Chapter 2: Matrices and Linear Algebra


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

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

Laplace equation

A common linear system in geometry processing is the Laplace equation:

$$\Delta z = 0$$

subject to some boundary conditions, for example Dirichlet boundary conditions (fixed value):

$$z|_{\partial S} = z_{bc}$$

In the discrete setting, the linear system can be written as:

$$\mathbf{L}\mathbf{z} = \mathbf{0}$$

where $\mathbf{L}$ is the $n \times n$ discrete Laplacian and $\mathbf{z}$ is a vector of per-vertex values. Most of $\mathbf{z}$ correspond to interior vertices and are unknown, but some of $\mathbf{z}$ represent values at boundary vertices. Their values are known so we may move their corresponding terms to the right-hand side.

Conceptually, this is very easy if we have sorted $\mathbf{z}$ so that interior vertices come first and then boundary vertices:

$$\begin{pmatrix} \mathbf{L}_{in,in} & \mathbf{L}_{in,b} \\ \mathbf{L}_{b,in} & \mathbf{L}_{b,b} \end{pmatrix} \begin{pmatrix} \mathbf{z}_{in} \\ \mathbf{z}_{b} \end{pmatrix} = \begin{pmatrix} \mathbf{0}_{in} \\ \mathbf{z}_{bc} \end{pmatrix}$$

The bottom block of equations is no longer meaningful so we'll only consider the top block:

$$\begin{pmatrix} \mathbf{L}_{in,in} & \mathbf{L}_{in,b} \end{pmatrix} \begin{pmatrix} \mathbf{z}_{in} \\ \mathbf{z}_{b} \end{pmatrix} = \mathbf{0}_{in}$$

We can move the known values to the right-hand side:

$$\mathbf{L}_{in,in}\mathbf{z}_{in} = -\mathbf{L}_{in,b}\mathbf{z}_{b}$$

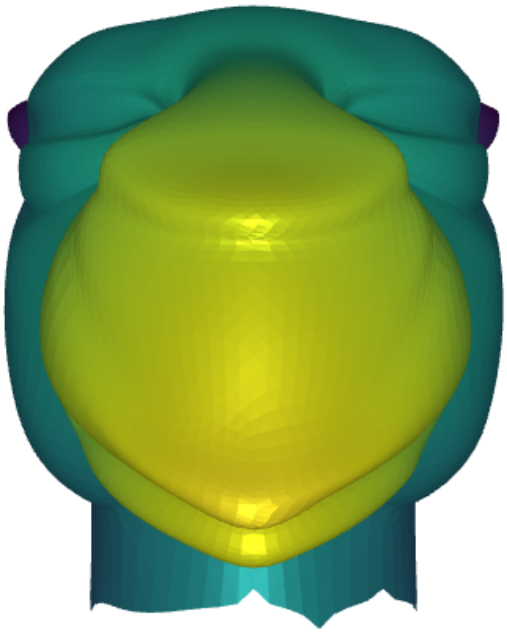
Finally we can solve this equation for the unknown values at interior vertices $\mathbf{z}_{in}$.

However, our vertices will often not be sorted in this way. One option would be to sort `V`, then proceed as above and then *unsort* the solution `Z` to match `V`. However, this solution is not very general.

With array slicing no explicit sort is needed. Instead we can *slice-out* submatrix blocks ($\mathbf{L}_{in,in}$, $\mathbf{L}_{in,b}$, etc.) and follow the linear algebra above directly. Then we can slice the solution *into* the rows of `Z` corresponding to the interior vertices.

```
from scipy.sparse.linalg import spsolve

v, f = igl.read_triangle_mesh(os.path.join(root_folder, "data",
"camelhead.off"))

## Find boundary vertices
e = igl.boundary_facets(f)
v_b = np.unique(e)

## List of all vertex indices
v_all = np.arange(v.shape[0])

## List of interior indices
v_in = np.setdiff1d(v_all, v_b)

## Construct and slice up Laplacian
l = igl.cotmatrix(v, f)
l_ii = l[v_in, :]
l_ii = l_ii[:, v_in]
```

```
l_ib = l[v_in, :]
l_ib = l_ib[:, v_b]

## Dirichlet boundary conditions from z-coordinate
z = v[:, 2]
bc = z[v_b]

## Solve PDE
z_in = spsolve(-l_ii, l_ib.dot(bc))

plot(v, f, z)
```



Quadratic energy minimization

The same Laplace equation may be equivalently derived by minimizing Dirichlet energy subject to the same boundary conditions:

$$\text{minimize}_{\ z} \frac{1}{2} \int_S \|\nabla z\|^2 dA$$

On our discrete mesh, recall that this becomes

$$\text{minimize}_{\ \mathbf{z}} \frac{1}{2}\mathbf{z}^T\mathbf{G}^T\mathbf{D}\mathbf{G}\mathbf{z} \rightarrow \text{minimize}_{\ \mathbf{z}} \mathbf{z}^T\mathbf{L}\mathbf{z}$$

The general problem of minimizing some energy over a mesh subject to fixed value boundary conditions is so wide spread that libigl has a dedicated api for solving such systems.

Let us consider a general quadratic minimization problem subject to different common constraints:

$$\underset{\mathbf{z}}{\text{minimize}} \ \frac{1}{2}\mathbf{z}^T\mathbf{Q}\mathbf{z} + \mathbf{z}^T\mathbf{B} + \text{constant},$$

subject to

$$\mathbf{z}_b = \mathbf{z}_{bc} \text{ and } \mathbf{A}_{eq}\mathbf{z} = \mathbf{B}_{eq},$$

where

- $\mathbf{Q}$ is a (usually sparse) $n \times n$ positive semi-definite matrix of quadratic coefficients (Hessian),
- $\mathbf{B}$ is a $n \times 1$ vector of linear coefficients,
- $\mathbf{z}_b$ is a $|b| \times 1$ portion of $\mathbf{z}$ corresponding to boundary or *fixed* vertices,
- $\mathbf{z}_{bc}$ is a $|b| \times 1$ vector of known values corresponding to $\mathbf{z}_b$,
- $\mathbf{A}_{eq}$ is a (usually sparse) $m \times n$ matrix of linear equality constraint coefficients (one row per constraint), and
- $\mathbf{B}_{eq}$ is a $m \times 1$ vector of linear equality constraint right-hand side values.

This specification is overly general as we could write $\mathbf{z}_b = \mathbf{z}_{bc}$ as rows of $\mathbf{A}_{eq}\mathbf{z} = \mathbf{B}_{eq}$, but these fixed value constraints appear so often that they merit a dedicated function: `min_quad_with_fixed`.

Linear equality constraints

We saw above that `min_quad_with_fixed` in libigl provides a compact way to solve general quadratic programs. Let's consider another example, this time with active linear equality constraints. Specifically let's solve the `bi-Laplace equation` or equivalently minimize the Laplace energy:

$$\Delta^2 z = 0 \leftrightarrow \underset{z}{\text{minimize}} \ \frac{1}{2}\int_S (\Delta z)^2 dA$$

subject to fixed value constraints and a linear equality constraint:

$z_a = 1, z_b = -1$ and $z_c = z_d$.

Notice that we can rewrite the last constraint in the familiar form from above:

$z_c - z_d = 0$.

Now we can assembly `Aeq` as a $1 \times n$ sparse matrix with a coefficient 1 in the column corresponding to vertex c and a -1 at d. The right-hand side `Beq` is simply zero.

Internally, `min_quad_with_fixed` solves using the Lagrange Multiplier method. This method adds additional variables for each linear constraint (in general a $m \times 1$ vector of variables λ) and then solves the saddle problem:

$$\underset{\mathbf{z},\lambda}{\text{find saddle}} \ \frac{1}{2}\mathbf{z}^T\mathbf{Q}\mathbf{z} + \mathbf{z}^T\mathbf{B} + \text{constant} + \lambda^T\left(\mathbf{A}_{eq}\mathbf{z} - \mathbf{B}_{eq}\right)$$

This can be rewritten in a more familiar form by stacking $\mathbf{z}$ and λ into one $(m+n) \times 1$ vector of unknowns:

$$\underset{\mathbf{z},\lambda}{\text{find saddle}} \ \frac{1}{2}\begin{pmatrix}\mathbf{z}^T & \lambda^T\end{pmatrix}\begin{pmatrix}\mathbf{Q} & \mathbf{A}_{eq}^T \\ \mathbf{A}_{eq} & 0\end{pmatrix}\begin{pmatrix}\mathbf{z} \\ \lambda\end{pmatrix} + \begin{pmatrix}\mathbf{z}^T & \lambda^T\end{pmatrix}\begin{pmatrix}\mathbf{B} \\ -\mathbf{B}_{eq}\end{pmatrix} + \text{consta}$$

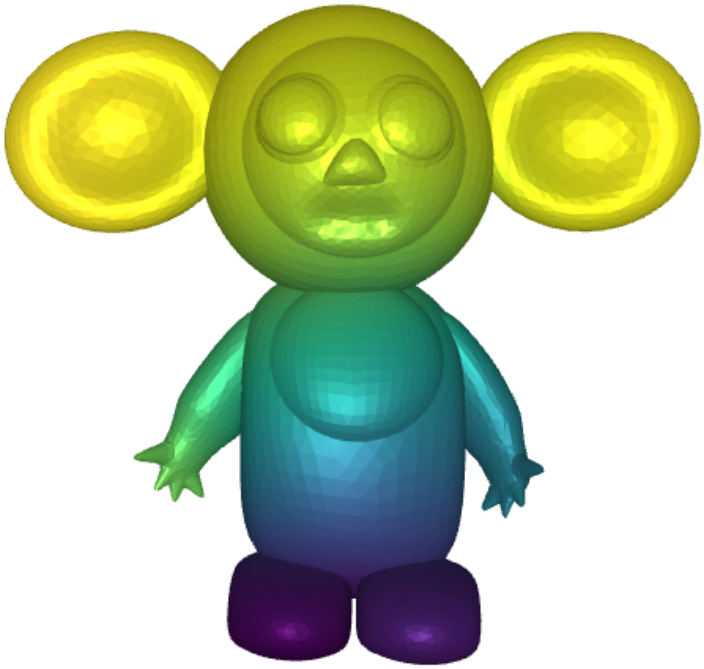
Differentiating with respect to $\left(\mathbf{z}^T \lambda^T\right)$ reveals a linear system and we can solve for $\mathbf{z}$ and λ. The only difference from the straight quadratic *minimization* system, is that this saddle problem system will not be positive definite. Thus, we must use a different factorization technique (LDLT rather than LLT): libigl's `min_quad_with_fixed` automatically chooses the correct solver in the presence of linear equality constraints.

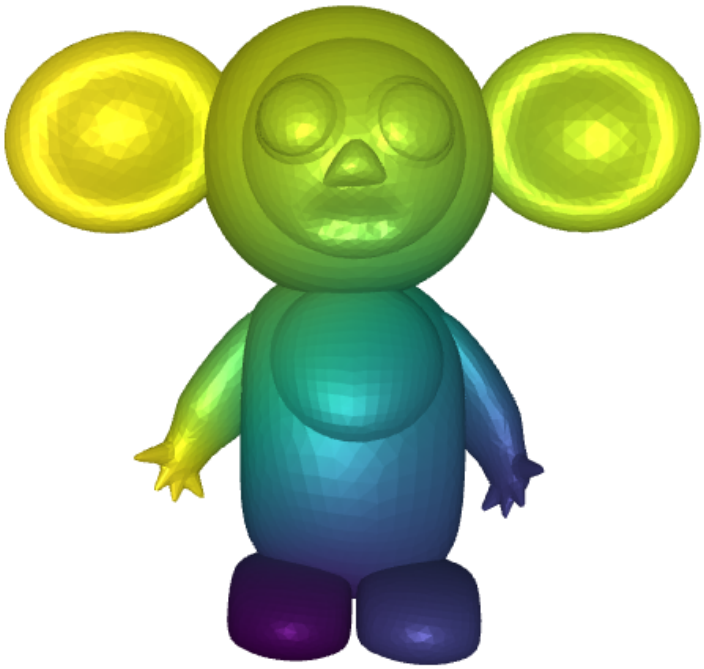
The following example first solves with just fixed value constraints (left: 1 and -1 on the left hand and foot respectively), then solves with an additional linear equality constraint (right: points on right hand and foot constrained to be equal).

```
v, f = igl.read_triangle_mesh(os.path.join(root_folder, "data",
"cheburashka.off"))

## Two fixed points: Left hand, left foot should have values 1 and
-1
b = np.array([4331, 5957])
bc = np.array([1., -1.])
B = np.zeros((v.shape[0], 1))

## Construct Laplacian and mass matrix
L = igl.cotmatrix(v, f)
M = igl.massmatrix(v, f, igl.MASSMATRIX_TYPE_VORONOI)
Minv = sp.sparse.diags(1 / M.diagonal())

## Bi-Laplacian
Q = L @ (Minv @ L)

## Solve with only equality constraints
Aeq = sp.sparse.csc_matrix((0, 0))
Beq = np.array([])
_, z1 = igl.min_quad_with_fixed(Q, B, b, bc, Aeq, Beq, True)

## Solve with equality and linear constraints
Aeq = sp.sparse.csc_matrix((1, v.shape[0]))
Aeq[0,6074] = 1
Aeq[0, 6523] = -1
Beq = np.array([0.])
_, z2 = igl.min_quad_with_fixed(Q, B, b, bc, Aeq, Beq, True)

## Normalize colors to same range
min_z = min(np.min(z1), np.min(z2))
max_z = max(np.max(z1), np.max(z2))
z = [(z1 - min_z) / (max_z - min_z), (z2 - min_z) / (max_z -
min_z)]
```

```
## Plot the functions
p = subplot(v, f, z[0], shading={"wireframe":False}, s=[1, 2, 0])
subplot(v, f, z[1], shading={"wireframe":False}, s=[1, 2, 1],
data=p)

# @interact(function=[('z0', 0), ('z1', 1)])
# def sf(function):
#     p.update_object(colors=z[function])
```

```
/home/sebastian/Tools/miniconda3/envs/bcp/lib/python3.8/site-
packages/scipy/sparse/_index.py:84:
SparseEfficiencyWarning: Changing the sparsity structure
of a csc_matrix is expensive. lil_matrix is more
efficient.
  self._set_intXint(row, col, x.flat[0])
```





Quadratic programming

We can generalize the quadratic optimization in the previous section even more by allowing inequality constraints. Specifically box constraints (lower and upper bounds):

$$\mathbf{l} \leq \mathbf{z} \leq \mathbf{u},$$

where $\mathbf{l}, \mathbf{u}$ are $n \times 1$ vectors of lower and upper bounds and general linear inequality constraints:

$$\mathbf{A}_{ieq}\mathbf{z} \leq \mathbf{B}_{ieq},$$

where $\mathbf{A}_{ieq}$ is a $k \times n$ matrix of linear coefficients and $\mathbf{B}_{ieq}$ is a $k \times 1$ matrix of constraint right-hand sides.

Again, we are overly general as the box constraints could be written as rows of the linear inequality constraints, but bounds appear frequently enough to merit a dedicated api.

Libigl implements its own active set routine for solving *quadratric programs* (QPs). This algorithm works by iteratively "activating" violated inequality constraints by enforcing them as equalities and "deactivating" constraints which are no longer needed.

After deciding which constraints are active at each iteration, the problem reduces to a quadratic minimization subject to linear *equality* constraints, and the method from the previous section is invoked. This is repeated until convergence.

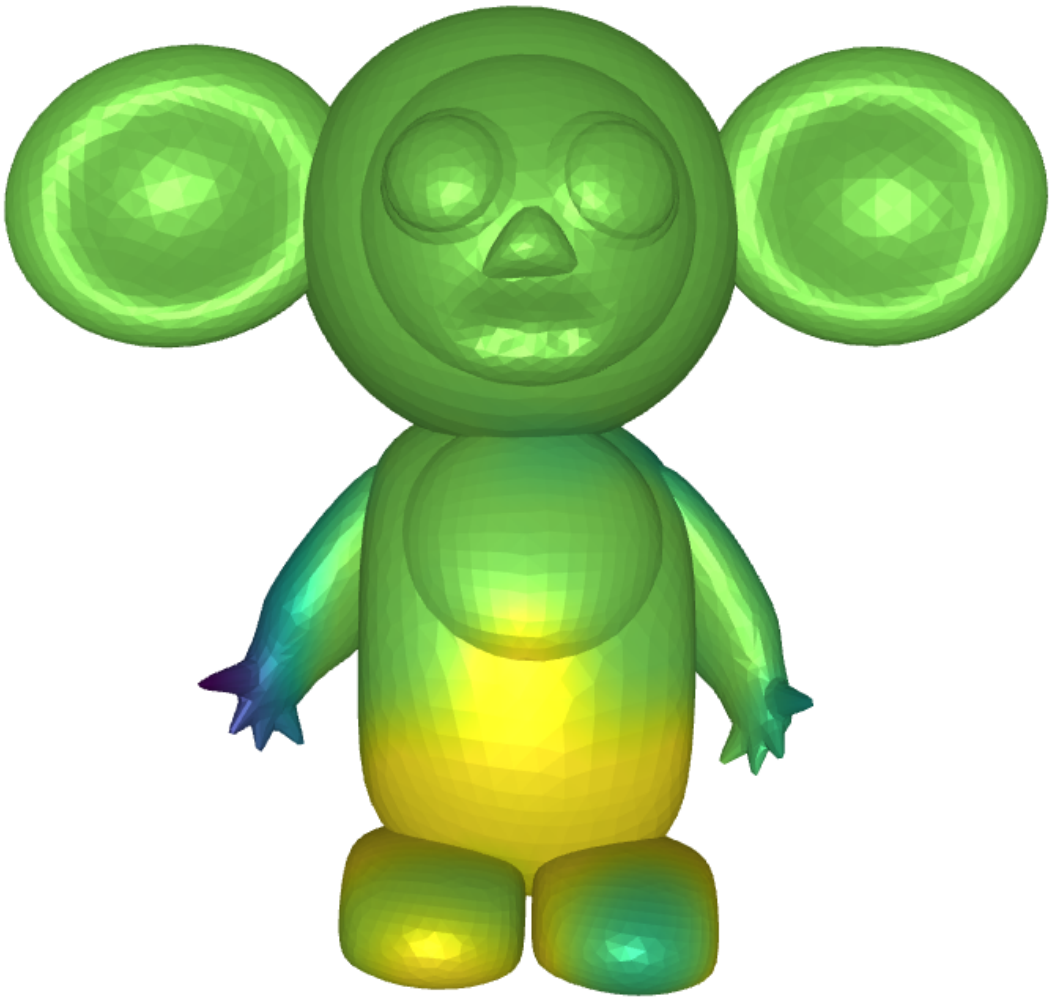
Currently the implementation is efficient for box constraints and sparse non-overlapping linear inequality constraints.

Unlike alternative interior-point methods, the active set method benefits from a warm-start (initial guess for the solution vector $\mathbf{z}$).

The following example uses an active set solver to optimize discrete biharmonic kernels *(Rustamov, 2011)* at multiple scales:

```
#TODO: Check why results differ, add interactivity

v, f, _ = igl.read_off(os.path.join(root_folder, "data",
"cheburashka.off"))

# One fixed point on belly
b = np.array([[2556]])
bc = np.array([[1.0]])

# Construct Laplacian and mass matrix
l = igl.cotmatrix(v, f)
m = igl.massmatrix(v, f, igl.MASSMATRIX_TYPE_VORONOI)
minv = sp.sparse.diags(1 / m.diagonal())

# Bi-Laplacian
q = l @ (minv @ l)

# Zero linear term
bz = np.zeros((v.shape[0], 1))

# Lower and upper bound
lx = np.zeros((v.shape[0], 1))
ux = np.ones((v.shape[0], 1))

# Equality constraint constrains solution to sum to 1
beq = np.array([[0.08]])
aeq = sp.sparse.csc_matrix(m.diagonal())
```

```
# Empty inequality constraints
aieq = sp.sparse.csc_matrix((0, 0))
bieq = np.array([])

z = igl.active_set(q, bz, b, bc, aeq, beq, aieq, bieq, lx, ux,
max_iter=8)
plot(v, f, z[1])
```



Eigen Decomposition

Libigl has rudimentary support for extracting eigen pairs of a generalized eigen value problem:

$$Ax = \lambda Bx$$

where A is a sparse symmetric matrix and B is a sparse positive definite matrix. Most commonly in geometry processing, we let $A = L$ the cotangent Laplacian and $B = M$ the per-vertex mass matrix *(Vallet, 2008)*. Typically applications will make use of the *low frequency* eigen modes. Analogous to the Fourier decomposition, a function f on a surface can be represented via its spectral decomposition of the eigen modes of the Laplace-Beltrami:

$$f = \sum_{i=1}^{\infty} a_i \phi_i$$

where each ϕ_i is an eigen function satisfying: $\Delta \phi_i = \lambda_i \phi_i$ and a_i are scalar coefficients. For a discrete triangle mesh, a completely analogous decomposition exists, albeit with finite sum:

$$\mathbf{f} = \sum_{i=1}^{n} a_i \phi_i$$

where now a column vector of values at vertices $\mathbf{f} \in \mathcal{R}^n$ specifies a piecewise linear function and $\phi_i \in \mathcal{R}^n$ is an eigen vector satisfying:

$$\mathbf{L}\phi_i = \lambda_i \mathbf{M} \phi_i.$$

Note that Vallet & Levy *(Vallet, 2008)* propose solving a symmetrized *standard* eigen problem $\mathbf{M}^{-1/2}\mathbf{L}\mathbf{M}^{-1/2}\phi_i = \lambda_i \phi_i$. Libigl implements a generalized eigen problem solver so this unnecessary symmetrization can be avoided.

Often the sum above is *truncated* to the first k eigen vectors. If the low frequency modes are chosen, i.e. those corresponding to small λ_i values, then this truncation effectively *regularizes* $\mathbf{f}$ to smooth, slowly changing functions over the mesh *(Hildebrandt, 2011)*. Modal analysis and model subspaces have been used frequently in real-time deformation *(Barbic, 2005)*.

In the following example, the first k eigen vectors of the discrete Laplace-Beltrami operator are computed and displayed in pseudocolors atop the beetle. Low frequency eigen vectors of the discrete Laplace-Beltrami operator vary smoothly and slowly over the model. At first, calculate the Laplace-Betrami operator and solve the generalized Eigen problem with scipy/arpack. Then, rescale the Eigen vectors and visualize them.

```
v, f = igl.read_triangle_mesh(os.path.join(root_folder, "data",
"beetle.off"))
l = -igl.cotmatrix(v, f)
m = igl.massmatrix(v, f, igl.MASSMATRIX_TYPE_VORONOI)

k = 10
d, u = sp.sparse.linalg.eigsh(l, k, m, sigma=0, which="LM")

u = (u - np.min(u)) / (np.max(u) - np.min(u))
bbd = 0.5 * np.linalg.norm(np.max(v, axis=0) - np.min(v, axis=0))

p = subplot(v, f, bbd * u[:, 0], shading={"wireframe":False,
"flat": False}, s=[1, 2, 0])
subplot(v, f, bbd * u[:, 1], shading={"wireframe":False, "flat":
False}, s=[1, 2, 1], data=p)

# @interact(ev=[("EV %i"%i, i) for i in range(k)])
# def sf(ev):
#     p.update_object(colors=u[:, ev])
```

References

1. Alec Jacobson, *Algorithms and Interfaces for Real-Time Deformation of 2D and 3D Shapes*, 2013. ↩
2. Michael Kazhdan, Jake Solomon, Mirela Ben-Chen, Can Mean-Curvature Flow Be Made Non-Singular, 2012. ↩
3. Mark Meyer, Mathieu Desbrun, Peter Schröder and Alan H. Barr, Discrete Differential-Geometry Operators for Triangulated 2-Manifolds, 2003. ↩
4. Joseph S. B. Mitchell, David M. Mount, Christos H. Papadimitriou. The Discrete Geodesic Problem, 1987 ↩
5. Daniele Panozzo, Enrico Puppo, Luigi Rocca, Efficient Multi-scale Curvature and Crease Estimation, 2010. ↩
6. Andrei Sharf, Thomas Lewiner, Gil Shklarski, Sivan Toledo, and Daniel Cohen-Or. Interactive topology-aware surface reconstruction, 2007.

↩