(v))*cos(u), (c + a*cos(v))*sin(u), a*sin(v)], 0. . . 2*Pi:Pi/16, 0. . . 2*Pi:Pi/8, closed=true)

9.20 LineMesh

This function creates a mesh composed of line elements from a parametric function. To use it:

```
var m = LineMesh(function, range, closed=boolean)
```

where

- `function` is a parametric function that has one parameter. It should return a list of coordinates or a column matrix corresponding to this parameter.
- `range` is the Range to use for the parametric function.
- `closed` is an optional parameter indicating whether to create a closed loop or not.

To use `LineMesh`, import the `meshtools` module:

```
import meshtools
```

Create a circle:

```
import constants
var m = LineMesh(fn (t) [sin(t), cos(t), 0], 0...2*Pi:2*Pi/50, closed=true)
```

9.21 PolyhedronMesh

This function creates a mesh from a polyhedron specification.

9.22 Equiangulate

Attempts to equiangulate a mesh.

9.23 MeshBuilder

The `MeshBuilder` class simplifies user creation of meshes. To use this class, begin by creating a `MeshBuilder` object:

```
var build = MeshBuilder()
```

You can then add vertices, edges, etc. one by one using `addvertex`, `addedge` and `addtriangle`. Each of these returns an element id:

```
var id1=build.addvertex(Matrix([0,0,0]))
var id2=build.addvertex(Matrix([1,1,1]))
build.addedge([id1, id2])
```

Once the mesh is ready, call the `build` method to construct the `Mesh`:

```
var m = build.build()
```

9.24 MeshRefiner

The `MeshRefiner` class is used to refine meshes, and to correct associated data structures that depend on the mesh.

9.25 Optimize

The `optimize` package contains a number of functions and classes to perform shape optimization.

9.26 OptimizationProblem

An `OptimizationProblem` object defines an optimization problem, which may include functionals to optimize as well as global and local constraints.

Create an `OptimizationProblem` with a mesh:

```
var problem = OptimizationProblem(mesh)
```

Add an energy:

```
var la = Area()
problem.addenergy(la)
```

Add an energy that operates on a selected region, and with an optional prefactor:

```
problem.addenergy(la, selection=sel, prefactor=2)
```

Add a constraint:

```
problem.addconstraint(la)
```

Add a local constraint (here a onesided level set constraint):

```
var ls = ScalarPotential(fn (x,y,z) z, fn (x,y,z) Matrix([0,0,1]))
problem.addlocalconstraint(ls, onesided=true)
```

9.27 Optimizer

`Optimizer` objects are used to optimize `Mesh`es and `Field`s. You should use the appropriate subclass: `ShapeOptimizer` or `FieldOptimizer` respectively.

9.28 ShapeOptimizer

A `ShapeOptimizer` object performs shape optimization: it moves the vertex positions to reduce an overall energy.

Create a `ShapeOptimizer` object with an `OptimizationProblem` and a `Mesh`:

```
var sopt = ShapeOptimizer(problem, m)
```

Take a step down the gradient with fixed stepsize:

```
sopt.relax(5) // Takes five steps
```

Linesearch down the gradient:

```
sopt.linesearch(5) // Performs five linesearches
```

Control a number of properties of the optimizer:

```
sopt.stepsize=0.1 // The stepsize to take
sopt.steplimit=0.5 // Maximum stepsize for optimizing methods
sopt.etol = 1e-8 // Energy convergence tolerance
sopt.ctol = 1e-9 // Tolerance to which constraints are satisfied
sopt.maxconstraintsteps = 20 // Maximum number of constraint steps to use
```

9.29 FieldOptimizer

A `FieldOptimizer` object performs field optimization: it changes elements of a `Field` to reduce an overall energy.

Create a `FieldOptimizer` object with an `OptimizationProblem` and a `Field`:

```
var sopt = FieldOptimizer(problem, fld)
```

Field optimizers provide the same options and methods as Shape optimizers: see the `ShapeOptimizer` documentation for details.

9.30 POVRay

The `povray` module provides integration with POVRay, a popular open source ray-tracing package for high quality graphical rendering. To use the module, first import it:

```
import povray
```

To raytrace a graphic, begin by creating a `POVRaytracer` object:

```
var pov = POVRaytracer(graphic)
```

Create a .pov file that can be run with POVRay:

```
pov.write("out.pov")
```

Create, render and display a scene using POVRay:

```
pov.render("out.pov")
```

The `POVRaytracer` constructor supports a number of optional arguments:

- `antialias` - whether to antialias the output or not
- `width` - image width
- `height` - image height
- `viewangle` - camera angle (higher means wider view)
- `viewpoint` - position of camera