Chapter 3: Shape deformation


```
import igl
import scipy as sp
import numpy as np
from meshplot import plot, subplot, interact

import os
root_folder = os.getcwd()
```

Modern mesh-based shape deformation methods satisfy user deformation constraints at handles (selected vertices or regions on the mesh) and propagate these handle deformations to the rest of the shape *smoothly* and *without removing or distorting details*. Libigl provides implementations of a variety of state-of-the-art deformation techniques, ranging from quadratic mesh-based energy minimizers, to skinning methods, to non-linear elasticity-inspired techniques.

Biharmonic deformation

The period of research between 2000 and 2010 produced a collection of techniques that cast the problem of handle-based shape deformation as a quadratic energy minimization problem or equivalently the solution to a linear partial differential equation.

There are many flavors of these techniques, but a prototypical subset are those that consider solutions to the bi-Laplace equation, that is a biharmonic function *(Botsch, 2004)*. This fourth-order PDE provides sufficient flexibility in boundary conditions to ensure C^1 continuity at handle constraints in the limit under refinement *(Jacobson, 2010)*.

Biharmonic surfaces

Let us first begin our discussion of biharmonic *deformation*, by considering biharmonic *surfaces*. We will casually define biharmonic surfaces as surface whose *position functions* are biharmonic with respect to some initial parameterization:

$$\Delta^2 \mathbf{x}' = 0$$

and subject to some handle constraints, conceptualized as “boundary conditions”:

$$\mathbf{x}'_b = \mathbf{x}_{bc}.$$

where $\mathbf{x}'$ is the unknown 3D position of a point on the surface. So we are asking that the bi-Laplacian of each of spatial coordinate function to be zero.

In libigl, one can solve a biharmonic problem with `harmonic_weights` and setting $k = 2$ (*bi*-harmonic).

This produces a smooth surface that interpolates the handle constraints, but all original details on the surface will be *smoothed away*. Most obviously, if the original surface is not already biharmonic, then giving all handles the identity deformation (keeping them at their rest positions) will **not** reproduce the original surface. Rather, the result will be the biharmonic surface that does interpolate those handle positions.

Thus, we may conclude that this is not an intuitive technique for shape deformation.

Biharmonic deformation fields

Now we know that one useful property for a deformation technique is “rest pose reproduction”: applying no deformation to the handles should apply no deformation to the shape.

To guarantee this by construction we can work with *deformation fields* (ie. displacements) $\mathbf{d}$ rather than directly with positions $\mathbf{x}$. Then the deformed positions can be recovered as

$$\mathbf{x}' = \mathbf{x} + \mathbf{d}.$$

A smooth deformation field $\mathbf{d}$ which interpolates the deformation fields of the handle constraints will impose a smooth deformed shape $\mathbf{x}'$. Naturally, we consider *biharmonic deformation fields*:

$$\Delta^2 \mathbf{d} = 0$$

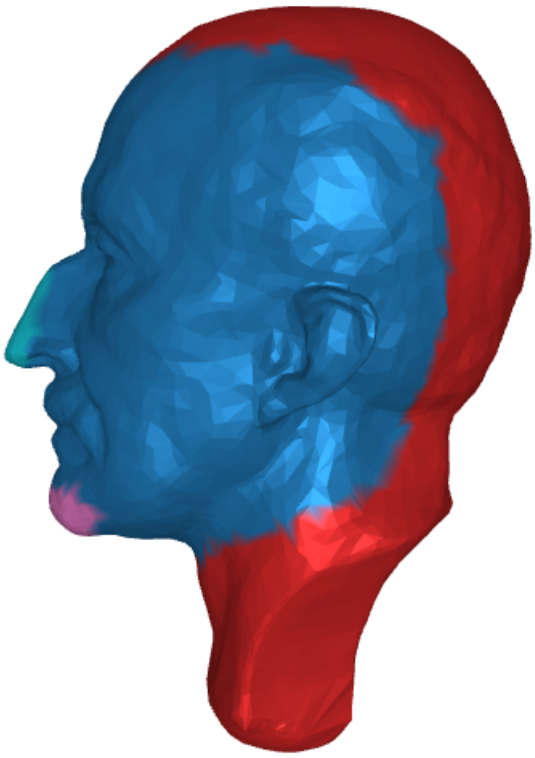
subject to the same handle constraints, but rewritten in terms of their implied deformation field at the boundary (handles).

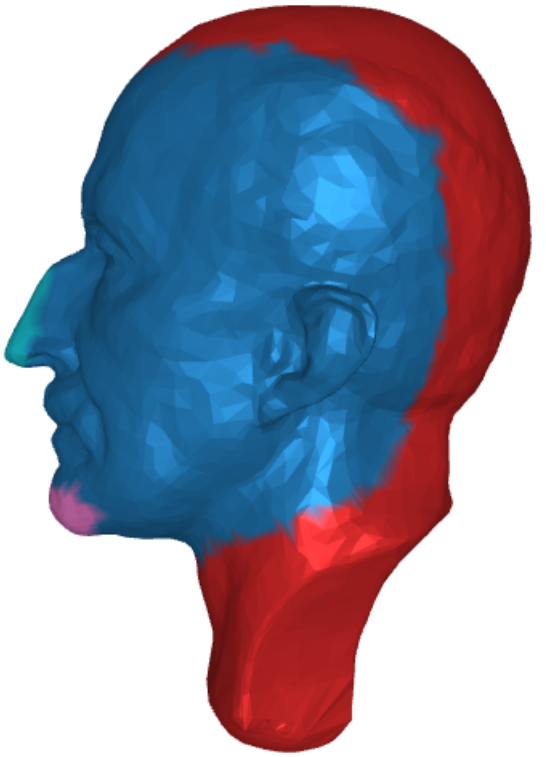
$$\mathbf{d}_b = \mathbf{x}_{bc} - \mathbf{x}_b.$$

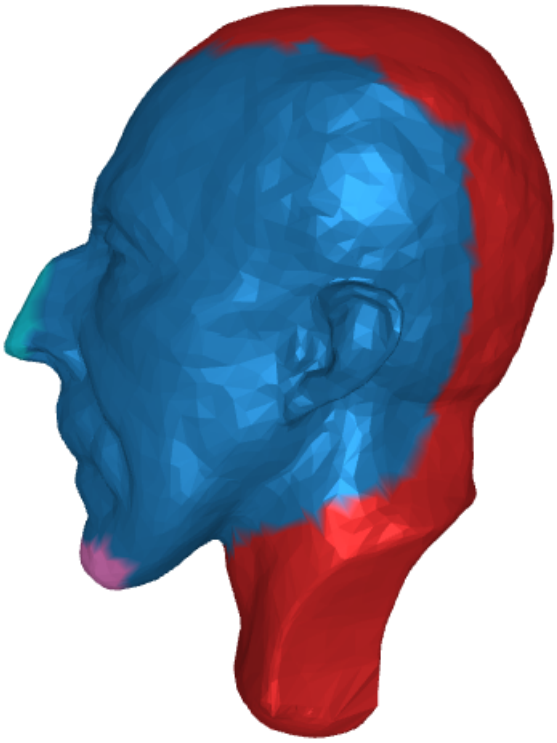
Again we can use `harmonic_weights` with $k = 2$, but this time solve for the deformation field and then recover the deformed positions:

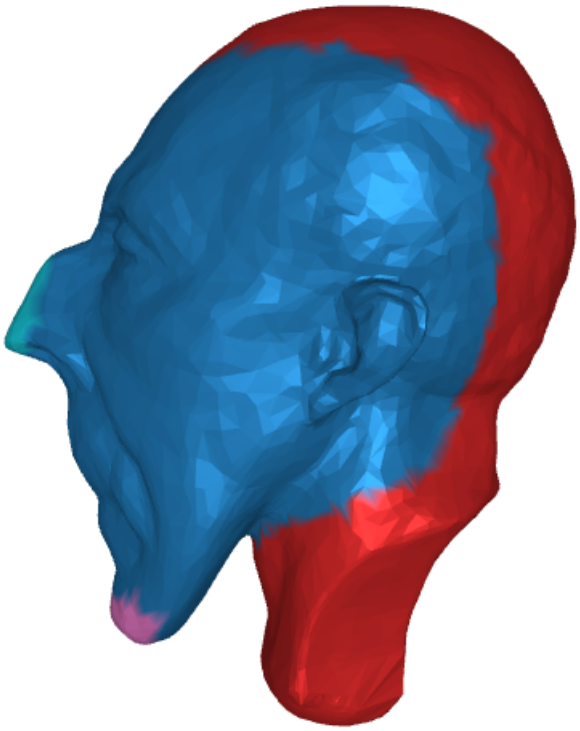
```
v, f = igl.read_triangle_mesh(os.path.join(root_folder, "data",
"decimated-max.obj"))
v[:,[0, 2]] = v[:,[2, 0]] # Swap X and Z axes
u = v.copy()

s = igl.read_dmat(os.path.join(root_folder, "data", "decimated-
max-selection.dmat"))
b = np.array([[t[0] for t in [(i, s[i]) for i in range(0,
v.shape[0])] if t[1] >= 0]]).T

## Boundary conditions directly on deformed positions
u_bc = np.zeros((b.shape[0], v.shape[1]))
v_bc = np.zeros((b.shape[0], v.shape[1]))

for bi in range(b.shape[0]):
    v_bc[bi] = v[b[bi]]

    if s[b[bi]] == 0: # Don't move handle 0
        u_bc[bi] = v[b[bi]]
    elif s[b[bi]] == 1: # Move handle 1 down
        u_bc[bi] = v[b[bi]] + np.array([[0, -50, 0]])
    else: # Move other handles forward
        u_bc[bi] = v[b[bi]] + np.array([[-25, 0, 0]])

p = subplot(v, f, s, shading={"wireframe": False, "colormap":
"tab10"}, s=[1, 4, 0])
for i in range(3):
    u_bc_anim = v_bc + i*0.6 * (u_bc - v_bc)
    d_bc = u_bc_anim - v_bc
    d = igl.harmonic_weights(v, f, b, d_bc, 2)
```

```
    u = v + d
    subplot(u, f, s, shading={"wireframe": False, "colormap":
"tab10"}, s=[1, 4, i+1], data=p)
p

# @interact(deformation_field=True, step=(0.0, 2.0))
# def update(deformation_field, step=0.0):
#     # Determine boundary conditions
#     u_bc_anim = v_bc + step * (u_bc - v_bc)

#     if deformation_field:
#         d_bc = u_bc_anim - v_bc
#         d = igl.harmonic_weights(v, f, b, d_bc, 2)
#         u = v + d
#     else:
#         u = igl.harmonic_weights(v, f, b, u_bc_anim, 2)
#     p.update_object(vertices=u)
```









Relationship to “differential coordinates” and Laplacian surface editing

Biharmonic functions (whether positions or displacements) are solutions to the bi-Laplace equation, but also minimizers of the “Laplacian energy”. For example, for displacements $\mathbf{d}$, the energy reads

$$\int_S \|\Delta \mathbf{d}\|^2 dA,$$

where we define $\Delta \mathbf{d}$ to simply apply the Laplacian coordinate-wise.

By linearity of the Laplace(-Beltrami) operator we can reexpress this energy in terms of the original positions $\mathbf{x}$ and the unknown positions $\mathbf{x}' = \mathbf{x} - \mathbf{d}$:

$$\int_S \|\Delta(\mathbf{x}' - \mathbf{x})\|^2 dA = \int_S \|\Delta \mathbf{x}' - \Delta \mathbf{x})\|^2 dA.$$

In the early work of Sorkine et al., the quantities $\Delta \mathbf{x}'$ and $\Delta \mathbf{x}$ were dubbed “differential coordinates” *(Sorkine, 2004)*. Their deformations (without linearized rotations) is thus equivalent to biharmonic deformation fields.

Polyharmonic deformation

We can generalize biharmonic deformation by considering different powers of the Laplacian, resulting in a series of PDEs of the form:

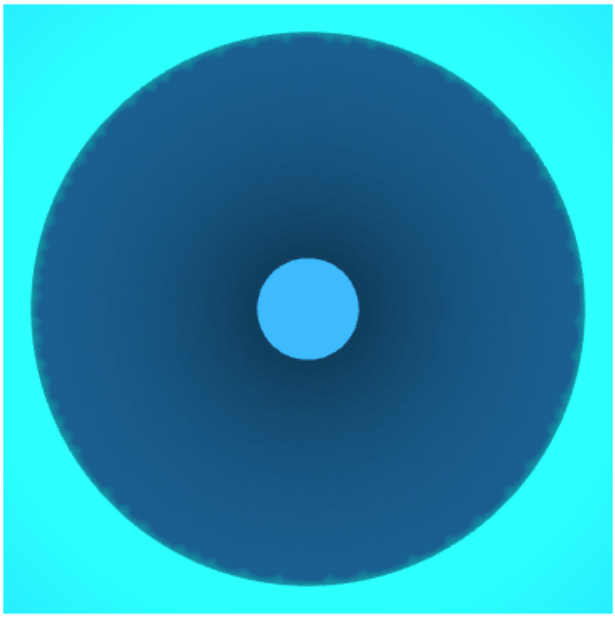
$$\Delta^k \mathbf{d} = 0.$$

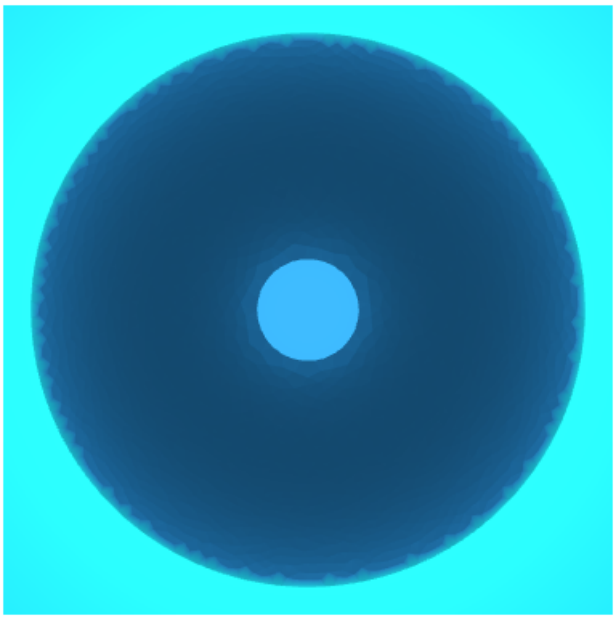
with $k \in 1, 2, 3, \ldots$. The choice of k determines the level of continuity at the handles. In particular, $k = 1$ implies C^0 at the boundary, $k = 2$ implies C^1, $k = 3$ implies C^2 and in general k implies C^{k-1}.

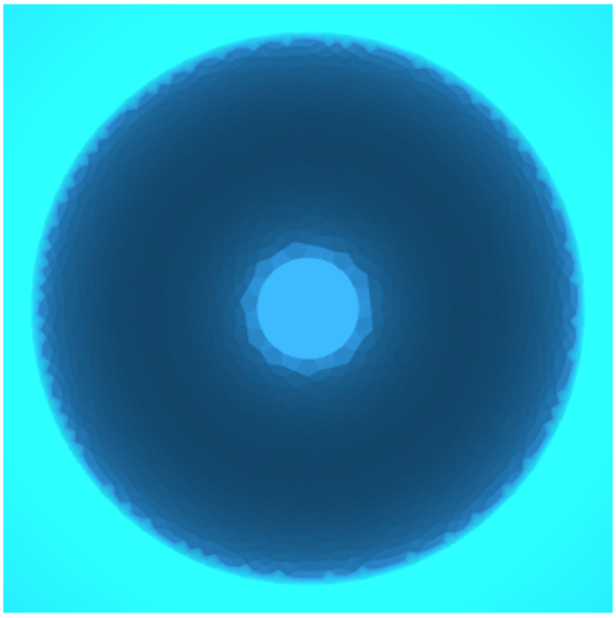
The following example deforms a flat domain (left) into a bump as a solution to various k-harmonic PDEs.

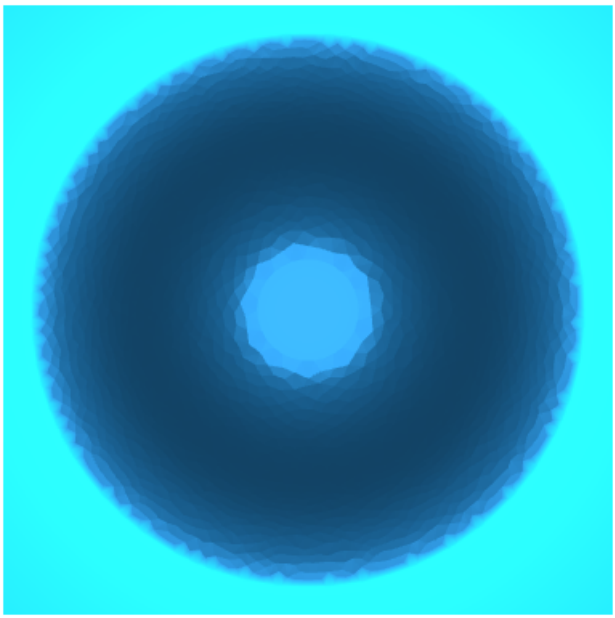
```
v, f = igl.read_triangle_mesh(os.path.join(root_folder, "data",
"bump-domain.obj"))
u = v.copy()

# Find boundary vertices outside annulus
vrn = np.linalg.norm(v, axis = 1)
is_outer = [vrn[i] - 1.00 > -1e-15 for i in range(v.shape[0])]
is_inner = [vrn[i] - 0.15 < 1e-15 for i in range(v.shape[0])]
in_b = [is_outer[i] or is_inner[i] for i in range(len(is_outer))]

b = np.array([i for i in range(v.shape[0]) if (in_b[i])]).T
bc = np.zeros(b.size)

for bi in range(b.size):
    bc[bi] = 0.0 if is_outer[b[bi]] else 1.0

c = np.array(is_outer)

for i in range(1,5):
    z = igl.harmonic_weights(v, f, b, bc, int(i))
    u[:, 2] = z
    if i == 1:
        p = subplot(u, f, c, shading={"wire_width": 0.01,
"colormap": "tab10"}, s=[1, 4, i-1])
    else:
        subplot(u, f, c, shading={"wire_width": 0.01, "colormap":
"tab10"}, s=[1, 4, i-1], data=p)
p

# p = plot(v, f, c, shading={"wire_width": 0.01, "colormap":
"tab10"})
```

```
# @interact(z_max=(0.0, 1.0), k=(1, 4))
# def update(z_max, k):
#     print(k)
#     z = igl.harmonic_weights(v, f, b, bc, int(k))
#     u[:, 2] = z_max * z
#     p.update_object(vertices=u)
```









As-rigid-as-possible

Skinning and other linear methods for deformation are inherently limited. Difficulties arise especially when large rotations are imposed by the handle constraints.

In the context of energy-minimization approaches, the problem stems from comparing positions (our displacements) in the coordinate frame of the undeformed shape. These quadratic energies are at best invariant to global rotations of the entire shape, but not smoothly varying local rotations. Thus linear techniques will not produce non-trivial bending and twisting.

Furthermore, when considering solid shapes (e.g. discretized with tetrahedral meshes) linear methods struggle to maintain local volume, and they often suffer from shrinking and bulging artifacts.

Non-linear deformation techniques present a solution to these problems. They work by comparing the deformation of a mesh vertex to its rest position *rotated* to a new coordinate frame which best matches the deformation. The non-linearity stems from the mutual dependence of the deformation and the best-fit

rotation. These techniques are often labeled “as-rigid-as-possible” as they penalize the sum of all local deformations’ deviations from rotations.

To arrive at such an energy, let’s consider a simple per-triangle energy:

$$E_{\text{linear}}(\mathbf{X}') = \sum_{t \in T} a_t \sum_{\{i,j\} \in t} w_{ij} \left\| \left(\mathbf{x}'_i - \mathbf{x}'_j \right) - \left(\mathbf{x}_i - \mathbf{x}_j \right) \right\|^2$$

where $\mathbf{X}'$ are the mesh’s unknown deformed vertex positions, t is a triangle in a list of triangles T, a_t is the area of triangle t and $\{i, j\}$ is an edge in triangle t. Thus, this energy measures the norm of change between an edge vector in the original mesh $\left(\mathbf{x}_i - \mathbf{x}_j \right)$ and the unknown mesh $\left(\mathbf{x}'_i - \mathbf{x}'_j \right)$.

This energy is **not** rotation invariant. If we rotate the mesh by 90 degrees the change in edge vectors not aligned with the axis of rotation will be large, despite the overall deformation being perfectly rigid.

So, the “as-rigid-as-possible” solution is to append auxiliary variables $\mathbf{R}_t$ for each triangle t which are constrained to be rotations. Then the energy is rewritten, this time comparing deformed edge vectors to their rotated rest counterparts:

$$E_{\text{arap}}(\mathbf{X}', \{\mathbf{R}_1, \ldots, \mathbf{R}_{|T|}\}) = \sum_{t \in T} a_t \sum_{\{i,j\} \in t} w_{ij} \left\| \left(\mathbf{x}'_i - \mathbf{x}'_j \right) - \mathbf{R}_t \left(\mathbf{x}_i - \mathbf{x}_j \right) \right\|^2.$$

The separation into the primary vertex position variables $\mathbf{X}'$ and the rotations $\{\mathbf{R}_1, \ldots, \mathbf{R}_{|T|}\}$ lead to strategy for optimization, too. If the rotations $\{\mathbf{R}_1, \ldots, \mathbf{R}_{|T|}\}$ are held fixed then the energy is quadratic in the remaining variables $\mathbf{X}'$ and can be optimized by solving a (sparse) global linear system. Alternatively, if $\mathbf{X}'$ are held fixed then each rotation is the solution to a localized *Procrustes* problem (found via 3×3 SVD or polar decompostion). These two steps—local and global—each weakly decrease the energy, thus we may safely iterate them until convergence.

The different flavors of “as-rigid-as-possible” depend on the dimension and codimension of the domain and the edge-sets T. The proposed surface manipulation technique by Sorkine and Alexa *(Sorkine, 2007)*, considers T to be the set of sets of edges emanating from each vertex (spokes). Later, Chao et al. derived the relationship between “as-rigid-as-possible” mesh energies and co-

rotational elasticity considering 0-codimension elements as edge-sets: triangles in 2D and tetrahedra in 3D *(Chao, 2010)*. They also showed how Sorkine and Alexa's edge-sets are not a discretization of a continuous energy, proposing instead edge-sets for surfaces containing all edges of elements incident on a vertex (spokes and rims). They show that this amounts to measuring bending, albeit in a discretization-dependent way.

Libigl, supports these common flavors. Selecting one is a matter of setting the energy type before the precompuation phase.

```
#arap_data.energy = igl::ARAP_ENERGY_TYPE_SPOKES;
#arap_data.energy = igl::ARAP_ENERGY_TYPE_SPOKES_AND_RIMS;
#arap_data.energy = igl::ARAP_ENERGY_TYPE_ELEMENTS;
arap = igl.ARAP(v, f, dimension, b)
```

Just like `igl.min_quad_with_fixed_*`, this precomputation phase only depends on the mesh, fixed vertex indices `b` and the energy parameters. To solve with certain constraints on the positions of vertices in `b`, we may call:

```
vn = arap.solve(bc, v)
```

which uses `v` as an initial guess and then computes the solution into it.

Libigl's implementation of as-rigid-as-possible deformation takes advantage of the highly optimized singular value decomposition code from McAdams et al. *(McAdams, 2011)* which leverages SSE intrinsics.

The following example deforms a surface as if it were made of an elastic material. The concept of local rigidity will be revisited shortly in the context of surface parameterization.

```
v, f = igl.read_triangle_mesh(os.path.join(root_folder, "data",
"decimated-knight.off"))
s = igl.read_dmat(os.path.join(root_folder, "data", "decimated-
knight-selection.dmat"))

# Vertices in selection
b = np.array([[t[0] for t in [(i, s[i]) for i in range(0,
v.shape[0])]
      if t[1] >= 0]]).T
```

```
# Centroid
mid = 0.5 * (np.max(v, axis=0) + np.min(v, axis=0))

# Precomputation
arap = igl.ARAP(v, f, 3, b)

# Set color based on selection
c = np.ones_like(f) * np.array([1.0, 228/255, 58/255])
for fi in range(0, f.shape[0]):
    if s[f[fi, 0]] >= 0 and s[f[fi, 1]] >= 0 and s[f[fi, 2]] >= 0:
        c[fi] = np.array([80/255, 64/255, 1.0])

# Plot the mesh with pseudocolors
p = subplot(v, f, c, s=[1, 4, 0])
for k in range(3):
    t= 1 + k*3
    bc = np.zeros((b.size, v.shape[1]))
    for i in range(0, b.size):
        bc[i] = v[b[i]]
        if s[b[i]] == 0:
            r = mid[0] * 0.25
            bc[i, 0] += r * np.sin(0.5 * t * 2 * np.pi)
            bc[i, 1] = bc[i, 1] - r + r * np.cos(np.pi + 0.5 * t *
2 * np.pi)
        elif s[b[i]] == 1:
            r = mid[1] * 0.15
            bc[i, 1] = bc[i, 1] + r + r * np.cos(np.pi + 0.15 * t
* 2 * np.pi)
            bc[i, 2] -= r * np.sin(0.15 * t * 2 * np.pi)
        elif s[b[i]] == 2:
            r = mid[1] * 0.15
            bc[i, 2] = bc[i, 2] + r + r * np.cos(np.pi + 0.35 * t
* 2 * np.pi)
            bc[i, 0] += r * np.sin(0.35 * t * 2 * np.pi)

    vn = arap.solve(bc, v)
    subplot(vn, f, c, s=[1, 4, k+1], data=p)
p


# p = plot(v, f, c, return_plot=True)

# @interact(t=(0.0, 10.0))
# def update(t=1.0):
#     bc = np.zeros((b.size, v.shape[1]))
#     for i in range(0, b.size):
#         bc[i] = v[b[i]]
#         if s[b[i]] == 0:
#             r = mid[0] * 0.25
```

```
#             bc[i, 0] += r * np.sin(0.5 * t * 2 * np.pi)
#             bc[i, 1] = bc[i, 1] - r + r * np.cos(np.pi + 0.5 * t
* 2 * np.pi)
#         elif s[b[i]] == 1:
#             r = mid[1] * 0.15
#             bc[i, 1] = bc[i, 1] + r + r * np.cos(np.pi + 0.15 *
t * 2 * np.pi)
#             bc[i, 2] -= r * np.sin(0.15 * t * 2 * np.pi)
#         elif s[b[i]] == 2:
#             r = mid[1] * 0.15
#             bc[i, 2] = bc[i, 2] + r + r * np.cos(np.pi + 0.35 *
t * 2 * np.pi)
#             bc[i, 0] += r * np.sin(0.35 * t * 2 * np.pi)

#     vn = arap.solve(bc, v)
#     p.update_object(vertices=vn)
```

References

1. Jernej Barbic and Doug James. Real-Time Subspace Integration for St.Venant-Kirchhoff Deformable Models, 2005. ↩
2. Klaus Hildebrandt, Christian Schulz, Christoph von Tycowicz, and Konrad Polthier. Interactive Surface Modeling using Modal Analysis, 2011. ↩
3. Raid M. Rustamov, Multiscale Biharmonic Kernels, 2011. ↩
4. Bruno Vallet and Bruno Lévy. Spectral Geometry Processing with Manifold Harmonics, 2008.

↩