# Contents

1 Introduction 3

2 Installation 5
   2.1 Requirements .................................................. 5
   2.2 From Git Hub ..................................................... 5
   2.3 From a Package .................................................. 6

3 Meshing 7
   3.1 Creating a Mesh ................................................. 7
   3.2 Plotting .......................................................... 8
   3.3 Saving and Loading ............................................. 9
   3.4 Nodes ............................................................. 10
   3.5 Elements ......................................................... 11

4 Tutorial: Creating 2D Meshes and Fitting to Data 15
   4.1 Building a Mesh ................................................. 15
   4.2 Generate Data ................................................... 16
   4.3 Plotting .......................................................... 16
   4.4 Fitting ............................................................ 18

5 Tutorial: PCA Meshes 23
   5.1 Creating a Population of 100 Meshes ......................... 23
   5.2 Performing PCA ................................................ 24
   5.3 Creating a PCA Mesh .......................................... 25

6 Indices and tables 29

Python Module Index 31
Morphic is a convenient module for mesh creation, analysis, rendering and fitting to data.

Currently it can handle 1D and 2D meshes with:

- Lagrange elements (up to 4th-order)
- Triangle elements (up to 4th order)
- Hermite elements (cubic-Hermite only)

Morphic also offers four types of nodes allowing creation of complex meshes:

- Standard nodes which store field values, e.g., x, y, z and temperature, with components such as derivatives.
- Dependent node which can be calculated from a location on another element, for example, hanging nodes.
- PCA nodes to handle PCA models (todo)
- Mapping nodes which can be based on values from other nodes. This allows versions of nodes or the creation of a node for a cubic-Hermite element where the derivatives might be computed from a Lagrange element.

Note: 3D and higher-dimensional meshes are possible but it has not been fully implemented.

The following tutorials will go through the process of building meshes and fitting meshes.
CHAPTER 2

Installation

The only way to install Morphic at the moment is to download the source from Git Hub. In the future, we plan to create an easy_install Python package.

Requirements

• numpy
• scipy

Optional

• sparsesvd for fitting
• matplotlib 2D plotting
• mayavi2 for 2D and 3D plotting

From Git Hub

Clone the Morphic code:

```
    git clone http://github.com/duanemalcolm/morphic
```

Add the morphic path to the Python path by adding it to your shell script:

```
    nano ~/.bashrc
    export PYTHONPATH=$PYTHONPATH:~/path/to/morphic
```

Run an example:
From a Package

<table>
<thead>
<tr>
<th>Note:</th>
<th>We plan to create easy install packages in the future</th>
</tr>
</thead>
</table>

```bash
cd ~/path/to/morphic/examples
python example_1d_linear.py plot
```
A mesh is a convenient way to describe field values over a domain. Examples of fields include temperature fields, velocity fields, strain fields, and geometric fields such as the line path a string might take or the surface or volume of an object.

A mesh typically consists of field values, a topography over which they are sampled and the basis functions used to evaluate the fields over the topography. In morphic nodes define field values and elements define the topography and basis functions.

**In this section, we will introduce:**

- creating mesh
- plotting a mesh
- saving and loading a mesh
- nodes
- elements

We will do this through examples, more details on the mesh, nodes and elements can be found in the API (LINK).

**Warning:** Morphic has been developed for 1D and 2D meshes. Support for higher order meshes are intended however it is not complete.

### Creating a Mesh

Before we can create a mesh we need to import the Morphic module:

```python
import morphic
```

Now we can create a mesh by:
mesh = morphic.Mesh()

The mesh is created from two biquadratic elements which joined along one edge. For this we add 15 nodes in a regular 5x3 grid where the z-value of the middle node of elements 1 and 2 are set to 1 and -1.

The command to add a standard node is:

```python
mesh.add_stdnode(id, field_values)
```

Here we add the 15 nodes each with an id and and x, y, z coordinate,

```python
mesh.add_stdnode(1, [0, 0, 0])
mesh.add_stdnode(2, [1, 0, 0])
mesh.add_stdnode(3, [2, 0, 0])
mesh.add_stdnode(4, [3, 0, 0])
mesh.add_stdnode(5, [4, 0, 0])
mesh.add_stdnode(6, [0, 1, 0])
mesh.add_stdnode(7, [1, 1, 1])
mesh.add_stdnode(8, [2, 1, 0])
mesh.add_stdnode(9, [3, 1, 0])
mesh.add_stdnode(10, [4, 1, 0])
mesh.add_stdnode(11, [0, 2, 0])
mesh.add_stdnode(12, [1, 2, 0])
mesh.add_stdnode(13, [2, 2, 0])
mesh.add_stdnode(14, [3, 2, 0])
mesh.add_stdnode(15, [4, 2, 0])
```

Now we add the two elements. The command for adding elements is:

```python
mesh.add_element(id, basis_functions, node_ids)
```

which requires an element id, the basis function used to evaluate fields over the element, and the node ids for creating the elements. Therefore, we add elements to the mesh by,

```python
mesh.add_element(1, ['L2', 'L2'], [1, 2, 3, 6, 7, 8, 11, 12, 13])
mesh.add_element(2, ['L2', 'L2'], [3, 4, 5, 8, 9, 10, 13, 14, 15])
```

Finally, we need to generate the mesh,

```python
mesh.generate()
```

This creates an efficient store of the mesh in order to compute properties of the mesh quickly.

The resultant plot of the mesh is shown below.

**Plotting**

Morphic has a `viewer` module for plotting meshes, which we import by,

```python
import morphic.viewer
```

Then we create a scene to plot into,
S = morphic.viewer.Scenes('my_scene', bgcolor=(1,1,1))

The first variable is the label to assign to the scene and bgcolor are the RBG values for the background colour of the scene.

We would like to plot the nodes and surface of the mesh which can be done by,

\[
X_n = \text{mesh.get_nodes()}
\]
\[
X_s, T_s = \text{mesh.get_surfaces(res=32)}
\]

The variable res=32 defines the discretization of each element. The return variable from mesh.get_nodes is an array x, y, z coordinates (Xn) of nodes in the mesh. The variables returned from mesh.get_surfaces are the x, y, z coordinates (Xs) and connectivity (Ts) of the triangulated surface of the elements.

Now we can render the nodes and surface,

\[
S\text{.plot_points('nodes', Xn, color=(1,0,1), size=0.1)}
\]
\[
S\text{.plot_surfaces('surface', Xs, Ts, scalars=Xs[:,2])}
\]

The first variable in each command is the label given to the rendering of the nodes and surfaces, color is the RGB colour to render the nodes, size is the size of the nodes, and scalars is the colour field rendered on the mesh surface, which in this case, is the z value of the coordinates field.

**Saving and Loading**

Saving a mesh is simply,
mesh.save('path/to/meshes/funky.mesh')

A mesh can be loaded two ways,

```python
mesh = morphic.Mesh('path/to/meshes/funky.mesh')
# OR
mesh = morphic.Mesh()
mesh.load('path/to/meshes/funky.mesh')
```

## Nodes

There are three types of nodes that can be added to a mesh:

- **Standard Nodes** Stores field values. The fields can include components, for example, in the case where field derivatives or PCA modes are included.

- **Dependent Nodes** Describes a node that depends on other parts of a mesh, typically, a node embedded in an element.

- **PCA Nodes** Describes a nodes whose values depend on the weighted sum of PCA modes.

A standard node can be added to the mesh by,

```python
node = mesh.add_stdnode(id, values)
```

where `id` is the unique identified for nodes, and `values` are the field values for the node. This command will return a node object.

The `id` variable can be defined by user as integer, string or `None`. If set to `None` a unique integer id will be assigned.

The `value` variable can be a one or two dimensional list or numpy array of field values. In the case of a one-dimensional array, e.g., `values = [0.2, 1.5, -0.4]`, each value is assumed to be a field value. In the case of a two-dimensional array, e.g., `values = [[0.2, 1, 0, 0], [1.5, 0, 1, 0], [-0.4, 0, 0, 0]]`, the rows represent the fields and the columns represents the field components. Examples of field components are field derivative or mode vectors for a PCA model.

### Accessing Nodes

Nodes are stored in a mesh as a list of node objects which can be accessed through a list or by direct reference by node id.

```python
list_of_nodes = mesh.nodes
node = mesh.nodes[5]  # if the id is an integer
node = mesh.nodes['my_node']  # if the id is a string
```

### Node Values

You can get or set values for a standard node by,
# Get or set values by
all_values = node.values
definitions = 
node.values = 
[ [1, 1, 0, 0.2], [2, 0, 1, 0.1] ]

# Can also get slices by,
field_values = node.values[:, 0]
field_deriv1 = node.values[:, 1]
y_field_values = node.values[1, :]

PCA Nodes

A PCA node describes a node whose values depend on the weighted sum of PCA modes. The definition to create a PCA node is,

definitions = 
node = mesh.add_pcanode(id, values_nid, weights_nid, variance_nid)

where the values_nid, weights_nid, and variance_nid are standard nodes that stores the pca mode values, the weights for the modes and the variance associated with the modes. The equation for the final values for the pca node is given by:

\[ X = X_{\text{pca}} \cdot (W \cdot V) \]

Note: After the values of the weights (or variance) are changed, mesh.update_pca_nodes() must be called to update/re-compute the values of PCA nodes.

Elements

An element can be added to a mesh by,

definitions = 

where id is the unique identified for elements, interp is the interpolation functions in each dimension, and nodes are the node ids for the element. This command will return a element object.

The id variable can be defined by user as integer, string or None. If set to None a unique integer id will be assigned.

The interp variable is a list of strings each representing the interpolation scheme in each dimension, for example, ‘interp = [‘L1’, ‘H3’] for a linear-cubic-Hermite two-dimensional element.

Interpolation schemes include:

- L1 - linear lagrange
- L2 - quadratic lagrange
- L3 - cubic lagrange
- L4 - quartic lagrange
- H3 - cubic-Hermite
- T11 - linear 2d-simplex
- T22 - quadratic 2d-simplex
- T33 - cubic 2d-simplex
Some examples of interpolation schemes:

- ['L1', 'L1'] = bilinear (2d)
- ['L3', 'L2'] = cubic-quadratic (2d)
- ['H3', 'L1', 'L1'] = cubic-Hermite-bilinear (3d) - note warning below.
- ['T22'] = biquadratic simplex (2d triangle)
- ['T11', 'L1'] = a linear prism (3d) - note warning below.

Warning: Morphic only supports one and two dimensional elements. Morphic can support some higher order elements but this is not fully implemented or thoroughly tested.

Accessing Elements

Elements are stored in a mesh as a list of element objects which can be accessed through a list or by direct reference by element id.

```python
list_of_elements = mesh.elements
element = mesh.elements[1]  # if the id is an integer
element = mesh.elements['my_element']  # if the id is a string
```

Element Properties

Element objects have a number of useful properties:

- `element.id` returns the element id.
- `element.basis` returns a list of the element basis functions.
- `element.shape` returns a string identifying the shape of the element, for example, `line`, `tri`, or `quad` for a lines, triangle or quadrilateral element.
- `element.nodes` returns a list of node objects.
- `element.node_ids` returns a list of node ids.

Element Interpolation

One can evaluate the field values at any point on the element. The general expression is

```python
element.evaluate(Xi, deriv=None)
```

where `Xi` is a list or numpy array of element locations and `deriv` is the definition of the derivative interpolation. The default is `None` which interpolates the field values.

Examples for derivative definitions are:

- `deriv=[1,0]` returns the first derivative in direction 1
- `deriv=[1,1]` returns the cross-derivative ($\partial^2 u/\partial x_1 \partial x_2$)
- `deriv=[0,2]` returns the second derivative in direction 2
Element Surface Normal

One can also compute the normal at any point on a 2D element. The general expression is

\[ \text{element.normal}(\text{Xi}) \]

where \text{Xi} is a list or numpy array of element location.
In this tutorial we build a 2D mesh using cubic-Hermite elements and fit the mesh to data generated using a cosine function.

For this tutorial we’ll need to import `scipy` and `morphic`.

```python
import scipy
import morphic
```

**Building a Mesh**

First, we create some node point which will be arranged in a regular 3x3 grid ranging from \( x = [-\pi, \pi] \) and \( y = [-\pi, \pi] \). The \( z \) values are set to zero.

```python
# Generate a regular grid of X, Y and Z values
pi = scipy.pi
Xn = scipy.array([[-pi, -pi, 0],
                  [ 0, -pi, 0],
                  [ pi, -pi, 0],
                  [-pi,  0, 0],
                  [ 0,  0, 0],
                  [ pi,  0, 0],
                  [-pi, pi, 0],
                  [ 0, pi, 0],
                  [ pi, pi, 0]])
```

Because we are using cubic-Hermite element, the nodal values require derivative values which we default to,

```python
deriv = scipy.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
```

Now, we create the mesh,
mesh = morphic.Mesh()

We add the nodes where for each node value we append the derivative values,

```python
for i, xn in enumerate(Xn):
    xn_ch = scipy.append(scipy.array([xn]).T, deriv, axis=1)
    mesh.add_stdnode(i+1, xn_ch)
```

Now we add four bicubic-Hermite elements, which are ['H3', 'H3'] basis,

```python
mesh.add_element(1, ['H3', 'H3'], [1, 2, 4, 5])
mesh.add_element(2, ['H3', 'H3'], [2, 3, 5, 6])
mesh.add_element(3, ['H3', 'H3'], [4, 5, 7, 8])
mesh.add_element(4, ['H3', 'H3'], [5, 6, 8, 9])
```

And finally generate the mesh structures,

```python
mesh.generate()
```

**Generate Data**

The data cloud is based on a cosine function, \( z = \cos(x + 1) \cos(y + 1) \). A 200 x 200 grid of points are created in the x and y dimensions on which the z values are generated. The res=24 parameter sets the discretization of the elements. The higher the value the smoother the rendered surface.

```python
xd = scipy.linspace(-scipy.pi, scipy.pi, 200)
X, Y = scipy.meshgrid(xd, xd.T)
Xd = scipy.array([X.reshape((X.size)), Y.reshape((Y.size)), scipy.zeros((X.size, 1))]).T
Xd[:, 2] = (scipy.cos(Xd[:, 0])+1) * (scipy.cos(Xd[:, 1])+1)
```

**Plotting**

Here we want to view our initial mesh and data. First we need to extract the coordinates of the nodes (without derivatives) and a surface of the mesh, which is done using the following commands,

```python
Xn = mesh.get_nodes()
Xs, Ts = mesh.get_surfaces(res=24)
```

Next we want to plot the mesh nodes and surface and the data which is done using the morphic.viewer module,

```python
from morphic import viewer
S = viewer.Scenes('MyPlot', bgcolor=(1,1,1))
S.plot_points('nodes', Xn, color=(0,1,0), size=0.1)
S.plot_surfaces('surface', Xs, Ts, color=(0.2,0.5,1))
S.plot_points('data', Xd[::7,:], color=(1,0,0), mode='point')
```

The resultant plot is shown where the nodes are plotted in green, the mesh surface in blue and the data in red,
Fitting

In this section we describe how to fit the mesh to the datacloud. Here we use the fitting method that generates and solves a linear system of equations, i.e., \( A \cdot x = b \), where \( A \) is the matrix of weights for points on the mesh, \( x \) are the mesh parameters, and \( b \) are the data point that the mesh points are fitted to. See TODO: add link

There are two demonstrations of the fitting process, first we fit without constraining mesh node parameters and second we fit with constraints on the node values and derivatives.

**Fit Part 1: No Constraints**

Ok, let’s get started. First we create a new fitting instance:

```
fit = morphic.Fit()
```

Then we add element points on the mesh to the fitting process. This is a 10x10 grid of points on each element. These are the points that will be projected to the datacloud.

```
Xi_fit = mesh.elements[1].grid(res=10)
for elem in mesh.elements:
    for xi in Xi_fit:
        fit.bind_element_point(elem.id, [xi], 'datacloud')
```

Then we generate the fit from the mesh. Here the fitting module will calculate the weights for each grid point on the mesh and assemble these in the \( A \) matrix for the \( A \cdot x = b \) system.

```
fit.update_from_mesh(mesh)
```

We add the data cloud to the fitting process and call a function called `generate_fast_data` which creates a kd-tree of the data for fast searching of the closest data points.

```
fit.set_data('datacloud', Xd)
fit.generate_fast_data()
```

Now we invert the \( A \) matrix in order to speed up the fitting process and if the mesh to the datacloud. Currently we run the fit iteratively where each iteration involves finding the closest data points for the current mesh and then fitting the mesh to the closest data points. When the RMS error between the mesh and datacloud stops reducing or 1000 iterations are reached, then the fit will stop. These stopping criteria can be changed, see the details on the solve method.

```
fit.invert_matrix()
mesh, rms_err1 = fit.solve(mesh, output=True)
```

The plot of the resultant fit shows a fairly good fit except the boundaries are not straight and also we expect the middle node to be at the peak of the data cloud. This can be achieved in the fit by constraining the appropriate node parameters. This is described in the next section.

**Fit Part 2: With Constraints**

In this section we are going to constrain appropriate node values and derivatives to get straight boundaries and a more symmetric mesh. We define constrains on nodes values using the following command:
where the `weight` defines the weighting of the constraint. The default is 1, which is what the binding for the element points is set to. The higher this number the closer the resultant fit will meet its constraint.

**Warning:** Setting the `weight` very high compared to the minimum weight in the fit can cause fitting issues.

**Note:** The indexing of the `field_num` and `component_num` starts at zero.

Just like how we added the data cloud, the data values we would like to constrain the node parameters to are defined using a data label. First we define an array of data labels indicating which each node is constrained. The rows in the array represent the node and the columns represent x, y and z values. This allows a easy programatic way of setting the constraints. First, the node values are constrained, then the derivatives.

```python
fix_nodes = [
    ['-pi', '-pi', 'zero'],
    ['zero', '-pi', 'zero'],
    ['pi', '-pi', 'zero'],
    ['-pi', 'zero', 'zero'],
    ['zero', 'zero', 'four'],
    ['pi', 'zero', 'zero'],
    ['-pi', 'pi', 'zero'],
    ['zero', 'pi', 'zero'],
    ['pi', 'pi', 'zero']
]
weight1 = 100
for i, fix in enumerate(fix_nodes):
    node_id = i + 1
    # Fix node values
    fit.bind_node_value(node_id, 0, 0, fix[0], weight=weight1)
    fit.bind_node_value(node_id, 1, 0, fix[1], weight=weight1)
    fit.bind_node_value(node_id, 2, 0, fix[2], weight=weight1)
    # Fix node derivatives
    fit.bind_node_value(node_id, 0, 2, 'zero', weight=weight1)  # dx/dxi2=0
    fit.bind_node_value(node_id, 1, 1, 'zero', weight=weight1)  # dy/dxi1=0
    fit.bind_node_value(node_id, 2, 1, 'zero', weight=weight1)  # dz/dxi1=0
    fit.bind_node_value(node_id, 2, 2, 'zero', weight=weight1)  # dz/dxi2=0

# Flattens the corners. Stops z from dipping below zero, d2z/(dx1.dx2)=0
weight2 = 100
fit.bind_node_value(1, 2, 3, 'zero', weight=weight2)
fit.bind_node_value(3, 2, 3, 'zero', weight=weight2)
fit.bind_node_value(7, 2, 3, 'zero', weight=weight2)
fit.bind_node_value(9, 2, 3, 'zero', weight=weight2)
fit.update_from_mesh(mesh)
```

In the final line, we regenerate the fit from the mesh after we define the node value constraints.

Next, as above, we add the data values we constrain the node values to and regenerate the fast data. In this case, no kd-tree is created since there is only one value per data label.
fit.set_data('pi', scipy.pi)
fit.set_data('-pi', -scipy.pi)
fit.set_data('zero', 0)
fit.set_data('four', 4.)
fit.generate_fast_data()

And again we invert the matrix and solve the fit,

```
fit.invert_matrix()
mesh, rms_err2 = fit.solve(mesh, output=True)
```

As shown in the resultant plot, the fit produces straigher boundaries and a symmetric mesh, i.e., the nodes are laid out in a regular fashion.
In this tutorial we build a 1D cubic-Hermite PCA mesh. The population of meshes used to create the PCA mesh is
generated by modifying some nodal parameters using a Gaussian distribution. The output from PCA of this population
of meshes are used to create the PCA mesh.

For this tutorial we’ll need to import `scipy`, `random`, `pylab`, `mdp` and, of course, `morphic`.

## Creating a Population of 100 Meshes

First, we create simple 1D cubic-Hermite mesh with two nodes and one element.

```python
mesh = morphic.Mesh()
mesh.add_stdnode(1, scipy.zeros((2, 2)))
mesh.add_stdnode(2, scipy.zeros((2, 2)))
mesh.add_element(1, ['H3'], [1, 2])
mesh.generate()
```

Then we create a function to randomise six the mesh parameters:

```python
def randomise_mesh(mesh):
    import random
    import scipy

    xa = random.gauss(-2.5, 1)
    xb = random.gauss(2.5, 1)
    ya = random.gauss(-0.1, 0.5)
    yb = random.gauss(0.3, 0.5)
    dya = random.gauss(0, 0.1)
    dyb = random.gauss(0, 0.2)

    mesh.nodes[1].values = scipy.array([[xa, 5], [ya, dya + dyb]])
    mesh.nodes[2].values = scipy.array([[xb, 5], [yb, -dya]])
```

In this tutorial we build a 1D cubic-Hermite PCA mesh. The population of meshes used to create the PCA mesh is
generated by modifying some nodal parameters using a Gaussian distribution. The output from PCA of this population
of meshes are used to create the PCA mesh.

For this tutorial we’ll need to import `scipy`, `random`, `pylab`, `mdp` and, of course, `morphic`. 

## Creating a Population of 100 Meshes

First, we create simple 1D cubic-Hermite mesh with two nodes and one element.

```python
mesh = morphic.Mesh()
mesh.add_stdnode(1, scipy.zeros((2, 2)))
mesh.add_stdnode(2, scipy.zeros((2, 2)))
mesh.add_element(1, ['H3'], [1, 2])
mesh.generate()
```

Then we create a function to randomise six the mesh parameters:

```python
def randomise_mesh(mesh):
    import random
    import scipy

    xa = random.gauss(-2.5, 1)
    xb = random.gauss(2.5, 1)
    ya = random.gauss(-0.1, 0.5)
    yb = random.gauss(0.3, 0.5)
    dya = random.gauss(0, 0.1)
    dyb = random.gauss(0, 0.2)

    mesh.nodes[1].values = scipy.array([[xa, 5], [ya, dya + dyb]])
    mesh.nodes[2].values = scipy.array([[xb, 5], [yb, -dya]])
```
An example of running this function a 100 times and plotting the results is shown in the figure below.

Performing PCA

Although PCA can be performed on any data, it is not good practice to perform PCA on data with mixed units, which in our case is a cubic-Hermite mesh that has position and derivative values. To overcome this issue we will convert the cubic-Hermite mesh to a cubic-langrange mesh, which will preserve all information.

To perform PCA, we convert all our meshes to cubic-langrange by evaluating the field values at $X = 0, 1/3, 2/3, 1$. These values are stored in a large array for PCA.

```python
Npop = 100 # number in the population
Xp = scipy.zeros((8, Npop))

for i in range(Npop):
    mesh = randomise_mesh(mesh)
    Xp[:,i] = mesh.elements[1].evaluate(
        [[0], [1/3], [2/3], [1]]).flatten()
```

```
return mesh
```
Then we use MDP to perform PCA and print out the average mesh, mode values, and variance for each mode,

```python
pca = mdp.nodes.PCANode(svd=True)
pca.execute(Xp.T)
#~ print pca.avg # Average mesh
#~ print pca.v # The modes
#~ print pca.d # Variance shows there are 6 significant modes.
```

Now we can create a mesh that uses the PCA results.

## Creating a PCA Mesh

Morphic has a node class to manage PCA values as nodes. It knows how to evaluate the values of the node given a set of weights.

There are 4 steps to creating a PCA mesh:

1. add the PCA weights as a standard node
2. add the PCA variance as a standard node
3. add the PCA nodes which references weights and variance nodes
4. add the elements which references the PCA nodes

For example,

```python
pcamesh = morphic.Mesh()

pcamesh.add_stdnode('weights', [1, 0, 0, 0]) # Step 1: weights node
pcamesh.add_stdnode('variance', [1, 0.5, 0.2, 0.1]) # Step 2: variance node

# Step 3: add the pca node for each node
pcamesh.add_pcanode(1, xm1, 'weights', 'variance')
pcedesh.add_pcanode(2, xm2, 'weights', 'variance')
pcedesh.add_pcanode(3, xm3, 'weights', 'variance')

# Step 4: add elements...
pcedesh.add_element(1, ['L2'], [1, 2, 3])
pcedesh.generate()
```

:**warning:**

The first value of the weights and variance nodes should be set to 1 since it is multiplied with the mean values from the PCA.

For our case, we will initialise some values and create a new mesh called `pcamesh` which will store and evaluate the PCA values,

```python
pcamesh = morphic.Mesh()

Nnodes = 4 # number of nodes
Ndims = 2 # number of dimensions
Ncomps = 1 # number of components
Nmodes = 6 # number of modes
```
Now we add the weights values

```python
weights = scipy.zeros((7)) # Average mesh plus 6 modes
weights[0] = 1.0 # the weight on the average mesh
pcamesh.add_stdnode('weights', weights, group='pca_init')
```

And the variance values which are square rooted so that the weights can be defined in terms of standard deviations,

```python
variance = scipy.zeros((7)) # Average mesh plus 7 modes
variance[0] = 1.0 # Set to 1 for the average mesh
variance[1:] = scipy.sqrt(pca.d[:6]) # variance for the 7 modes
pcamesh.add_stdnode('variance', variance, group='pca_init')
```

The mode values are added as PCA nodes which reference the weights and variance nodes,

```python
xn = scipy.zeros((Ndims, Ncomps, Nmodes+1)) # node values array
for i in range(Nnodes):
    idx = i * Ndims
    # Add a PCA node using the node values, weights and variance
    xn[:, 0, 0] = pca.avg[0, idx:idx+Ndims] # add mean values
    xn[:, 0, 1:] = pca.v[idx:idx+Ndims,:Nmodes] # add 7 mode values
pcamesh.add_pcanode(i+1, xn, 'weights', 'variance', group='pca')
```

Finally we add the element and generate the mesh,

```python
pcamesh.add_element(1, ['L3'], [1, 2, 3, 4])
pcamesh.generate()
```

We can now modify the values of the weights and analyse or plot the mesh. Whenever the weights are change the mesh must be updated to recalculate the PCA node values. For example,

```python
pcamesh.nodes['weights'].values[2] = 1.3
pcamesh.update_pca_nodes()
```

To show the outputs of the PCA mesh, we plot the top 4 modes of the mesh for weights of -2, -1, 0, 1, and 2.

The python function used to plot these modes are below,

```python
def plot_modes(pcamesh, mode, sp, title=None):
    colors = {-2:'r', -1:'y', 0:'g', 1:'c', 2:'b'}
    pcamesh.nodes['weights'].values[1:] = 0 # Reset weights to zero
    pcamesh.update_pca_nodes()
    pcamesh.nodes['weights'].values[mode] = w
    pcamesh.update_pca_nodes()
    Xl = pcamesh.get_lines(32)[0]
    pylab.plot(Xl[:,0], Xl[:,1], colors[w])
```

```python
pylab.hold('off')
```
5.3. Creating a PCA Mesh
```python
pylab.xticks([])  
pylab.yticks([])  
if title != None:  
    pylab.title(title)
```
CHAPTER 6

Indices and tables

- genindex
- modindex
- search
morphic, 28
Index

M
morphic (module), 28