H❤️rtDown: Document Processor for Executable Linear Algebra Papers

Yong Li

GEORGE MASON UNIVERSITY CraGL Computational Reality Creativity and Graphics Lab

Shoaib Kamil

Adobe

Alec Jacobson

University of Toronto Adobe

Yotam Gingold

GEORGE MASON UNIVERSITY CraGL Computational Reality Creativity and Graphics Lab

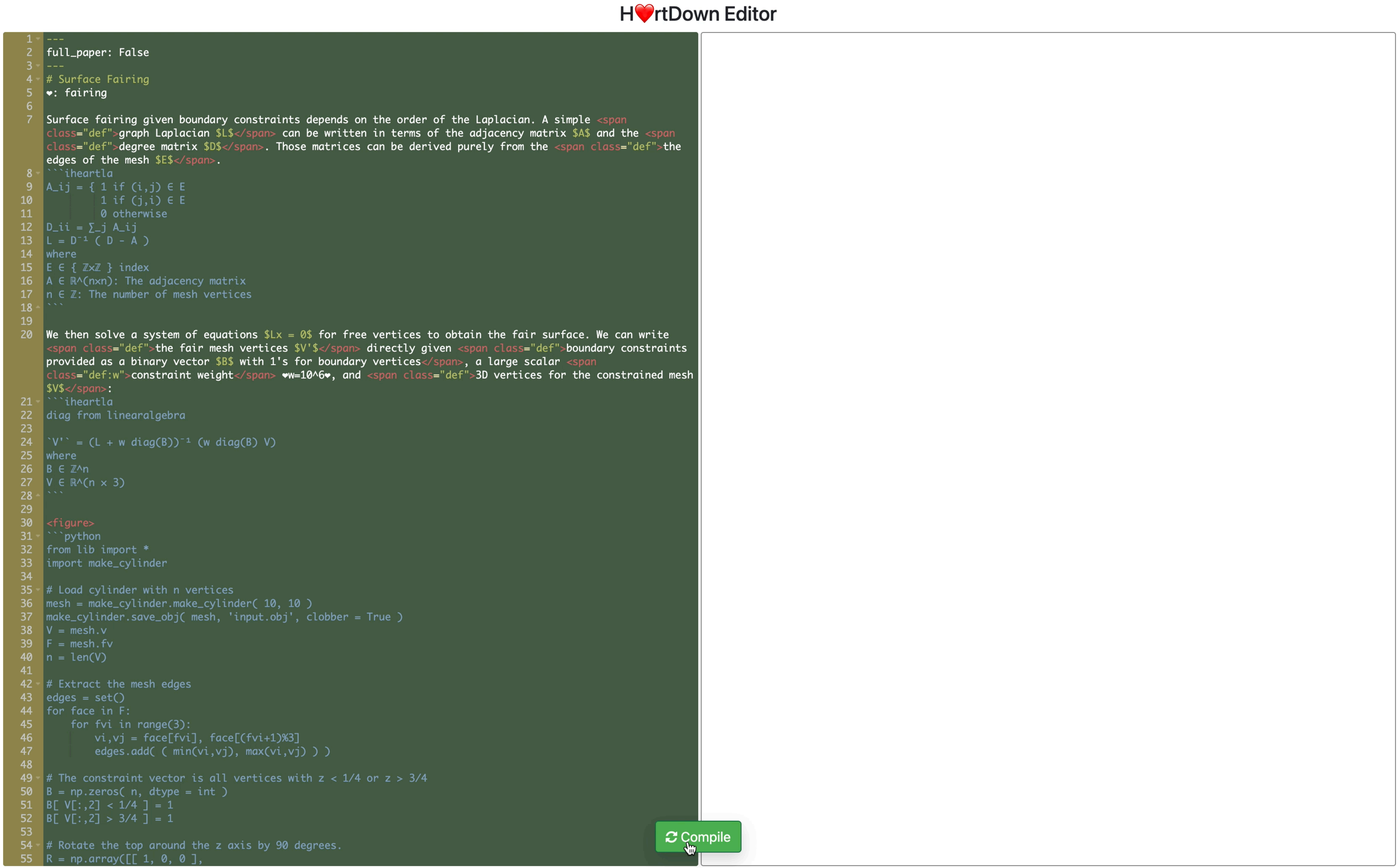
H❤️rtDown

~~~
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those matrices can be derived purely from the <span class="def">the edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤ×ℤ } index
A ∈ ℝ^(n×n): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
~~~

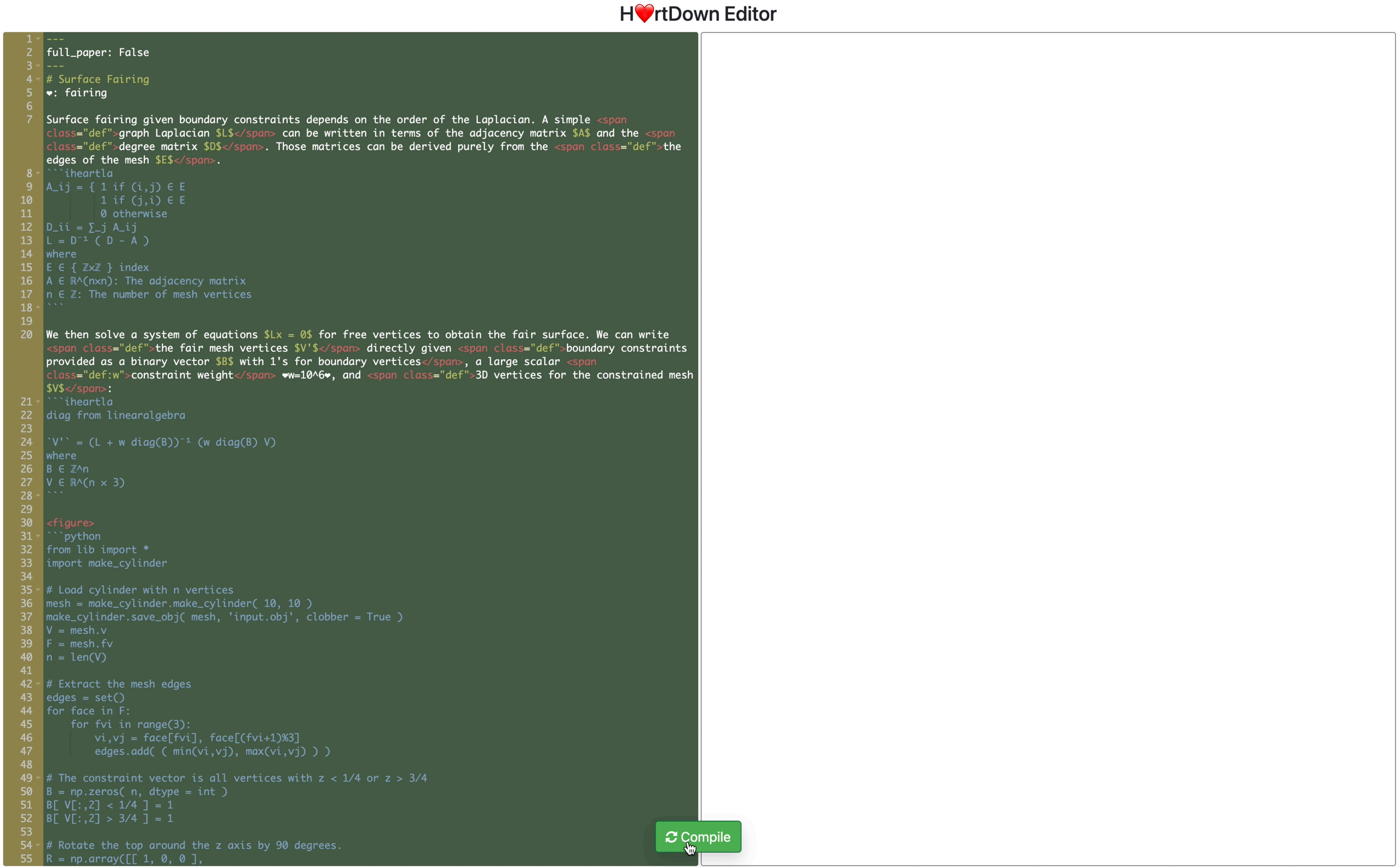
We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write <span class="def">the fair mesh vertices V'</span> directly given <span class="def">boundary constraints provided as a binary vector B with 1's for boundary vertices</span>, a large scalar <span class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh V</span>:
```iheartla
diag from linearalgebra

`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
where
```

~~~
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those matrices can be derived purely from the <span class="def">the edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤxℤ } index
A ∈ ℝ^(nxn): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
~~~

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write <span class="def">the fair mesh vertices V'</span> directly given <span class="def">boundary constraints provided as a binary vector B with 1's for boundary vertices</span>, a large scalar <span class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh V</span>:
```iheartla
diag from linearalgebra

`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
where
```



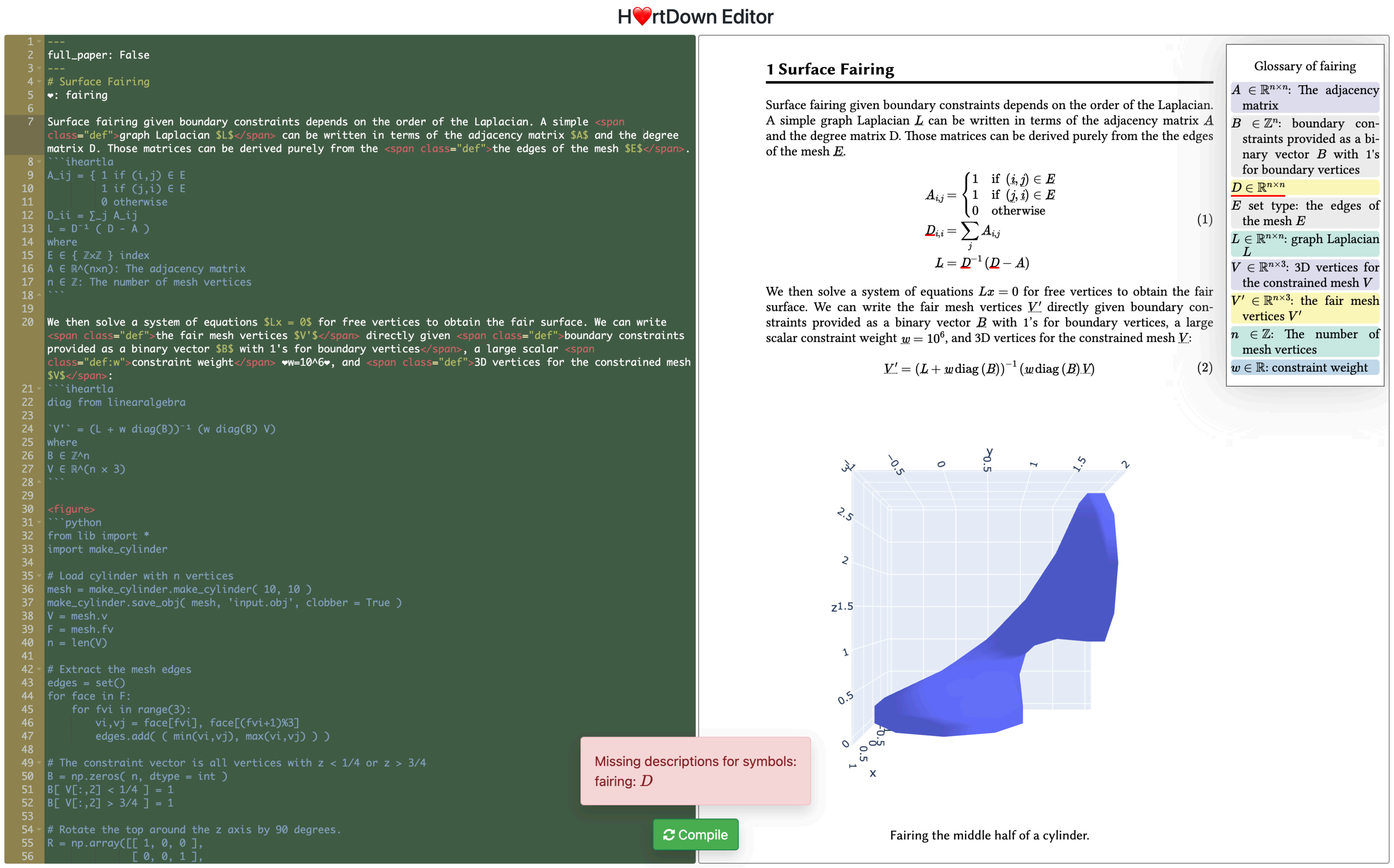
~~~
---
full_paper: False
---
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those matrices can be derived purely from the <span class="def">the edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤxℤ } index
A ∈ ℝ^(n×n): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
```

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write <span class="def">the fair mesh vertices $V'$</span> directly given <span class="def">boundary constraints provided as a binary vector $B$ with 1's for boundary vertices</span>, a large scalar <span class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh $V$</span>:
```iheartla
diag from linearalgebra

`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
where
B ∈ ℤ^n
V ∈ ℝ^(n × 3)
```

<figure>
```python
from lib import *
import make_cylinder

# Load cylinder with n vertices
mesh = make_cylinder.make_cylinder( 10, 10 )
make_cylinder.save_obj( mesh, 'input.obj', clobber = True )
V = mesh.v
F = mesh.fv
n = len(V)

# Extract the mesh edges
edges = set()
for face in F:
    for fvi in range(3):
        vi,vj = face[fvi], face[(fvi+1)%3]
        edges.add( ( min(vi,vj), max(vi,vj) ) )

# The constraint vector is all vertices with z < 1/4 or z > 3/4
B = np.zeros( n, dtype = int )
B[ V[:,2] < 1/4 ] = 1
B[ V[:,2] > 3/4 ] = 1

# Rotate the top around the z axis by 90 degrees.
R = np.array([[ 1, 0, 0 ],
~~~



~~~
---
full_paper: False
---
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span
class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the <span
class="def">degree matrix $D$</span>. Those matrices can be derived purely from the <span class="def">the
edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤxℤ } index
A ∈ ℝ^(nxn): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
```

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write
<span class="def">the fair mesh vertices $V'$</span> directly given <span class="def">boundary constraints
provided as a binary vector $B$ with 1's for boundary vertices</span>, a large scalar <span
class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh
$V$</span>:
```iheartla
diag from linearalgebra

`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
where
B ∈ ℤ^n
V ∈ ℝ^(n × 3)
```


<figure>
```python
from lib import *
import make_cylinder

# Load cylinder with n vertices
mesh = make_cylinder.make_cylinder( 10, 10 )
make_cylinder.save_obj( mesh, 'input.obj', clobber = True )
V = mesh.v
F = mesh.fv
n = len(V)

# Extract the mesh edges
edges = set()
for face in F:
    for fvi in range(3):
        vi,vj = face[fvi], face[(fvi+1)%3]
        edges.add( ( min(vi,vj), max(vi,vj) ) )

# The constraint vector is all vertices with z < 1/4 or z > 3/4
B = np.zeros( n, dtype = int )
B[ V[:,2] < 1/4 ] = 1
B[ V[:,2] > 3/4 ] = 1

# Rotate the top around the z axis by 90 degrees.
R = np.array([[ 1, 0, 0 ],
~~~

```
simple <span
ix $A$ and the <span
<span class="def">the




rface. We can write
">boundary constraints
r <span
or the constrained mesh
```

```
simple <span
ix $A$ and the <span
<span class="def">the




rface. We can write
'>boundary constraints
r <span
or the constrained mesh
```



H❤rtDown Editor

~~~
---
full_paper: False
---
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the degree matrix D. Those matrices can be derived purely from the <span class="def">the edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤxℤ } index
A ∈ ℝ^(nxn): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
~~~

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write <span class="def">the fair mesh vertices V'</span> directly given <span class="def">boundary constraints provided as a binary vector B with 1's for boundary vertices</span>, a large scalar <span class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh V</span>:
```iheartla
diag from linearalgebra

`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
where
B ∈ ℤ^n
V ∈ ℝ^(n x 3)
```

<figure>
```python
from lib import *
import make_cylinder

# Load cylinder with n vertices
mesh = make_cylinder.make_cylinder( 10, 10 )
make_cylinder.save_obj( mesh, 'input.obj', clobber = True )
V = mesh.v
F = mesh.fv
n = len(V)

# Extract the mesh edges
edges = set()
for face in F:
    for fvi in range(3):
        vi,vj = face[fvi], face[(fvi+1)%3]
        edges.add( ( min(vi,vj), max(vi,vj) ) )

# The constraint vector is all vertices with z < 1/4 or z > 3/4
B = np.zeros( n, dtype = int )
B[ V[:,2] < 1/4 ] = 1
B[ V[:,2] > 3/4 ] = 1

# Rotate the top around the z axis by 90 degrees.
R = np.array([[ 1, 0, 0 ],
              [ 0, 0, 1 ],
```

1 Surface Fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple graph Laplacian L can be written in terms of the adjacency matrix A and the degree matrix D. Those matrices can be derived purely from the the edges of the mesh E.

$$A_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 1 & \text{if } (j,i) \in E \\ 0 & \text{otherwise} \end{cases}$$
$$D_{i,i} = \sum_j A_{i,j}$$
$$L = D^{-1}(D - A) \qquad (1)$$

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write the fair mesh vertices V' directly given boundary constraints provided as a binary vector B with 1's for boundary vertices, a large scalar constraint weight $w = 10^6$, and 3D vertices for the constrained mesh V:

$$V' = (L + w\,\mathrm{diag}(B))^{-1}(w\,\mathrm{diag}(B)V) \qquad (2)$$

Glossary of fairing

$A \in \mathbb{R}^{n\times n}$: The adjacency matrix

$B \in \mathbb{Z}^{n}$: boundary constraints provided as a binary vector B with 1's for boundary vertices

$D \in \mathbb{R}^{n\times n}$

E set type: the edges of the mesh E

$L \in \mathbb{R}^{n\times n}$: graph Laplacian L

$V \in \mathbb{R}^{n\times 3}$: 3D vertices for the constrained mesh V

$V' \in \mathbb{R}^{n\times 3}$: the fair mesh vertices V'

$n \in \mathbb{Z}$: The number of mesh vertices

$w \in \mathbb{R}$: constraint weight

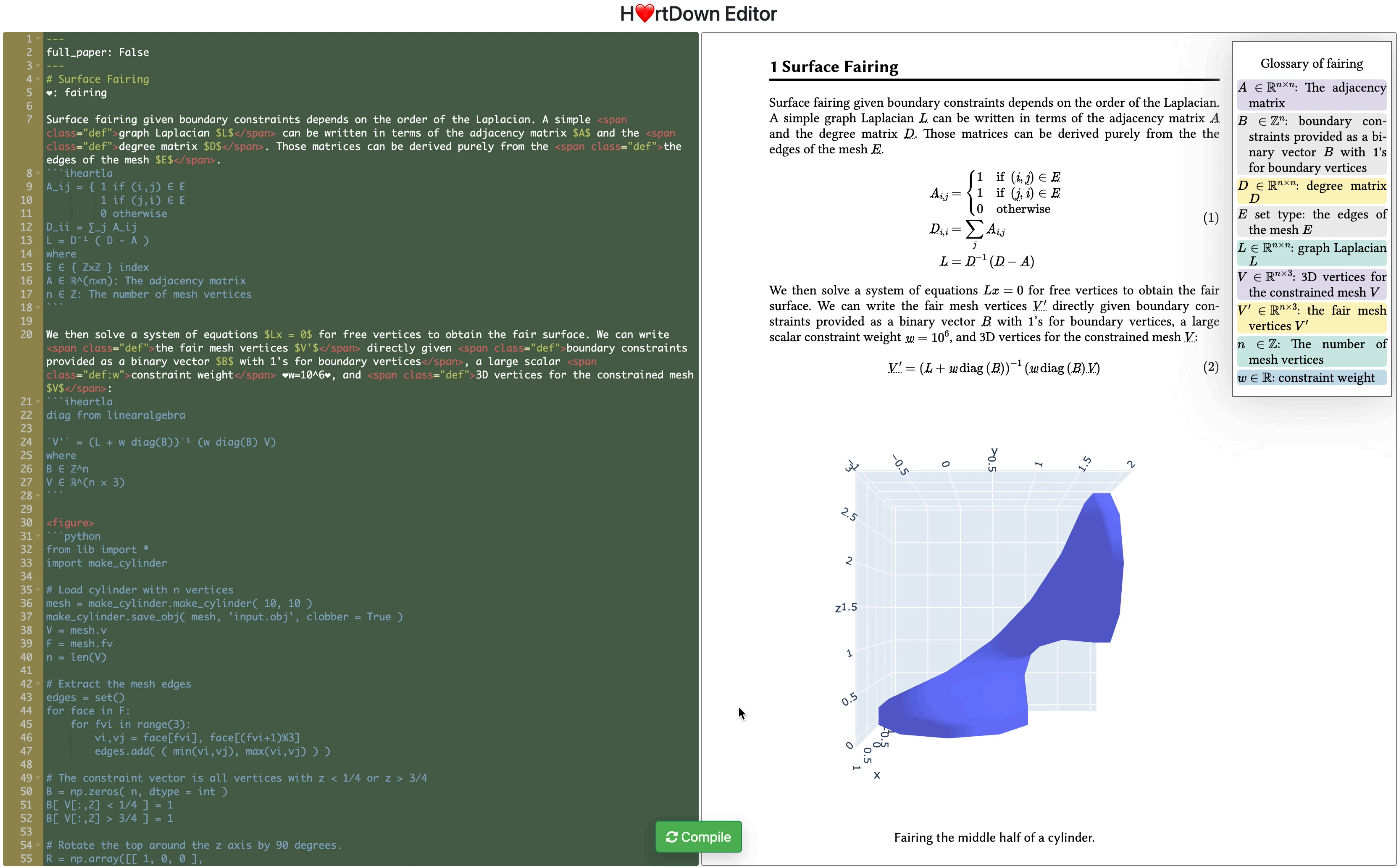
Fairing the middle half of a cylinder.

Missing descriptions for symbols: fairing: D

Missing descriptions for symbols:
fairing: D

Compile
z
1.5
1
0.5
0
x
-0.5
0
0.5
1

Fairing the m



~~~
---
full_paper: False
---
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those matrices can be derived purely from the <span class="def">the edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤxℤ } index
A ∈ ℝ^(nxn): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
```

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write <span class="def">the fair mesh vertices $V'$</span> directly given <span class="def">boundary constraints provided as a binary vector $B$ with 1's for boundary vertices</span>, a large scalar <span class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh $V$</span>:
```iheartla
diag from linearalgebra

`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
where
B ∈ ℤ^n
V ∈ ℝ^(n × 3)
```

<figure>
```python
from lib import *
import make_cylinder

# Load cylinder with n vertices
mesh = make_cylinder.make_cylinder( 10, 10 )
make_cylinder.save_obj( mesh, 'input.obj', clobber = True )
V = mesh.v
F = mesh.fv
n = len(V)

# Extract the mesh edges
edges = set()
for face in F:
    for fvi in range(3):
        vi,vj = face[fvi], face[(fvi+1)%3]
        edges.add( ( min(vi,vj), max(vi,vj) ) )

# The constraint vector is all vertices with z < 1/4 or z > 3/4
B = np.zeros( n, dtype = int )
B[ V[:,2] < 1/4 ] = 1
B[ V[:,2] > 3/4 ] = 1

# Rotate the top around the z axis by 90 degrees.
R = np.array([[ 1, 0, 0 ],
~~~

1 Surface Fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple graph Laplacian L can be written in terms of the adjacency matrix A and the degree matrix D. Those matrices can be derived purely from the the edges of the mesh E.

$$\begin{aligned} A_{i,j} &= \begin{cases} 1 & \text{if } (i,j) \in E \\ 1 & \text{if } (j,i) \in E \\ 0 & \text{otherwise} \end{cases} \\ D_{i,i} &= \sum_j A_{i,j} \\ L &= D^{-1}(D - A) \end{aligned} \tag{1}$$

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write the fair mesh vertices V' directly given boundary constraints provided as a binary vector B with 1's for boundary vertices, a large scalar constraint weight $w = 10^6$, and 3D vertices for the constrained mesh V:

$$V' = (L + w\,\mathrm{diag}(B))^{-1}(w\,\mathrm{diag}(B)V) \tag{2}$$

Glossary of fairing

$A \in \mathbb{R}^{n\times n}$: The adjacency matrix

$B \in \mathbb{Z}^{n}$: boundary constraints provided as a binary vector B with 1's for boundary vertices

$D \in \mathbb{R}^{n\times n}$: degree matrix D

E set type: the edges of the mesh E

$L \in \mathbb{R}^{n\times n}$: graph Laplacian L

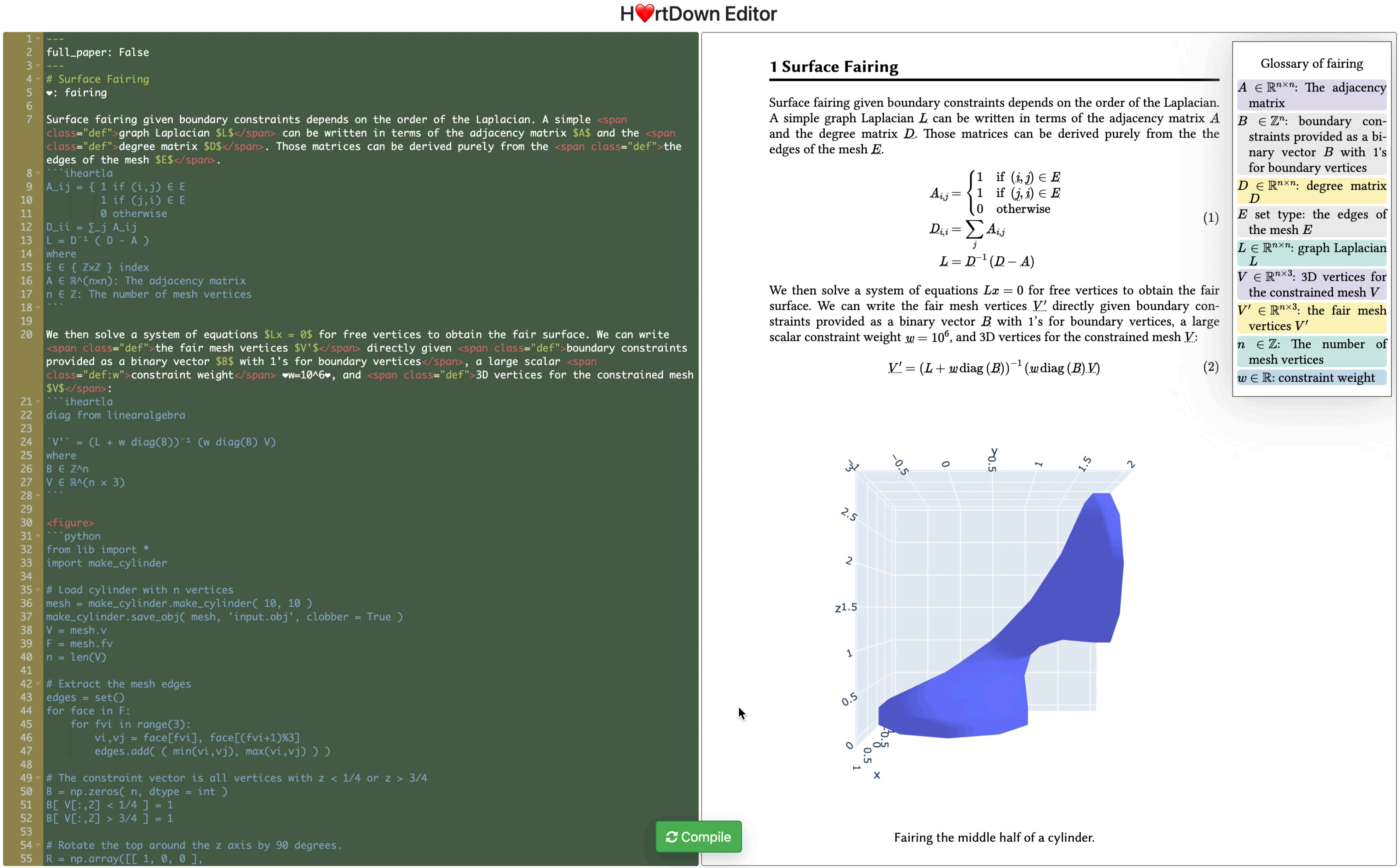
$V \in \mathbb{R}^{n\times 3}$: 3D vertices for the constrained mesh V

$V' \in \mathbb{R}^{n\times 3}$: the fair mesh vertices V'

$n \in \mathbb{Z}$: The number of mesh vertices

$w \in \mathbb{R}$: constraint weight

Fairing the middle half of a cylinder.



~~~
---
full_paper: False
---
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those matrices can be derived purely from the <span class="def">the edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤxℤ } index
A ∈ ℝ^(nxn): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
```

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write <span class="def">the fair mesh vertices $V'$</span> directly given <span class="def">boundary constraints provided as a binary vector $B$ with 1's for boundary vertices</span>, a large scalar <span class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh $V$</span>:
```iheartla
diag from linearalgebra

`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
where
B ∈ ℤ^n
V ∈ ℝ^(n × 3)
```


<figure>
```python
from lib import *
import make_cylinder

# Load cylinder with n vertices
mesh = make_cylinder.make_cylinder( 10, 10 )
make_cylinder.save_obj( mesh, 'input.obj', clobber = True )
V = mesh.v
F = mesh.fv
n = len(V)

# Extract the mesh edges
edges = set()
for face in F:
    for fvi in range(3):
        vi,vj = face[fvi], face[(fvi+1)%3]
        edges.add( ( min(vi,vj), max(vi,vj) ) )

# The constraint vector is all vertices with z < 1/4 or z > 3/4
B = np.zeros( n, dtype = int )
B[ V[:,2] < 1/4 ] = 1
B[ V[:,2] > 3/4 ] = 1

# Rotate the top around the z axis by 90 degrees.
R = np.array([[ 1, 0, 0 ],
~~~

1 Surface Fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple graph Laplacian L can be written in terms of the adjacency matrix A and the degree matrix D. Those matrices can be derived purely from the the edges of the mesh E.

$$\begin{aligned} A_{i,j} &= \begin{cases} 1 & \text{if } (i,j) \in E \\ 1 & \text{if } (j,i) \in E \\ 0 & \text{otherwise} \end{cases} \\ D_{i,i} &= \sum_j A_{i,j} \\ L &= D^{-1}(D - A) \end{aligned} \tag{1}$$

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write the fair mesh vertices V' directly given boundary constraints provided as a binary vector B with 1's for boundary vertices, a large scalar constraint weight $w = 10^6$, and 3D vertices for the constrained mesh V:

$$V' = (L + w\,\mathrm{diag}\,(B))^{-1}\,(w\,\mathrm{diag}\,(B)\,V) \tag{2}$$

Fairing the middle half of a cylinder.

Glossary of fairing

$A \in \mathbb{R}^{n\times n}$: The adjacency matrix

$B \in \mathbb{Z}^{n}$: boundary constraints provided as a binary vector B with 1's for boundary vertices

$D \in \mathbb{R}^{n\times n}$: degree matrix D

E set type: the edges of the mesh E

$L \in \mathbb{R}^{n\times n}$: graph Laplacian L

$V \in \mathbb{R}^{n\times 3}$: 3D vertices for the constrained mesh V

$V' \in \mathbb{R}^{n\times 3}$: the fair mesh vertices V'

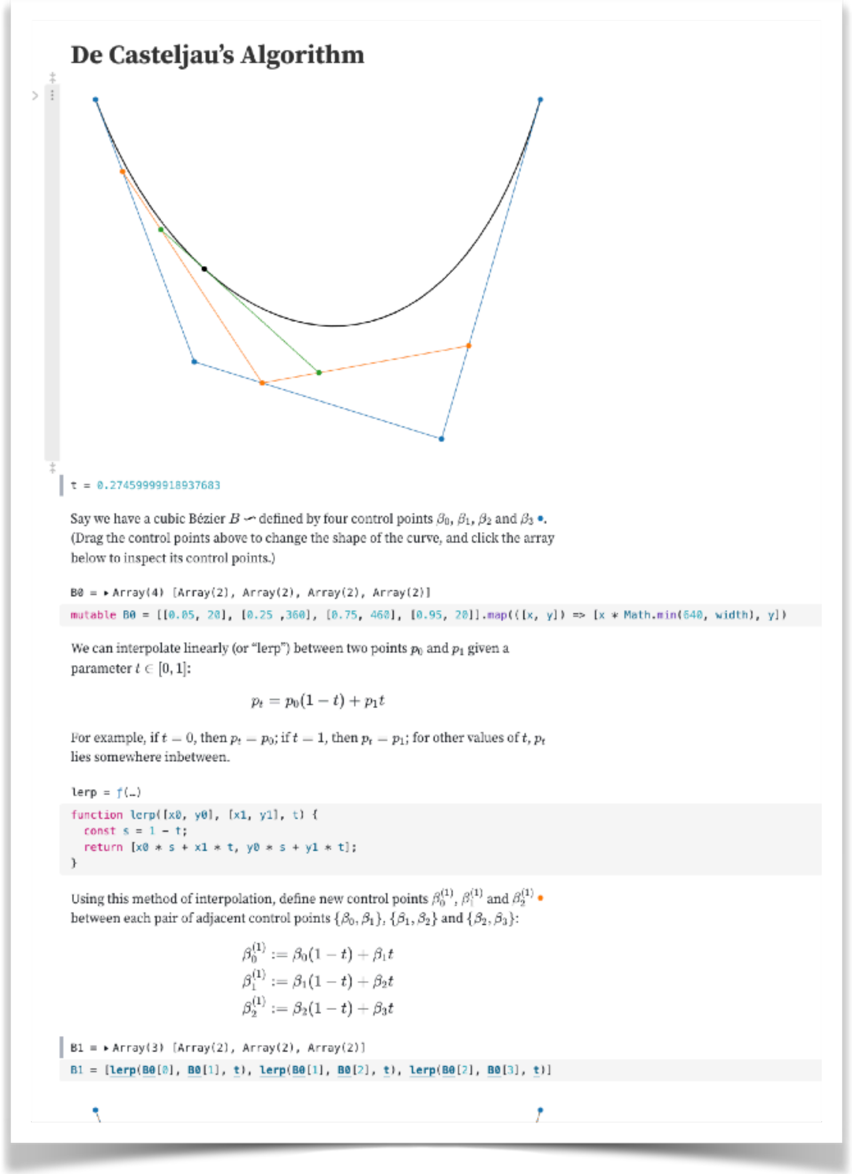
$n \in \mathbb{Z}$: The number of mesh vertices

$w \in \mathbb{R}$: constraint weight

Related Work

Literate programming environments

- Literate Programming [Knuth 1984]
- Markdown [Gruber and Swartz 2004]
- Notebooks [Arnon 1988; Kery et al. 2018; Rule et al. 2018; Wolfram 1988]
- Pluto [Plas 2020]
- Observable [Bostock 2017]



De Casteljau's Algorithm

`t = 0.27459999918937683`

Say we have a cubic Bézier B ⤙ defined by four control points $\beta_0, \beta_1, \beta_2$ and β_3 •. (Drag the control points above to change the shape of the curve, and click the array below to inspect its control points.)

```
B0 = ▸ Array(4) [Array(2), Array(2), Array(2), Array(2)]
mutable B0 = [[0.05, 20], [0.25 ,360], [0.75, 460], [0.95, 20]].map(([x, y]) => [x * Math.min(640, width), y])
```

We can interpolate linearly (or "lerp") between two points p_0 and p_1 given a parameter $t \in [0, 1]$:

$$p_t = p_0(1 - t) + p_1 t$$

For example, if $t = 0$, then $p_t = p_0$; if $t = 1$, then $p_t = p_1$; for other values of t, p_t lies somewhere inbetween.

```
lerp = f(…)
function lerp([x0, y0], [x1, y1], t) {
  const s = 1 - t;
  return [x0 * s + x1 * t, y0 * s + y1 * t];
}
```

Using this method of interpolation, define new control points $\beta_0^{(1)}, \beta_1^{(1)}$ and $\beta_2^{(1)}$ • between each pair of adjacent control points $\{\beta_0, \beta_1\}$, $\{\beta_1, \beta_2\}$ and $\{\beta_2, \beta_3\}$:

$$\begin{aligned}\beta_0^{(1)} &:= \beta_0(1 - t) + \beta_1 t\\ \beta_1^{(1)} &:= \beta_1(1 - t) + \beta_2 t\\ \beta_2^{(1)} &:= \beta_2(1 - t) + \beta_3 t\end{aligned}$$

```
B1 = ▸ Array(3) [Array(2), Array(2), Array(2)]
B1 = [lerp(B0[0], B0[1], t), lerp(B0[1], B0[2], t), lerp(B0[2], B0[3], t)]
```

Reactive documents and publishing

- Idyll [Conlen and Heer 2018]
- Tangle [Victor 2011]
- Distill [Team 2021]
- Authorea [Goodman et al. 2017]
- Nota [Crichton 2021]
- [Bonneel et al. 2020]
- ScholarPhi [Head et al. 2021]

Augmenting Scientific Papers with Just-in-Time, Position-Sensitive Definitions of Terms and Symbols

Andrew Head
andrewhead@berkeley.edu
UC Berkeley

Kyle Lo
kylel@allenai.org
Allen Institute for AI

Dongyeop Kang
dongyeopk@berkeley.edu
UC Berkeley

Raymond Fok
rayfok@cs.washington.edu
University of Washington

Sam Skjonsberg
sams@allenai.org
Allen Institute for AI

Daniel S. Weld
danw@allenai.org
Allen Institute for AI
University of Washington

Marti A. Hearst
hearst@berkeley.edu
UC Berkeley

ABSTRACT

Despite the central importance of research papers to scientific progress, they can be difficult to read. Comprehension is often stymied when the information needed to understand a passage resides somewhere else—in another section, or in another paper. In this work, we envision how interfaces can bring definitions of technical terms and symbols to readers when and where they need them most. We introduce *ScholarPhi*, an augmented reading interface with four novel features: (1) tooltips that surface position-sensitive definitions from elsewhere in a paper, (2) a filter over the paper that "declutters" it to reveal how the term or symbol is used across the paper, (3) automatic equation diagrams that expose multiple definitions in parallel, and (4) an automatically generated glossary of important terms and symbols. A usability study showed that the tool helps researchers of all experience levels read papers. Furthermore, researchers were eager to have ScholarPhi's definitions available to support their everyday reading.

CCS CONCEPTS

• **Human-centered computing → Interactive systems and tools.**

KEYWORDS

interactive documents, reading interfaces,scientific papers, definitions, nonce words

ACM Reference Format:
Andrew Head, Kyle Lo, Dongyeop Kang, Raymond Fok, Sam Skjonsberg, Daniel S. Weld, and Marti A. Hearst. 2021. Augmenting Scientific Papers with Just-in-Time, Position-Sensitive Definitions of Terms and Symbols. In *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3411764.3445648

1 INTRODUCTION

Researchers are charged with keeping on top of immense, rapidly-changing literatures. Naturally, then, reading constitutes a major part of a researcher's everyday work. Senior researchers, such as faculty members, spend over one hundred hours a year reading the literature, consuming over one hundred papers annually [97]. And despite the formidable background knowledge that a researcher gains over the course of their career, they will still often find that papers are prohibitively difficult to read.

As they read, a researcher is constantly trying to fit the information they find into schemas of their prior knowledge, but the success of this assimilation is by no means guaranteed [7]. A researcher may struggle to understand a paper due to gaps in their own knowledge, or due to the intrinsic difficulty of reading a specific paper [7]. Reading is made all the more challenging by the fact that scholars increasingly read selectively, looking for specific information by skimming and scanning [34, 70, 98].

We are motivated by the question: Can a novel interface improve the reading experience by reducing distractions that interrupt the reading flow? This work takes a measured step to address the general design question by focusing on the specific case of helping readers understand cryptic technical terms and symbols defined

nonce word (the symbol "k")
buttons to open definition, formula, and usage lists
work is trained to maximize the expected log likelihood
number of components (page 2).
See 41 usages
definition
hyperlink to definition in context
usage count

Figure 1: ScholarPhi helps readers understand *nonce words*—unique technical terms and symbols—defined within scientific papers. When a reader comes across a nonce word that they do not understand, ScholarPhi lets them click the word to view a position-sensitive definition in a compact tooltip. The tooltip lets the reader jump to the definition in context. It also lets them open lists of prose definitions, defining formulae, and usages of the word. ScholarPhi augments the reading experience with this and a host of other features (see Section 4) to assist readers.

CHI '21, May 8–13, 2021, Yokohama, Japan
© 2021 Copyright held by the owner/author(s).
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *CHI Conference on Human Factors in Computing Systems (CHI '21), May 8–13, 2021, Yokohama, Japan*, https://doi.org/10.1145/3411764.3445648.

Compilable math and augmentations

- Fortress [Allen et al. 2005]
- Lean [de Moura et al. 2015]
- Julia [Bezanson et al. 2017]
- [Alcock and Wilkinson 2011]
- [Dragunov and Herlocker 2003]
- [Head et al. 2021, 2022]
- Penrose [Ye et al. 2020]
- I❤️LA [Li et al. 2021]

```
given
p_i ∈ ℝ³: points on lines
d_i ∈ ℝ³: unit directions along lines

P_i = ( I₃ - d_i d_iᵀ )
q = ( ∑_i P_i )⁻¹ ( ∑_i P_i p_i )
```

Design Goals

Design Goals

- Support **authoring**, **reading**, and making use of (**experimenting** with)

Design Goals

- Support **authoring**, **reading**, and making use of (**experimenting** with)
  - Correct and reproducible documents

Design Goals

- Support **authoring**, **reading**, and making use of (**experimenting** with)
  - Correct and reproducible documents
  - Minimal authoring overhead

Design Goals

- Support **authoring**, **reading**, and making use of (**experimenting** with)
  - Correct and reproducible documents
  - Minimal authoring overhead
- **Ecological compatibility**

Design Goals

- Support **authoring**, **reading**, and making use of (**experimenting** with)
  - Correct and reproducible documents
  - Minimal authoring overhead
- **Ecological compatibility**
  - Don’t change **what** authors put in papers (prose, math, figures, tables)

Design Goals

- Support **authoring**, **reading**, and making use of (**experimenting** with)
  - Correct and reproducible documents
  - Minimal authoring overhead
- **Ecological compatibility**
  - Don’t change **what** authors put in papers (prose, math, figures, tables)
  - Minimal changes to **how** they write
    - Plain text documents

156 SIGGRAPH 2020 papers

Formative Study

- All appear to be written using LaTeX.

Formative Study

- All appear to be written using LaTeX.
- Observations:

Formative Study

- All appear to be written using LaTeX.
- Observations:
    I. Prose organizes the document, interleaved with math.

Formative Study

- All appear to be written using LaTeX.
- Observations:
  I. Prose organizes the document, interleaved with math.
  II. Math appears out of order. Symbols used before defined.

Math appears out of order [Wronski et al. 2019]

Math appears out of order [Wronski et al. 2019]

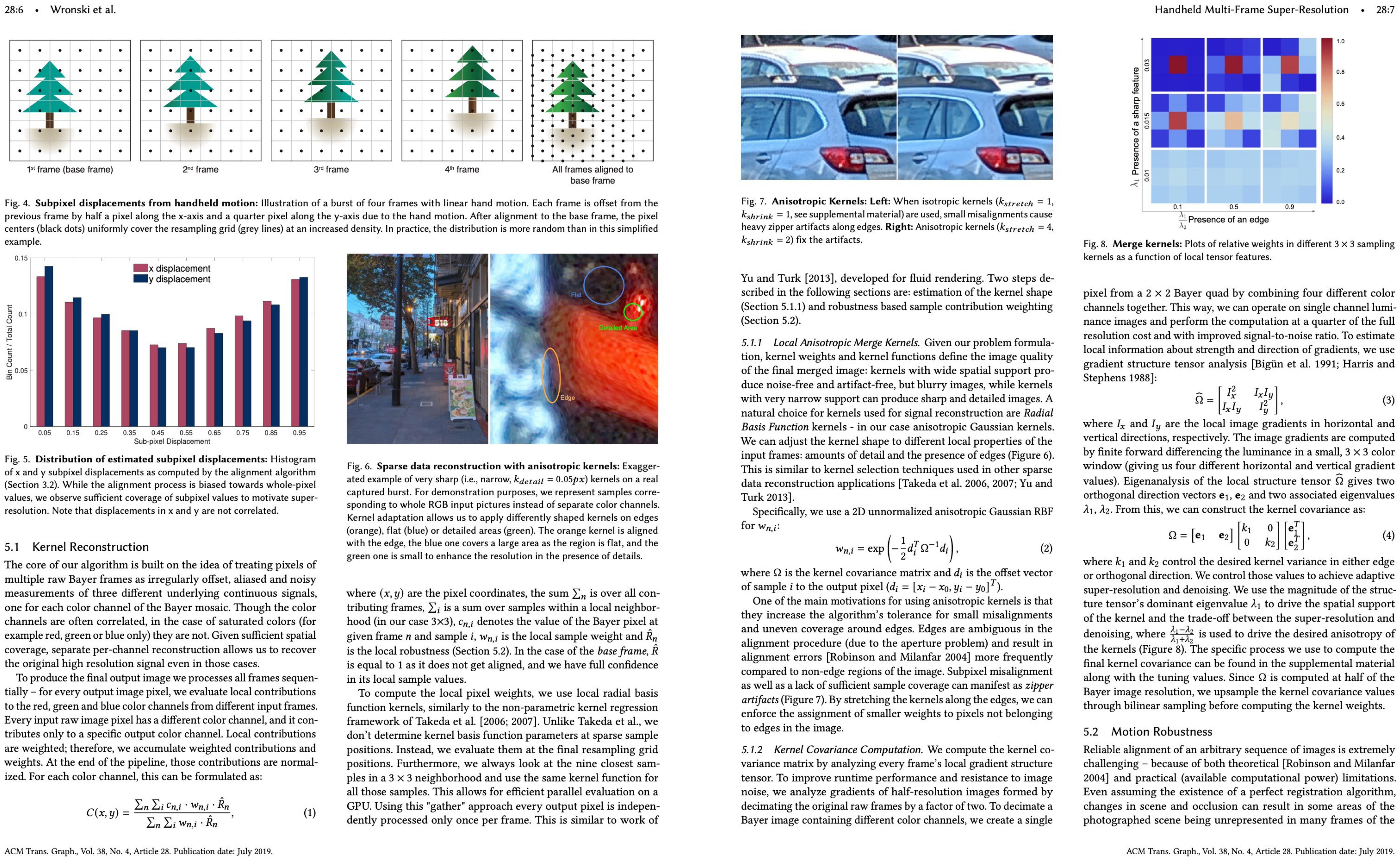


Fig. 4. **Subpixel displacements from handheld motion:** Illustration of a burst of four frames with linear hand motion. Each frame is offset from the previous frame by half a pixel along the x-axis and a quarter pixel along the y-axis due to the hand motion. After alignment to the base frame, the pixel centers (black dots) uniformly cover the resampling grid (grey lines) at an increased density. In practice, the distribution is more random than in this simplified example.

Fig. 5. **Distribution of estimated subpixel displacements:** Histogram of x and y subpixel displacements as computed by the alignment algorithm (Section 3.2). While the alignment process is biased towards whole-pixel values, we observe sufficient coverage of subpixel values to motivate super-resolution. Note that displacements in x and y are not correlated.

Fig. 6. **Sparse data reconstruction with anisotropic kernels:** Exaggerated example of very sharp (i.e., narrow, $k_{detail} = 0.05px$) kernels on a real captured burst. For demonstration purposes, we represent samples corresponding to whole RGB input pictures instead of separate color channels. Kernel adaptation allows us to apply differently shaped kernels on edges (orange), flat (blue) or detailed areas (green). The orange kernel is aligned with the edge, the blue one covers a large area as the region is flat, and the green one is small to enhance the resolution in the presence of details.

5.1 Kernel Reconstruction

The core of our algorithm is built on the idea of treating pixels of multiple raw Bayer frames as irregularly offset, aliased and noisy measurements of three different underlying continuous signals, one for each color channel of the Bayer mosaic. Though the color channels are often correlated, in the case of saturated colors (for example red, green or blue only) they are not. Given sufficient spatial coverage, separate per-channel reconstruction allows us to recover the original high resolution signal even in those cases.

To produce the final output image we processes all frames sequentially – for every output image pixel, we evaluate local contributions to the red, green and blue color channels from different input frames. Every input raw image pixel has a different color channel, and it contributes only to a specific output color channel. Local contributions are weighted; therefore, we accumulate weighted contributions and weights. At the end of the pipeline, those contributions are normalized. For each color channel, this can be formulated as:

$$C(x,y) = \frac{\sum_n \sum_i c_{n,i} \cdot w_{n,i} \cdot \hat{R}_n}{\sum_n \sum_i w_{n,i} \cdot \hat{R}_n}, \tag{1}$$

where (x, y) are the pixel coordinates, the sum $\sum_n$ is over all contributing frames, $\sum_i$ is a sum over samples within a local neighborhood (in our case 3×3), $c_{n,i}$ denotes the value of the Bayer pixel at given frame n and sample i, $w_{n,i}$ is the local sample weight and $\hat{R}_n$ is the local robustness (Section 5.2). In the case of the *base frame*, $\hat{R}$ is equal to 1 as it does not get aligned, and we have full confidence in its local sample values.

To compute the local pixel weights, we use local radial basis function kernels, similarly to the non-parametric kernel regression framework of Takeda et al. [2006; 2007]. Unlike Takeda et al., we don't determine kernel basis function parameters at sparse sample positions. Instead, we evaluate them at the final resampling grid positions. Furthermore, we always look at the nine closest samples in a 3×3 neighborhood and use the same kernel function for all those samples. This allows for efficient parallel evaluation on a GPU. Using this "gather" approach every output pixel is independently processed only once per frame. This is similar to work of

Fig. 7. **Anisotropic Kernels: Left:** When isotropic kernels ($k_{stretch} = 1$, $k_{shrink} = 1$, see supplemental material) are used, small misalignments cause heavy zipper artifacts along edges. **Right:** Anisotropic kernels ($k_{stretch} = 4$, $k_{shrink} = 2$) fix the artifacts.

Yu and Turk [2013], developed for fluid rendering. Two steps described in the following sections are: estimation of the kernel shape (Section 5.1.1) and robustness based sample contribution weighting (Section 5.2).

5.1.1 Local Anisotropic Merge Kernels. Given our problem formulation, kernel weights and kernel functions define the image quality of the final merged image: kernels with wide spatial support produce noise-free and artifact-free, but blurry images, while kernels with very narrow support can produce sharp and detailed images. A natural choice for kernels used for signal reconstruction are *Radial Basis Function* kernels - in our case anisotropic Gaussian kernels. We can adjust the kernel shape to different local properties of the input frames: amounts of detail and the presence of edges (Figure 6). This is similar to kernel selection techniques used in other sparse data reconstruction applications [Takeda et al. 2006, 2007; Yu and Turk 2013].

Specifically, we use a 2D unnormalized anisotropic Gaussian RBF for $w_{n,i}$:

$$w_{n,i} = \exp\left(-\frac{1}{2} d_i^T \Omega^{-1} d_i\right), \tag{2}$$

where Ω is the kernel covariance matrix and d_i is the offset vector of sample i to the output pixel ($d_i = [x_i - x_0, y_i - y_0]^T$).

One of the main motivations for using anisotropic kernels is that they increase the algorithm's tolerance for small misalignments and uneven coverage around edges. Edges are ambiguous in the alignment procedure (due to the aperture problem) and result in alignment errors [Robinson and Milanfar 2004] more frequently compared to non-edge regions of the image. Subpixel misalignment as well as a lack of sufficient sample coverage can manifest as *zipper artifacts* (Figure 7). By stretching the kernels along the edges, we can enforce the assignment of smaller weights to pixels not belonging to edges in the image.

5.1.2 Kernel Covariance Computation. We compute the kernel covariance matrix by analyzing every frame's local gradient structure tensor. To improve runtime performance and resistance to image noise, we analyze gradients of half-resolution images formed by decimating the original raw frames by a factor of two. To decimate a Bayer image containing different color channels, we create a single

1.0
0.8
0.6
0.4
0.2
0.0
0.03
0.015
0.01
λ_1 Presence of a sharp feature
0.1
0.5
0.9
$\frac{\lambda_1}{\lambda_2}$ Presence of an edge

Fig. 8. **Merge kernels:** Plots of relative weights in different 3×3 sampling kernels as a function of local tensor features.

pixel from a 2×2 Bayer quad by combining four different color channels together. This way, we can operate on single channel luminance images and perform the computation at a quarter of the full resolution cost and with improved signal-to-noise ratio. To estimate local information about strength and direction of gradients, we use gradient structure tensor analysis [Bigün et al. 1991; Harris and Stephens 1988]:

$$\widehat{\Omega} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \tag{3}$$

where I_x and I_y are the local image gradients in horizontal and vertical directions, respectively. The image gradients are computed by finite forward differencing the luminance in a small, 3×3 color window (giving us four different horizontal and vertical gradient values). Eigenanalysis of the local structure tensor $\widehat{\Omega}$ gives two orthogonal direction vectors $\mathbf{e}_1, \mathbf{e}_2$ and two associated eigenvalues λ_1, λ_2. From this, we can construct the kernel covariance as:

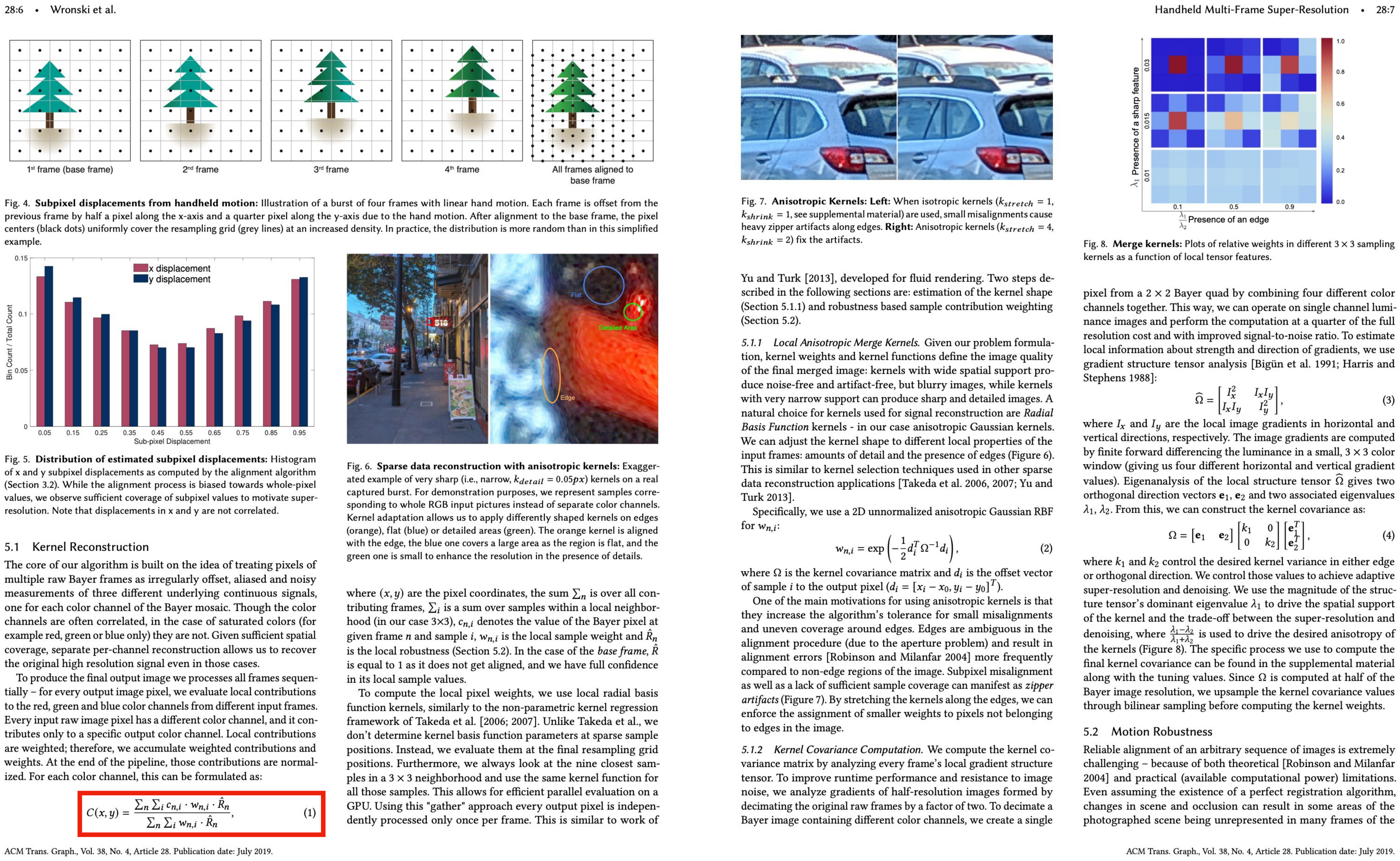
$$\Omega = \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 \end{bmatrix} \begin{bmatrix} k_1 & 0 \\ 0 & k_2 \end{bmatrix} \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \end{bmatrix}, \tag{4}$$

where k_1 and k_2 control the desired kernel variance in either edge or orthogonal direction. We control those values to achieve adaptive super-resolution and denoising. We use the magnitude of the structure tensor's dominant eigenvalue λ_1 to drive the spatial support of the kernel and the trade-off between the super-resolution and denoising, where $\frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2}$ is used to drive the desired anisotropy of the kernels (Figure 8). The specific process we use to compute the final kernel covariance can be found in the supplemental material along with the tuning values. Since Ω is computed at half of the Bayer image resolution, we upsample the kernel covariance values through bilinear sampling before computing the kernel weights.

5.2 Motion Robustness

Reliable alignment of an arbitrary sequence of images is extremely challenging – because of both theoretical [Robinson and Milanfar 2004] and practical (available computational power) limitations. Even assuming the existence of a perfect registration algorithm, changes in scene and occlusion can result in some areas of the photographed scene being unrepresented in many frames of the

Math appears out of order [Wronski et al. 2019]



1st frame (base frame)
2nd frame
3rd frame
4th frame
All frames aligned to base frame

Fig. 4. **Subpixel displacements from handheld motion:** Illustration of a burst of four frames with linear hand motion. Each frame is offset from the previous frame by half a pixel along the x-axis and a quarter pixel along the y-axis due to the hand motion. After alignment to the base frame, the pixel centers (black dots) uniformly cover the resampling grid (grey lines) at an increased density. In practice, the distribution is more random than in this simplified example.

Fig. 5. **Distribution of estimated subpixel displacements:** Histogram of x and y subpixel displacements as computed by the alignment algorithm (Section 3.2). While the alignment process is biased towards whole-pixel values, we observe sufficient coverage of subpixel values to motivate super-resolution. Note that displacements in x and y are not correlated.

Fig. 6. **Sparse data reconstruction with anisotropic kernels:** Exaggerated example of very sharp (i.e., narrow, $k_{detail} = 0.05px$) kernels on a real captured burst. For demonstration purposes, we represent samples corresponding to whole RGB input pictures instead of separate color channels. Kernel adaptation allows us to apply differently shaped kernels on edges (orange), flat (blue) or detailed areas (green). The orange kernel is aligned with the edge, the blue one covers a large area as the region is flat, and the green one is small to enhance the resolution in the presence of details.

5.1 Kernel Reconstruction

The core of our algorithm is built on the idea of treating pixels of multiple raw Bayer frames as irregularly offset, aliased and noisy measurements of three different underlying continuous signals, one for each color channel of the Bayer mosaic. Though the color channels are often correlated, in the case of saturated colors (for example red, green or blue only) they are not. Given sufficient spatial coverage, separate per-channel reconstruction allows us to recover the original high resolution signal even in those cases.

To produce the final output image we processes all frames sequentially – for every output image pixel, we evaluate local contributions to the red, green and blue color channels from different input frames. Every input raw image pixel has a different color channel, and it contributes only to a specific output color channel. Local contributions are weighted; therefore, we accumulate weighted contributions and weights. At the end of the pipeline, those contributions are normalized. For each color channel, this can be formulated as:

$$\boxed{C(x,y) = \frac{\sum_n \sum_i c_{n,i} \cdot w_{n,i} \cdot \hat{R}_n}{\sum_n \sum_i w_{n,i} \cdot \hat{R}_n}, \qquad (1)}$$

where (x, y) are the pixel coordinates, the sum $\sum_n$ is over all contributing frames, $\sum_i$ is a sum over samples within a local neighborhood (in our case 3×3), $c_{n,i}$ denotes the value of the Bayer pixel at given frame n and sample i, $w_{n,i}$ is the local sample weight and $\hat{R}_n$ is the local robustness (Section 5.2). In the case of the *base frame*, $\hat{R}$ is equal to 1 as it does not get aligned, and we have full confidence in its local sample values.

To compute the local pixel weights, we use local radial basis function kernels, similarly to the non-parametric kernel regression framework of Takeda et al. [2006; 2007]. Unlike Takeda et al., we don't determine kernel basis function parameters at sparse sample positions. Instead, we evaluate them at the final resampling grid positions. Furthermore, we always look at the nine closest samples in a 3 × 3 neighborhood and use the same kernel function for all those samples. This allows for efficient parallel evaluation on a GPU. Using this "gather" approach every output pixel is independently processed only once per frame. This is similar to work of

Fig. 7. **Anisotropic Kernels: Left:** When isotropic kernels ($k_{stretch} = 1$, $k_{shrink} = 1$, see supplemental material) are used, small misalignments cause heavy zipper artifacts along edges. **Right:** Anisotropic kernels ($k_{stretch} = 4$, $k_{shrink} = 2$) fix the artifacts.

λ_1 Presence of a sharp feature
0.03
0.015
0.01
$\frac{\lambda_1}{\lambda_2}$ Presence of an edge
0.1
0.5
0.9
1.0
0.8
0.6
0.4
0.2
0.0

Fig. 8. **Merge kernels:** Plots of relative weights in different 3 × 3 sampling kernels as a function of local tensor features.

Yu and Turk [2013], developed for fluid rendering. Two steps described in the following sections are: estimation of the kernel shape (Section 5.1.1) and robustness based sample contribution weighting (Section 5.2).

5.1.1 Local Anisotropic Merge Kernels. Given our problem formulation, kernel weights and kernel functions define the image quality of the final merged image: kernels with wide spatial support produce noise-free and artifact-free, but blurry images, while kernels with very narrow support can produce sharp and detailed images. A natural choice for kernels used for signal reconstruction are *Radial Basis Function* kernels - in our case anisotropic Gaussian kernels. We can adjust the kernel shape to different local properties of the input frames: amounts of detail and the presence of edges (Figure 6). This is similar to kernel selection techniques used in other sparse data reconstruction applications [Takeda et al. 2006, 2007; Yu and Turk 2013].

Specifically, we use a 2D unnormalized anisotropic Gaussian RBF for $w_{n,i}$:

$$w_{n,i} = \exp\left(-\frac{1}{2} d_i^T \Omega^{-1} d_i\right), \qquad (2)$$

where Ω is the kernel covariance matrix and d_i is the offset vector of sample i to the output pixel ($d_i = [x_i - x_0, y_i - y_0]^T$).

One of the main motivations for using anisotropic kernels is that they increase the algorithm's tolerance for small misalignments and uneven coverage around edges. Edges are ambiguous in the alignment procedure (due to the aperture problem) and result in alignment errors [Robinson and Milanfar 2004] more frequently compared to non-edge regions of the image. Subpixel misalignment as well as a lack of sufficient sample coverage can manifest as *zipper artifacts* (Figure 7). By stretching the kernels along the edges, we can enforce the assignment of smaller weights to pixels not belonging to edges in the image.

5.1.2 Kernel Covariance Computation. We compute the kernel covariance matrix by analyzing every frame's local gradient structure tensor. To improve runtime performance and resistance to image noise, we analyze gradients of half-resolution images formed by decimating the original raw frames by a factor of two. To decimate a Bayer image containing different color channels, we create a single pixel from a 2 × 2 Bayer quad by combining four different color channels together. This way, we can operate on single channel luminance images and perform the computation at a quarter of the full resolution cost and with improved signal-to-noise ratio. To estimate local information about strength and direction of gradients, we use gradient structure tensor analysis [Bigün et al. 1991; Harris and Stephens 1988]:

$$\widehat{\Omega} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \qquad (3)$$

where I_x and I_y are the local image gradients in horizontal and vertical directions, respectively. The image gradients are computed by finite forward differencing the luminance in a small, 3 × 3 color window (giving us four different horizontal and vertical gradient values). Eigenanalysis of the local structure tensor $\widehat{\Omega}$ gives two orthogonal direction vectors $\mathbf{e}_1$, $\mathbf{e}_2$ and two associated eigenvalues λ_1, λ_2. From this, we can construct the kernel covariance as:

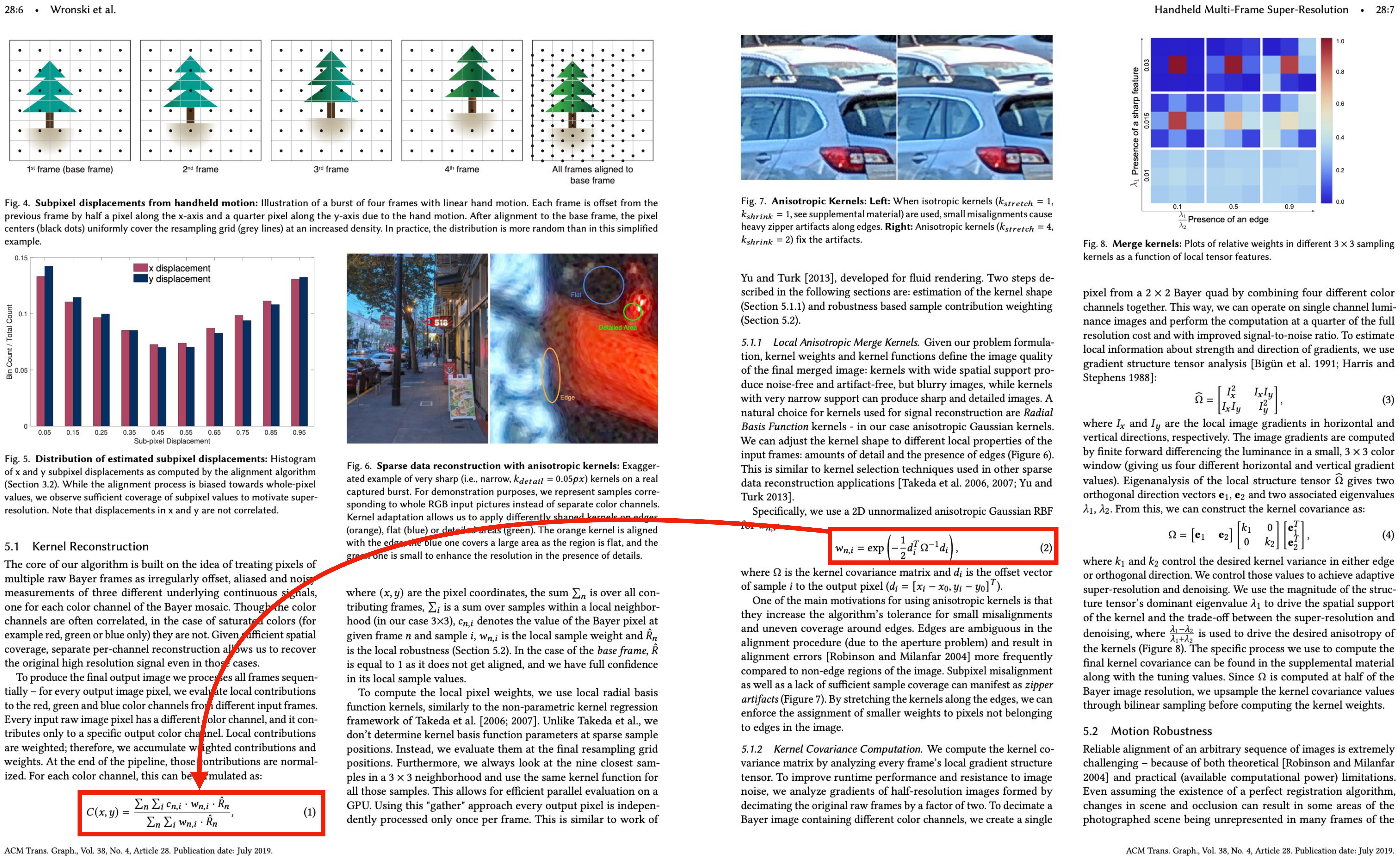
$$\Omega = \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 \end{bmatrix} \begin{bmatrix} k_1 & 0 \\ 0 & k_2 \end{bmatrix} \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \end{bmatrix}, \qquad (4)$$

where k_1 and k_2 control the desired kernel variance in either edge or orthogonal direction. We control those values to achieve adaptive super-resolution and denoising. We use the magnitude of the structure tensor's dominant eigenvalue λ_1 to drive the spatial support of the kernel and the trade-off between the super-resolution and denoising, where $\frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2}$ is used to drive the desired anisotropy of the kernels (Figure 8). The specific process we use to compute the final kernel covariance can be found in the supplemental material along with the tuning values. Since Ω is computed at half of the Bayer image resolution, we upsample the kernel covariance values through bilinear sampling before computing the kernel weights.

5.2 Motion Robustness

Reliable alignment of an arbitrary sequence of images is extremely challenging – because of both theoretical [Robinson and Milanfar 2004] and practical (available computational power) limitations. Even assuming the existence of a perfect registration algorithm, changes in scene and occlusion can result in some areas of the photographed scene being unrepresented in many frames of the

Math appears out of order [Wronski et al. 2019]



28:6 • Wronski et al.

Handheld Multi-Frame Super-Resolution • 28:7

Fig. 4. **Subpixel displacements from handheld motion:** Illustration of a burst of four frames with linear hand motion. Each frame is offset from the previous frame by half a pixel along the x-axis and a quarter pixel along the y-axis due to the hand motion. After alignment to the base frame, the pixel centers (black dots) uniformly cover the resampling grid (grey lines) at an increased density. In practice, the distribution is more random than in this simplified example.

Fig. 5. **Distribution of estimated subpixel displacements:** Histogram of x and y subpixel displacements as computed by the alignment algorithm (Section 3.2). While the alignment process is biased towards whole-pixel values, we observe sufficient coverage of subpixel values to motivate super-resolution. Note that displacements in x and y are not correlated.

Fig. 6. **Sparse data reconstruction with anisotropic kernels:** Exaggerated example of very sharp (i.e., narrow, $k_{detail} = 0.05px$) kernels on a real captured burst. For demonstration purposes, we represent samples corresponding to whole RGB input pictures instead of separate color channels. Kernel adaptation allows us to apply differently shaped kernels on edges (orange), flat (blue) or detailed areas (green). The orange kernel is aligned with the edge, the blue one covers a large area as the region is flat, and the green one is small to enhance the resolution in the presence of details.

5.1 Kernel Reconstruction

The core of our algorithm is built on the idea of treating pixels of multiple raw Bayer frames as irregularly offset, aliased and noisy measurements of three different underlying continuous signals, one for each color channel of the Bayer mosaic. Though the color channels are often correlated, in the case of saturated colors (for example red, green or blue only) they are not. Given sufficient spatial coverage, separate per-channel reconstruction allows us to recover the original high resolution signal even in those cases.

To produce the final output image we processes all frames sequentially – for every output image pixel, we evaluate local contributions to the red, green and blue color channels from different input frames. Every input raw image pixel has a different color channel, and it contributes only to a specific output color channel. Local contributions are weighted; therefore, we accumulate weighted contributions and weights. At the end of the pipeline, those contributions are normalized. For each color channel, this can be formulated as:

$$C(x,y) = \frac{\sum_n \sum_i c_{n,i} \cdot w_{n,i} \cdot \hat{R}_n}{\sum_n \sum_i w_{n,i} \cdot \hat{R}_n}, \tag{1}$$

where (x, y) are the pixel coordinates, the sum $\sum_n$ is over all contributing frames, $\sum_i$ is a sum over samples within a local neighborhood (in our case 3×3), $c_{n,i}$ denotes the value of the Bayer pixel at given frame n and sample i, $w_{n,i}$ is the local sample weight and $\hat{R}_n$ is the local robustness (Section 5.2). In the case of the *base frame*, $\hat{R}$ is equal to 1 as it does not get aligned, and we have full confidence in its local sample values.

To compute the local pixel weights, we use local radial basis function kernels, similarly to the non-parametric kernel regression framework of Takeda et al. [2006; 2007]. Unlike Takeda et al., we don't determine kernel basis function parameters at sparse sample positions. Instead, we evaluate them at the final resampling grid positions. Furthermore, we always look at the nine closest samples in a 3 × 3 neighborhood and use the same kernel function for all those samples. This allows for efficient parallel evaluation on a GPU. Using this "gather" approach every output pixel is independently processed only once per frame. This is similar to work of

ACM Trans. Graph., Vol. 38, No. 4, Article 28. Publication date: July 2019.

Fig. 7. **Anisotropic Kernels: Left:** When isotropic kernels ($k_{stretch} = 1$, $k_{shrink} = 1$, see supplemental material) are used, small misalignments cause heavy zipper artifacts along edges. **Right:** Anisotropic kernels ($k_{stretch} = 4$, $k_{shrink} = 2$) fix the artifacts.

Fig. 8. **Merge kernels:** Plots of relative weights in different 3 × 3 sampling kernels as a function of local tensor features.

Yu and Turk [2013], developed for fluid rendering. Two steps described in the following sections are: estimation of the kernel shape (Section 5.1.1) and robustness based sample contribution weighting (Section 5.2).

5.1.1 Local Anisotropic Merge Kernels. Given our problem formulation, kernel weights and kernel functions define the image quality of the final merged image: kernels with wide spatial support produce noise-free and artifact-free, but blurry images, while kernels with very narrow support can produce sharp and detailed images. A natural choice for kernels used for signal reconstruction are *Radial Basis Function* kernels - in our case anisotropic Gaussian kernels. We can adjust the kernel shape to different local properties of the input frames: amounts of detail and the presence of edges (Figure 6). This is similar to kernel selection techniques used in other sparse data reconstruction applications [Takeda et al. 2006, 2007; Yu and Turk 2013].

Specifically, we use a 2D unnormalized anisotropic Gaussian RBF for $w_{n,i}$:

$$w_{n,i} = \exp\left(-\frac{1}{2} d_i^T \Omega^{-1} d_i\right), \tag{2}$$

where Ω is the kernel covariance matrix and d_i is the offset vector of sample i to the output pixel ($d_i = [x_i - x_0, y_i - y_0]^T$).

One of the main motivations for using anisotropic kernels is that they increase the algorithm's tolerance for small misalignments and uneven coverage around edges. Edges are ambiguous in the alignment procedure (due to the aperture problem) and result in alignment errors [Robinson and Milanfar 2004] more frequently compared to non-edge regions of the image. Subpixel misalignment as well as a lack of sufficient sample coverage can manifest as *zipper artifacts* (Figure 7). By stretching the kernels along the edges, we can enforce the assignment of smaller weights to pixels not belonging to edges in the image.

5.1.2 Kernel Covariance Computation. We compute the kernel covariance matrix by analyzing every frame's local gradient structure tensor. To improve runtime performance and resistance to image noise, we analyze gradients of half-resolution images formed by decimating the original raw frames by a factor of two. To decimate a Bayer image containing different color channels, we create a single pixel from a 2 × 2 Bayer quad by combining four different color channels together. This way, we can operate on single channel luminance images and perform the computation at a quarter of the full resolution cost and with improved signal-to-noise ratio. To estimate local information about strength and direction of gradients, we use gradient structure tensor analysis [Bigün et al. 1991; Harris and Stephens 1988]:

$$\widehat{\Omega} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \tag{3}$$

where I_x and I_y are the local image gradients in horizontal and vertical directions, respectively. The image gradients are computed by finite forward differencing the luminance in a small, 3 × 3 color window (giving us four different horizontal and vertical gradient values). Eigenanalysis of the local structure tensor $\widehat{\Omega}$ gives two orthogonal direction vectors $\mathbf{e}_1$, $\mathbf{e}_2$ and two associated eigenvalues λ_1, λ_2. From this, we can construct the kernel covariance as:

$$\Omega = \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 \end{bmatrix} \begin{bmatrix} k_1 & 0 \\ 0 & k_2 \end{bmatrix} \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \end{bmatrix}, \tag{4}$$

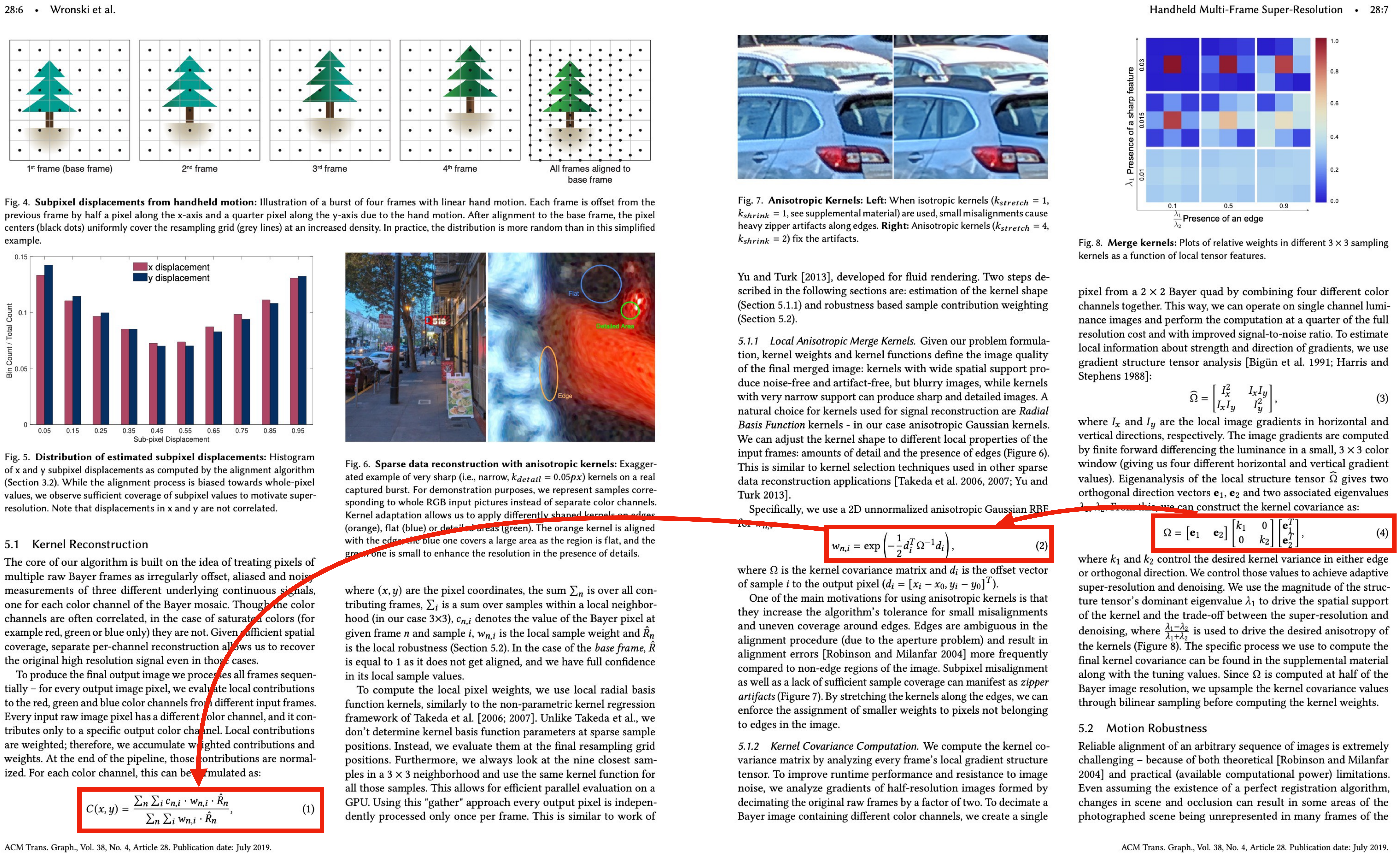
where k_1 and k_2 control the desired kernel variance in either edge or orthogonal direction. We control those values to achieve adaptive super-resolution and denoising. We use the magnitude of the structure tensor's dominant eigenvalue λ_1 to drive the spatial support of the kernel and the trade-off between the super-resolution and denoising, where $\frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2}$ is used to drive the desired anisotropy of the kernels (Figure 8). The specific process we use to compute the final kernel covariance can be found in the supplemental material along with the tuning values. Since Ω is computed at half of the Bayer image resolution, we upsample the kernel covariance values through bilinear sampling before computing the kernel weights.

5.2 Motion Robustness

Reliable alignment of an arbitrary sequence of images is extremely challenging – because of both theoretical [Robinson and Milanfar 2004] and practical (available computational power) limitations. Even assuming the existence of a perfect registration algorithm, changes in scene and occlusion can result in some areas of the photographed scene being unrepresented in many frames of the

ACM Trans. Graph., Vol. 38, No. 4, Article 28. Publication date: July 2019.

Math appears out of order [Wronski et al. 2019]



28:6 • Wronski et al.

Fig. 4. **Subpixel displacements from handheld motion:** Illustration of a burst of four frames with linear hand motion. Each frame is offset from the previous frame by half a pixel along the x-axis and a quarter pixel along the y-axis due to the hand motion. After alignment to the base frame, the pixel centers (black dots) uniformly cover the resampling grid (grey lines) at an increased density. In practice, the distribution is more random than in this simplified example.

Fig. 5. **Distribution of estimated subpixel displacements:** Histogram of x and y subpixel displacements as computed by the alignment algorithm (Section 3.2). While the alignment process is biased towards whole-pixel values, we observe sufficient coverage of subpixel values to motivate super-resolution. Note that displacements in x and y are not correlated.

Fig. 6. **Sparse data reconstruction with anisotropic kernels:** Exaggerated example of very sharp (i.e., narrow, $k_{detail} = 0.05px$) kernels on a real captured burst. For demonstration purposes, we represent samples corresponding to whole RGB input pictures instead of separate color channels. Kernel adaptation allows us to apply differently shaped kernels on edges (orange), flat (blue) or detailed areas (green). The orange kernel is aligned with the edge, the blue one covers a large area as the region is flat, and the green one is small to enhance the resolution in the presence of details.

5.1 Kernel Reconstruction

The core of our algorithm is built on the idea of treating pixels of multiple raw Bayer frames as irregularly offset, aliased and noisy measurements of three different underlying continuous signals, one for each color channel of the Bayer mosaic. Though the color channels are often correlated, in the case of saturated colors (for example red, green or blue only) they are not. Given sufficient spatial coverage, separate per-channel reconstruction allows us to recover the original high resolution signal even in those cases.

To produce the final output image we processes all frames sequentially – for every output image pixel, we evaluate local contributions to the red, green and blue color channels from different input frames. Every input raw image pixel has a different color channel, and it contributes only to a specific output color channel. Local contributions are weighted; therefore, we accumulate weighted contributions and weights. At the end of the pipeline, those contributions are normalized. For each color channel, this can be formulated as:

$$\boxed{C(x,y) = \frac{\sum_n \sum_i c_{n,i} \cdot w_{n,i} \cdot \hat{R}_n}{\sum_n \sum_i w_{n,i} \cdot \hat{R}_n}, \qquad (1)}$$

where (x,y) are the pixel coordinates, the sum $\sum_n$ is over all contributing frames, $\sum_i$ is a sum over samples within a local neighborhood (in our case 3×3), $c_{n,i}$ denotes the value of the Bayer pixel at given frame n and sample i, $w_{n,i}$ is the local sample weight and $\hat{R}_n$ is the local robustness (Section 5.2). In the case of the *base frame*, $\hat{R}$ is equal to 1 as it does not get aligned, and we have full confidence in its local sample values.

To compute the local pixel weights, we use local radial basis function kernels, similarly to the non-parametric kernel regression framework of Takeda et al. [2006; 2007]. Unlike Takeda et al., we don't determine kernel basis function parameters at sparse sample positions. Instead, we evaluate them at the final resampling grid positions. Furthermore, we always look at the nine closest samples in a 3 × 3 neighborhood and use the same kernel function for all those samples. This allows for efficient parallel evaluation on a GPU. Using this "gather" approach every output pixel is independently processed only once per frame. This is similar to work of

ACM Trans. Graph., Vol. 38, No. 4, Article 28. Publication date: July 2019.

Handheld Multi-Frame Super-Resolution • 28:7

Fig. 7. **Anisotropic Kernels: Left:** When isotropic kernels ($k_{stretch} = 1$, $k_{shrink} = 1$, see supplemental material) are used, small misalignments cause heavy zipper artifacts along edges. **Right:** Anisotropic kernels ($k_{stretch} = 4$, $k_{shrink} = 2$) fix the artifacts.

Yu and Turk [2013], developed for fluid rendering. Two steps described in the following sections are: estimation of the kernel shape (Section 5.1.1) and robustness based sample contribution weighting (Section 5.2).

5.1.1 Local Anisotropic Merge Kernels. Given our problem formulation, kernel weights and kernel functions define the image quality of the final merged image: kernels with wide spatial support produce noise-free and artifact-free, but blurry images, while kernels with very narrow support can produce sharp and detailed images. A natural choice for kernels used for signal reconstruction are *Radial Basis Function* kernels - in our case anisotropic Gaussian kernels. We can adjust the kernel shape to different local properties of the input frames: amounts of detail and the presence of edges (Figure 6). This is similar to kernel selection techniques used in other sparse data reconstruction applications [Takeda et al. 2006, 2007; Yu and Turk 2013].

Specifically, we use a 2D unnormalized anisotropic Gaussian RBF for $w_{n,i}$:

$$\boxed{w_{n,i} = \exp\left(-\frac{1}{2} d_i^T \Omega^{-1} d_i\right), \qquad (2)}$$

where Ω is the kernel covariance matrix and d_i is the offset vector of sample i to the output pixel ($d_i = [x_i - x_0, y_i - y_0]^T$).

One of the main motivations for using anisotropic kernels is that they increase the algorithm's tolerance for small misalignments and uneven coverage around edges. Edges are ambiguous in the alignment procedure (due to the aperture problem) and result in alignment errors [Robinson and Milanfar 2004] more frequently compared to non-edge regions of the image. Subpixel misalignment as well as a lack of sufficient sample coverage can manifest as *zipper artifacts* (Figure 7). By stretching the kernels along the edges, we can enforce the assignment of smaller weights to pixels not belonging to edges in the image.

5.1.2 Kernel Covariance Computation. We compute the kernel covariance matrix by analyzing every frame's local gradient structure tensor. To improve runtime performance and resistance to image noise, we analyze gradients of half-resolution images formed by decimating the original raw frames by a factor of two. To decimate a Bayer image containing different color channels, we create a single pixel from a 2 × 2 Bayer quad by combining four different color channels together. This way, we can operate on single channel luminance images and perform the computation at a quarter of the full resolution cost and with improved signal-to-noise ratio. To estimate local information about strength and direction of gradients, we use gradient structure tensor analysis [Bigün et al. 1991; Harris and Stephens 1988]:

$$\widehat{\Omega} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \qquad (3)$$

where I_x and I_y are the local image gradients in horizontal and vertical directions, respectively. The image gradients are computed by finite forward differencing the luminance in a small, 3 × 3 color window (giving us four different horizontal and vertical gradient values). Eigenanalysis of the local structure tensor $\widehat{\Omega}$ gives two orthogonal direction vectors $\mathbf{e}_1$, $\mathbf{e}_2$ and two associated eigenvalues λ_1, λ_2. From this, we can construct the kernel covariance as:

$$\boxed{\Omega = \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 \end{bmatrix} \begin{bmatrix} k_1 & 0 \\ 0 & k_2 \end{bmatrix} \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \end{bmatrix}, \qquad (4)}$$

where k_1 and k_2 control the desired kernel variance in either edge or orthogonal direction. We control those values to achieve adaptive super-resolution and denoising. We use the magnitude of the structure tensor's dominant eigenvalue λ_1 to drive the spatial support of the kernel and the trade-off between the super-resolution and denoising, where $\frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2}$ is used to drive the desired anisotropy of the kernels (Figure 8). The specific process we use to compute the final kernel covariance can be found in the supplemental material along with the tuning values. Since Ω is computed at half of the Bayer image resolution, we upsample the kernel covariance values through bilinear sampling before computing the kernel weights.

Fig. 8. **Merge kernels:** Plots of relative weights in different 3 × 3 sampling kernels as a function of local tensor features.

5.2 Motion Robustness

Reliable alignment of an arbitrary sequence of images is extremely challenging – because of both theoretical [Robinson and Milanfar 2004] and practical (available computational power) limitations. Even assuming the existence of a perfect registration algorithm, changes in scene and occlusion can result in some areas of the photographed scene being unrepresented in many frames of the

ACM Trans. Graph., Vol. 38, No. 4, Article 28. Publication date: July 2019.

Math appears out of order [Wronski et al. 2019]

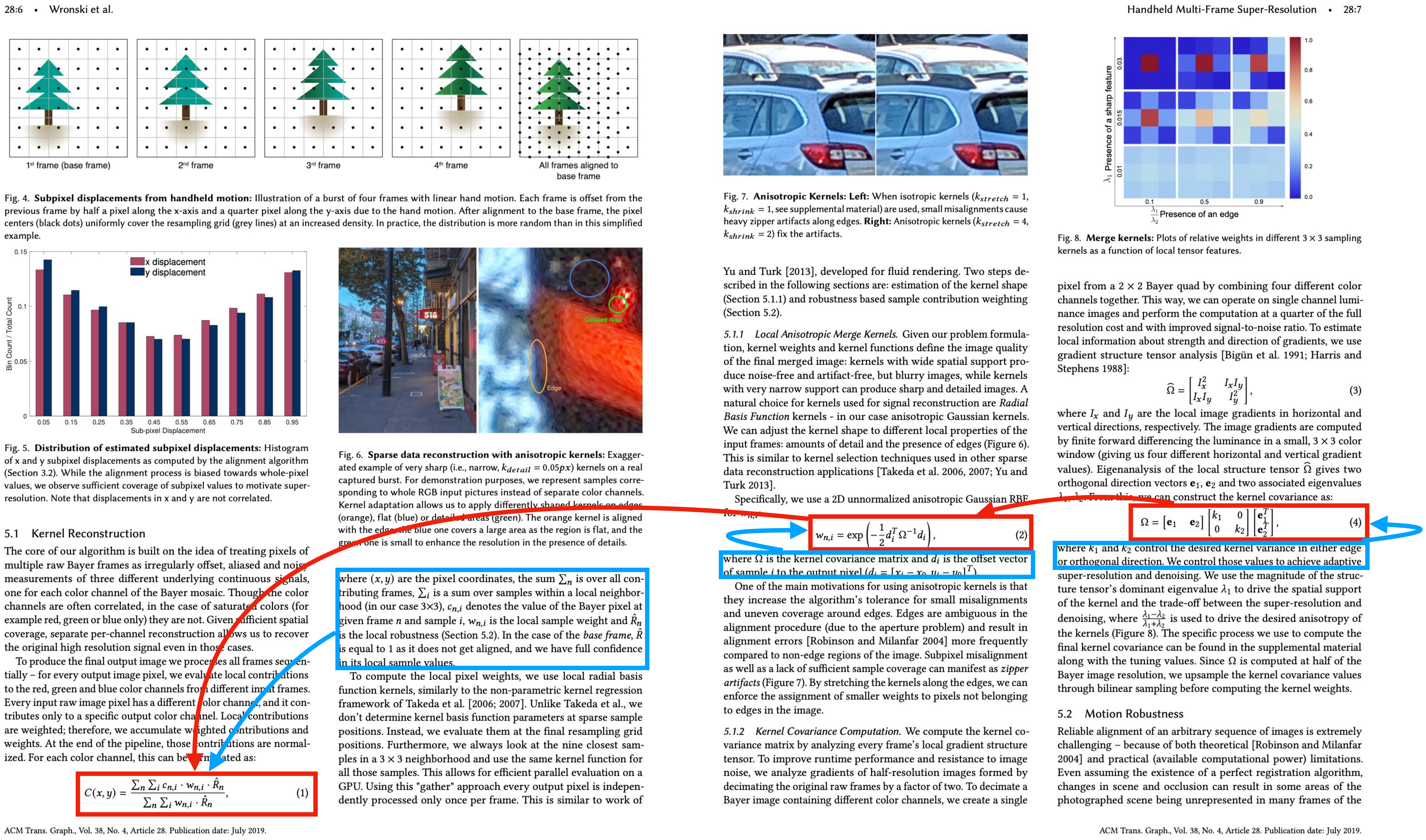


Fig. 4. **Subpixel displacements from handheld motion:** Illustration of a burst of four frames with linear hand motion. Each frame is offset from the previous frame by half a pixel along the x-axis and a quarter pixel along the y-axis due to the hand motion. After alignment to the base frame, the pixel centers (black dots) uniformly cover the resampling grid (grey lines) at an increased density. In practice, the distribution is more random than in this simplified example.

Fig. 5. **Distribution of estimated subpixel displacements:** Histogram of x and y subpixel displacements as computed by the alignment algorithm (Section 3.2). While the alignment process is biased towards whole-pixel values, we observe sufficient coverage of subpixel values to motivate super-resolution. Note that displacements in x and y are not correlated.

Fig. 6. **Sparse data reconstruction with anisotropic kernels:** Exaggerated example of very sharp (i.e., narrow, $k_{detail} = 0.05px$) kernels on a real captured burst. For demonstration purposes, we represent samples corresponding to whole RGB input pictures instead of separate color channels. Kernel adaptation allows us to apply differently shaped kernels on edges (orange), flat (blue) or detailed areas (green). The orange kernel is aligned with the edge, the blue one covers a large area as the region is flat, and the green one is small to enhance the resolution in the presence of details.

5.1 Kernel Reconstruction

The core of our algorithm is built on the idea of treating pixels of multiple raw Bayer frames as irregularly offset, aliased and noisy measurements of three different underlying continuous signals, one for each color channel of the Bayer mosaic. Though the color channels are often correlated, in the case of saturated colors (for example red, green or blue only) they are not. Given sufficient spatial coverage, separate per-channel reconstruction allows us to recover the original high resolution signal even in those cases.

To produce the final output image we processes all frames sequentially – for every output image pixel, we evaluate local contributions to the red, green and blue color channels from different input frames. Every input raw image pixel has a different color channel, and it contributes only to a specific output color channel. Local contributions are weighted; therefore, we accumulate weighted contributions and weights. At the end of the pipeline, those contributions are normalized. For each color channel, this can be formulated as:

$$C(x, y) = \frac{\sum_n \sum_i c_{n,i} \cdot w_{n,i} \cdot \hat{R}_n}{\sum_n \sum_i w_{n,i} \cdot \hat{R}_n}, \tag{1}$$

where (x, y) are the pixel coordinates, the sum $\sum_n$ is over all contributing frames, $\sum_i$ is a sum over samples within a local neighborhood (in our case 3×3), $c_{n,i}$ denotes the value of the Bayer pixel at given frame n and sample i, $w_{n,i}$ is the local sample weight and $\hat{R}_n$ is the local robustness (Section 5.2). In the case of the *base frame*, $\hat{R}$ is equal to 1 as it does not get aligned, and we have full confidence in its local sample values.

To compute the local pixel weights, we use local radial basis function kernels, similarly to the non-parametric kernel regression framework of Takeda et al. [2006; 2007]. Unlike Takeda et al., we don't determine kernel basis function parameters at sparse sample positions. Instead, we evaluate them at the final resampling grid positions. Furthermore, we always look at the nine closest samples in a 3 × 3 neighborhood and use the same kernel function for all those samples. This allows for efficient parallel evaluation on a GPU. Using this "gather" approach every output pixel is independently processed only once per frame. This is similar to work of

Fig. 7. **Anisotropic Kernels: Left:** When isotropic kernels ($k_{stretch}$ = 1, k_{shrink} = 1, see supplemental material) are used, small misalignments cause heavy zipper artifacts along edges. **Right:** Anisotropic kernels ($k_{stretch}$ = 4, k_{shrink} = 2) fix the artifacts.

Yu and Turk [2013], developed for fluid rendering. Two steps described in the following sections are: estimation of the kernel shape (Section 5.1.1) and robustness based sample contribution weighting (Section 5.2).

5.1.1 Local Anisotropic Merge Kernels. Given our problem formulation, kernel weights and kernel functions define the image quality of the final merged image: kernels with wide spatial support produce noise-free and artifact-free, but blurry images, while kernels with very narrow support can produce sharp and detailed images. A natural choice for kernels used for signal reconstruction are *Radial Basis Function* kernels - in our case anisotropic Gaussian kernels. We can adjust the kernel shape to different local properties of the input frames: amounts of detail and the presence of edges (Figure 6). This is similar to kernel selection techniques used in other sparse data reconstruction applications [Takeda et al. 2006, 2007; Yu and Turk 2013].

Specifically, we use a 2D unnormalized anisotropic Gaussian RBF for $w_{n,i}$:

$$w_{n,i} = \exp\left(-\frac{1}{2} d_i^T \Omega^{-1} d_i\right), \tag{2}$$

where Ω is the kernel covariance matrix and d_i is the offset vector of sample i to the output pixel ($d_i = [x_i - x_0, y_i - y_0]^T$).

One of the main motivations for using anisotropic kernels is that they increase the algorithm's tolerance for small misalignments and uneven coverage around edges. Edges are ambiguous in the alignment procedure (due to the aperture problem) and result in alignment errors [Robinson and Milanfar 2004] more frequently compared to non-edge regions of the image. Subpixel misalignment as well as a lack of sufficient sample coverage can manifest as *zipper artifacts* (Figure 7). By stretching the kernels along the edges, we can enforce the assignment of smaller weights to pixels not belonging to edges in the image.

5.1.2 Kernel Covariance Computation. We compute the kernel covariance matrix by analyzing every frame's local gradient structure tensor. To improve runtime performance and resistance to image noise, we analyze gradients of half-resolution images formed by decimating the original raw frames by a factor of two. To decimate a Bayer image containing different color channels, we create a single pixel from a 2 × 2 Bayer quad by combining four different color channels together. This way, we can operate on single channel luminance images and perform the computation at a quarter of the full resolution cost and with improved signal-to-noise ratio. To estimate local information about strength and direction of gradients, we use gradient structure tensor analysis [Bigün et al. 1991; Harris and Stephens 1988]:

$$\widehat{\Omega} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \tag{3}$$

where I_x and I_y are the local image gradients in horizontal and vertical directions, respectively. The image gradients are computed by finite forward differencing the luminance in a small, 3 × 3 color window (giving us four different horizontal and vertical gradient values). Eigenanalysis of the local structure tensor $\widehat{\Omega}$ gives two orthogonal direction vectors $\mathbf{e}_1$, $\mathbf{e}_2$ and two associated eigenvalues λ_1, λ_2. From this, we can construct the kernel covariance as:

$$\Omega = \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 \end{bmatrix} \begin{bmatrix} k_1 & 0 \\ 0 & k_2 \end{bmatrix} \begin{bmatrix} \mathbf{e}_1^T \\ \mathbf{e}_2^T \end{bmatrix}, \tag{4}$$

where k_1 and k_2 control the desired kernel variance in either edge or orthogonal direction. We control those values to achieve adaptive super-resolution and denoising. We use the magnitude of the structure tensor's dominant eigenvalue λ_1 to drive the spatial support of the kernel and the trade-off between the super-resolution and denoising, where $\frac{\lambda_1 - \lambda_2}{\lambda_1 + \lambda_2}$ is used to drive the desired anisotropy of the kernels (Figure 8). The specific process we use to compute the final kernel covariance can be found in the supplemental material along with the tuning values. Since Ω is computed at half of the Bayer image resolution, we upsample the kernel covariance values through bilinear sampling before computing the kernel weights.

Fig. 8. **Merge kernels:** Plots of relative weights in different 3 × 3 sampling kernels as a function of local tensor features.

5.2 Motion Robustness

Reliable alignment of an arbitrary sequence of images is extremely challenging – because of both theoretical [Robinson and Milanfar 2004] and practical (available computational power) limitations. Even assuming the existence of a perfect registration algorithm, changes in scene and occlusion can result in some areas of the photographed scene being unrepresented in many frames of the

Formative Study

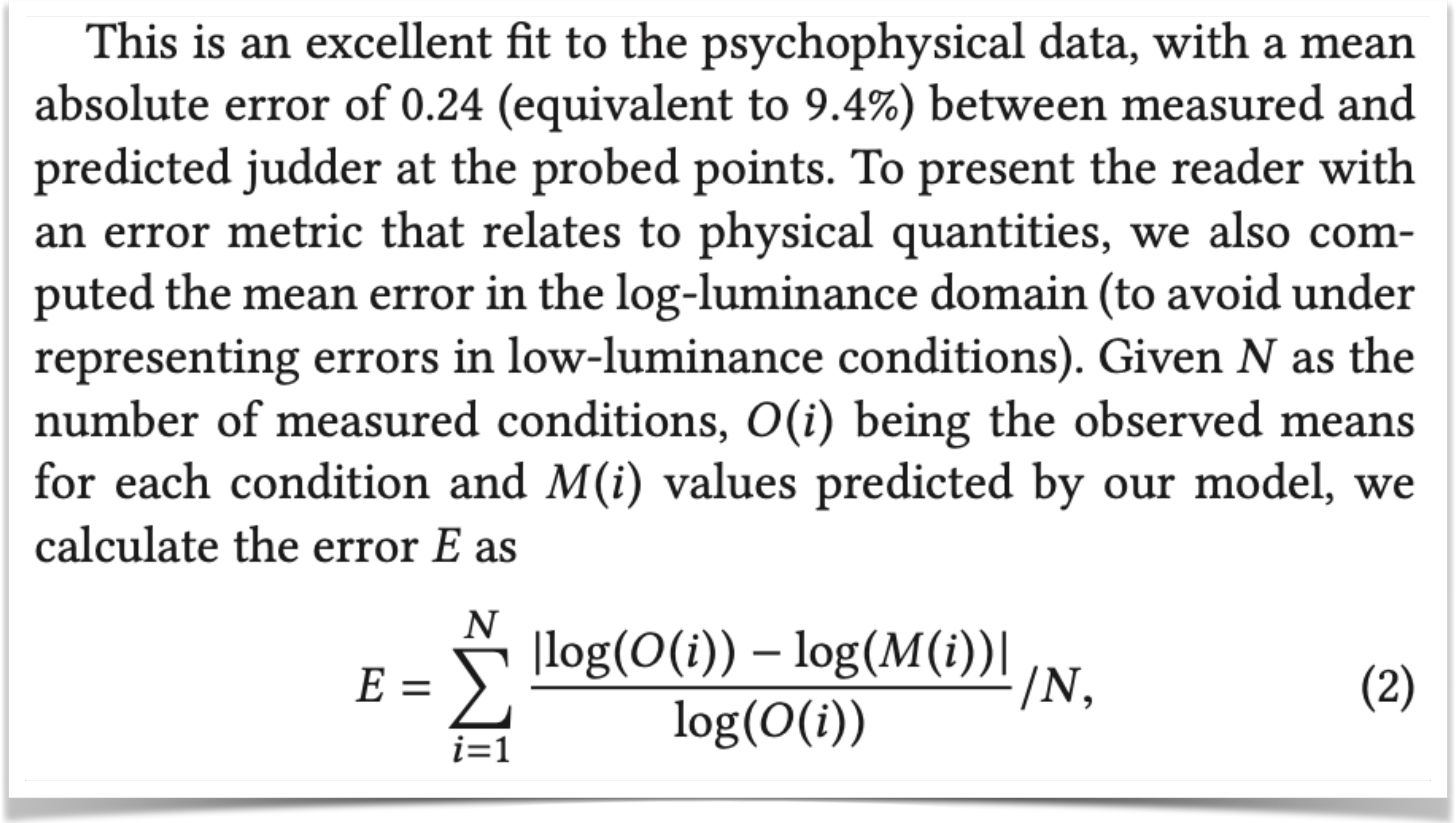
- All appear to be written using LaTeX.
- Observations:
  I. Prose organizes the document, interleaved with math.
  II. Math appears out of order. Symbols used before defined.

Formative Study

- All appear to be written using LaTeX.
- Observations:
  I. Prose organizes the document, interleaved with math.
  II. Math appears out of order. Symbols used before defined.
  III. Symbols re-used in different contexts.

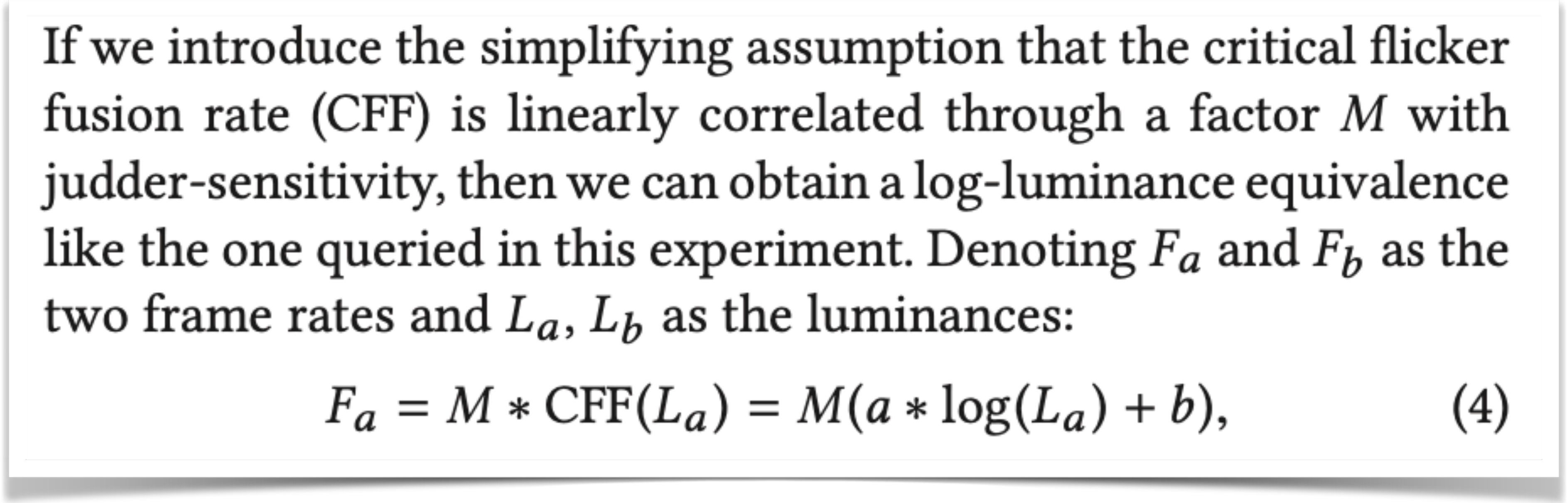
Formative Study

- All appear to be written using LaTeX.
- Observations:
  - I. Prose organizes the document, interleaved with math.
  - II. Math appears out of order. Symbols used before defined.
  - III. Symbols re-used in different contexts.



This is an excellent fit to the psychophysical data, with a mean absolute error of 0.24 (equivalent to 9.4%) between measured and predicted judder at the probed points. To present the reader with an error metric that relates to physical quantities, we also computed the mean error in the log-luminance domain (to avoid under representing errors in low-luminance conditions). Given N as the number of measured conditions, $O(i)$ being the observed means for each condition and $M(i)$ values predicted by our model, we calculate the error E as

$$E = \sum_{i=1}^{N} \frac{|\log(O(i)) - \log(M(i))|}{\log(O(i))} / N, \tag{2}$$



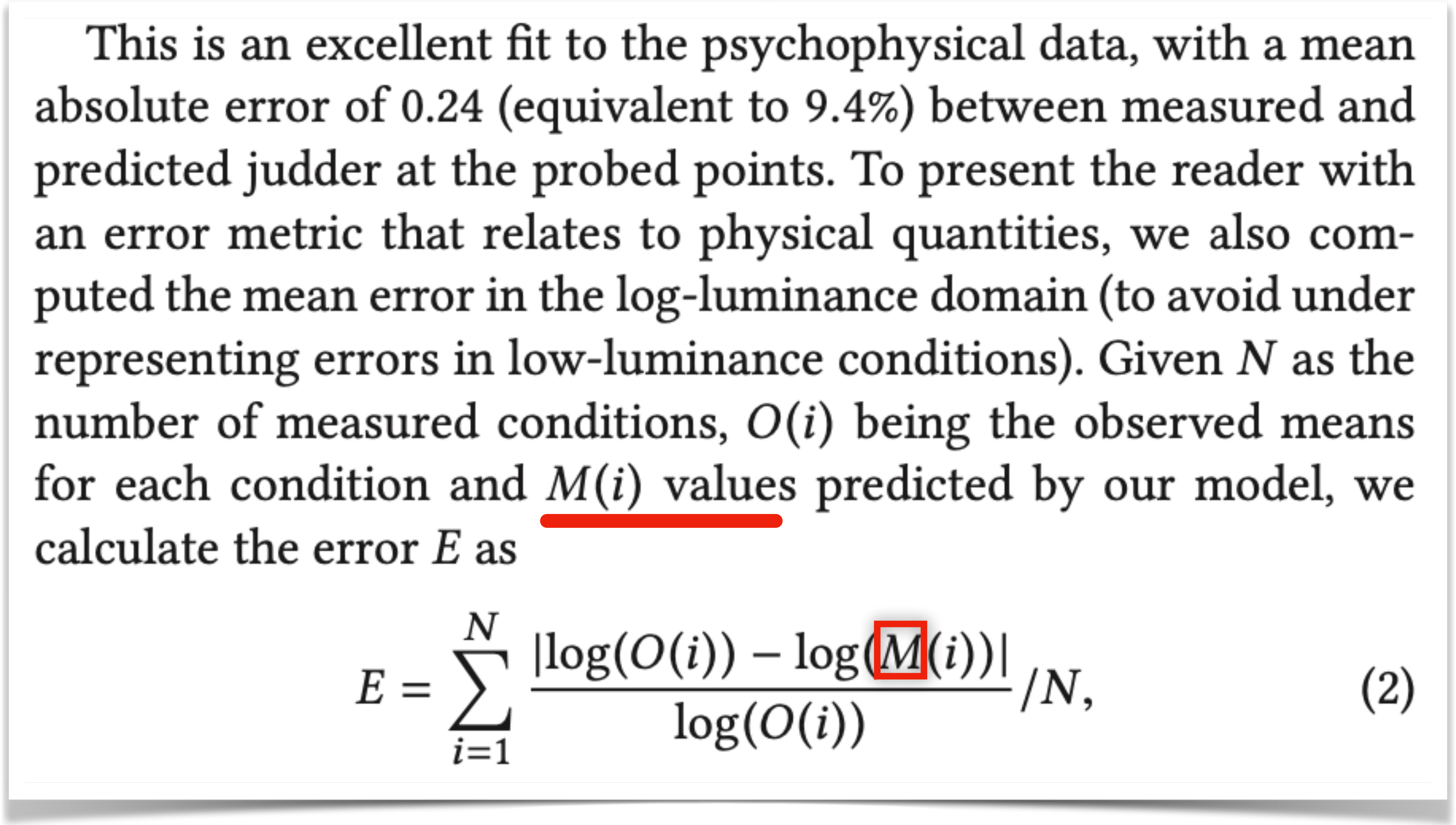
If we introduce the simplifying assumption that the critical flicker fusion rate (CFF) is linearly correlated through a factor M with judder-sensitivity, then we can obtain a log-luminance equivalence like the one queried in this experiment. Denoting F_a and F_b as the two frame rates and L_a, L_b as the luminances:

$$F_a = M * \text{CFF}(L_a) = M(a * \log(L_a) + b), \tag{4}$$

[Chapiro et al. 2019]

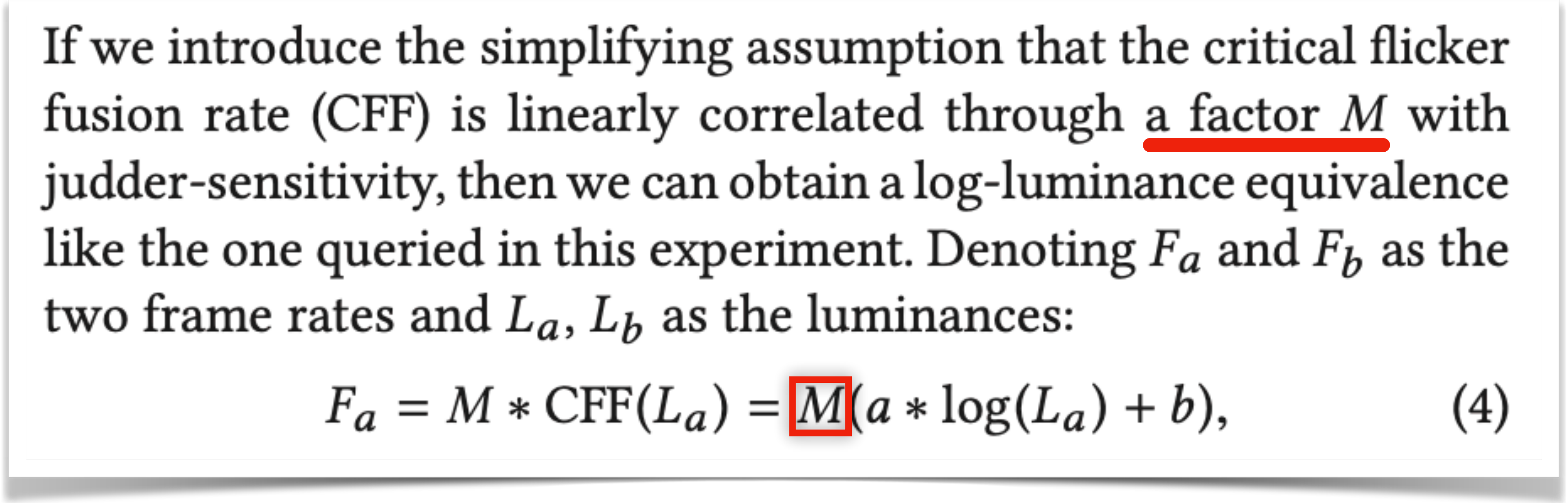
Formative Study

- All appear to be written using LaTeX.
- Observations:
  - I. Prose organizes the document, interleaved with math.
  - II. Math appears out of order. Symbols used before defined.
  - III. Symbols re-used in different contexts.



This is an excellent fit to the psychophysical data, with a mean absolute error of 0.24 (equivalent to 9.4%) between measured and predicted judder at the probed points. To present the reader with an error metric that relates to physical quantities, we also computed the mean error in the log-luminance domain (to avoid under representing errors in low-luminance conditions). Given N as the number of measured conditions, $O(i)$ being the observed means for each condition and $\underline{M(i)\text{ values}}$ predicted by our model, we calculate the error E as

$$E = \sum_{i=1}^{N} \frac{|\log(O(i)) - \log(\boxed{M}(i))|}{\log(O(i))} / N, \qquad (2)$$



If we introduce the simplifying assumption that the critical flicker fusion rate (CFF) is linearly correlated through $\underline{\text{a factor } M}$ with judder-sensitivity, then we can obtain a log-luminance equivalence like the one queried in this experiment. Denoting F_a and F_b as the two frame rates and L_a, L_b as the luminances:

$$F_a = M * \mathrm{CFF}(L_a) = \boxed{M}(a * \log(L_a) + b), \qquad (4)$$

[Chapiro et al. 2019]

Formative Study

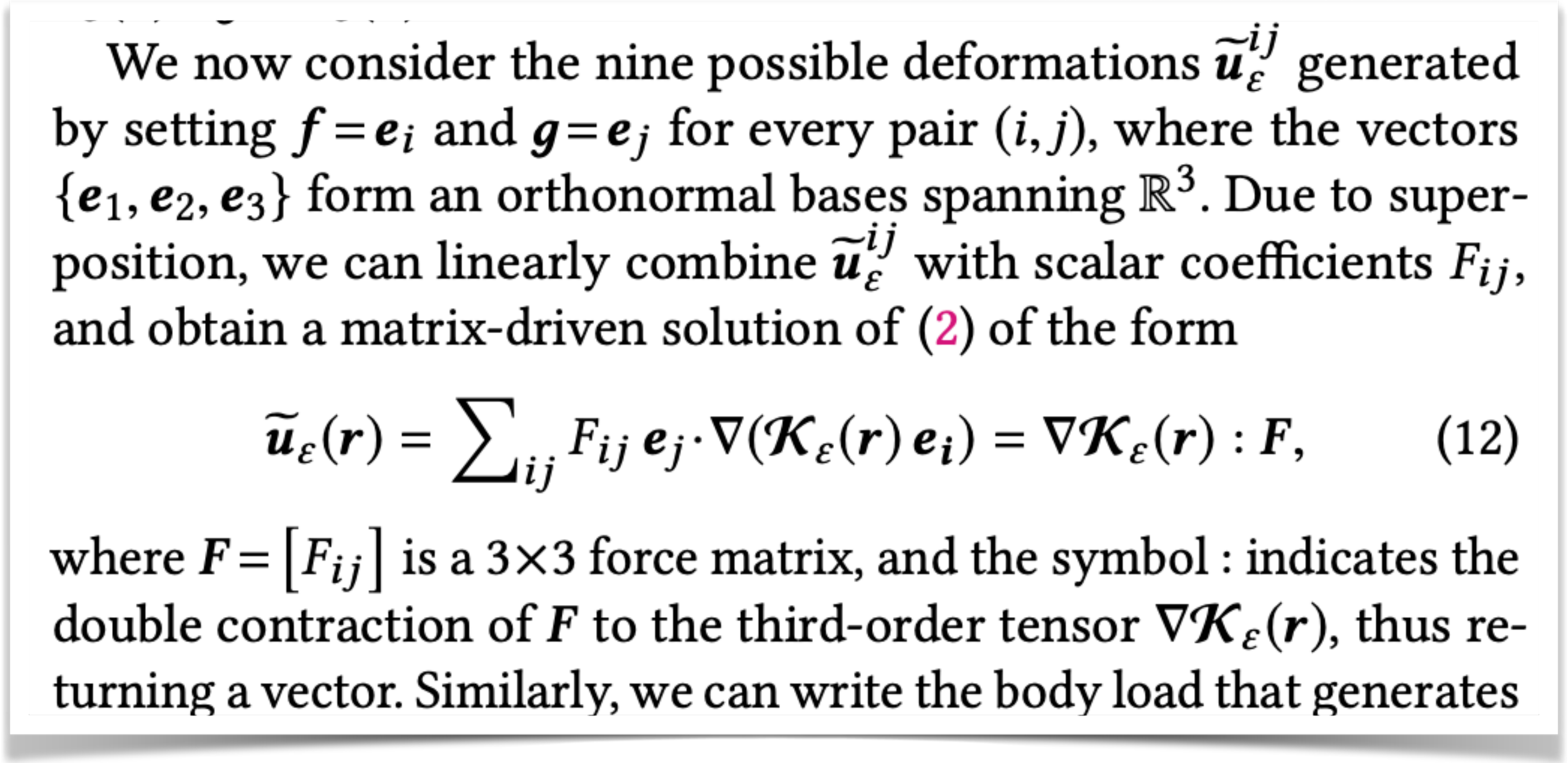
- All appear to be written using LaTeX.
- Observations:
  I. Prose organizes the document, interleaved with math.
  II. Math appears out of order. Symbols used before defined.
  III. Symbols re-used in different contexts.

Formative Study

- All appear to be written using LaTeX.
- Observations:
    I. Prose organizes the document, interleaved with math.
    II. Math appears out of order. Symbols used before defined.
    III. Symbols re-used in different contexts.
    IV. Symbol appears in executable formulas and non-executable derivations.

Formative Study

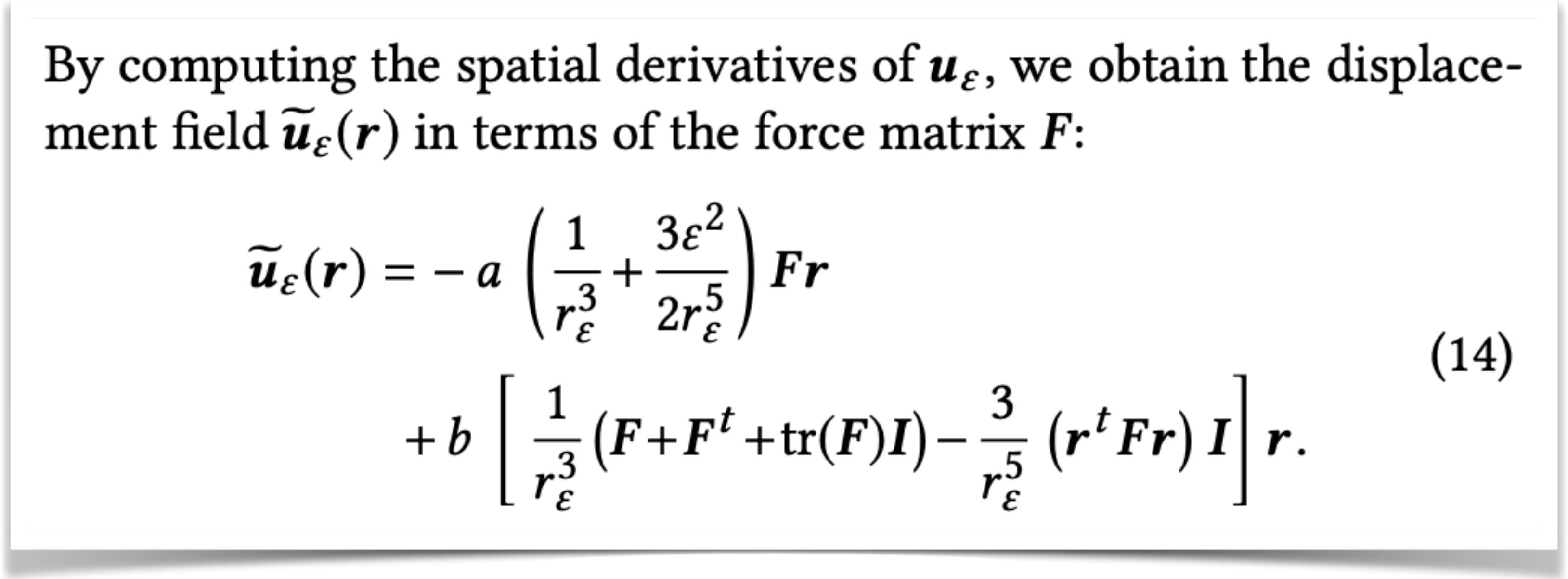
- All appear to be written using LaTeX.
- Observations:
    - I. Prose organizes the document, interleaved with math.
    - II. Math appears out of order. Symbols used before defined.
    - III. Symbols re-used in different contexts.
    - IV. Symbol appears in executable formulas and non-executable derivations.



We now consider the nine possible deformations $\widetilde{\boldsymbol{u}}_\varepsilon^{ij}$ generated by setting $\boldsymbol{f}=\boldsymbol{e}_i$ and $\boldsymbol{g}=\boldsymbol{e}_j$ for every pair (i,j), where the vectors $\{\boldsymbol{e}_1,\boldsymbol{e}_2,\boldsymbol{e}_3\}$ form an orthonormal bases spanning $\mathbb{R}^3$. Due to superposition, we can linearly combine $\widetilde{\boldsymbol{u}}_\varepsilon^{ij}$ with scalar coefficients F_{ij}, and obtain a matrix-driven solution of (2) of the form

$$\widetilde{\boldsymbol{u}}_\varepsilon(\boldsymbol{r}) = \sum_{ij} F_{ij}\, \boldsymbol{e}_j \cdot \nabla(\boldsymbol{\mathcal{K}}_\varepsilon(\boldsymbol{r})\, \boldsymbol{e}_i) = \nabla\boldsymbol{\mathcal{K}}_\varepsilon(\boldsymbol{r}) : \boldsymbol{F}, \qquad (12)$$

where $\boldsymbol{F}=\left[F_{ij}\right]$ is a 3×3 force matrix, and the symbol : indicates the double contraction of $\boldsymbol{F}$ to the third-order tensor $\nabla\boldsymbol{\mathcal{K}}_\varepsilon(\boldsymbol{r})$, thus returning a vector. Similarly, we can write the body load that generates



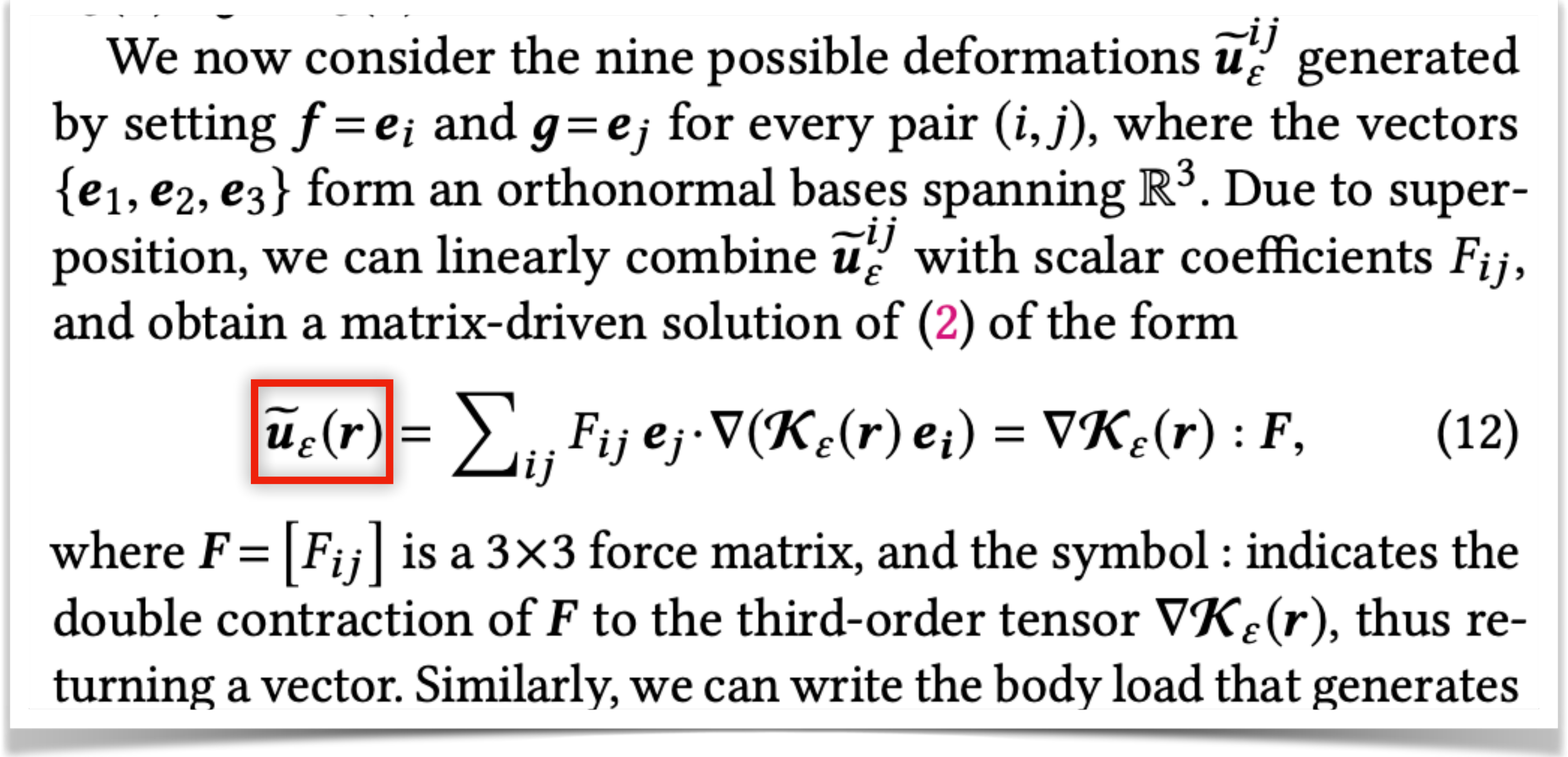
By computing the spatial derivatives of $\boldsymbol{u}_\varepsilon$, we obtain the displacement field $\widetilde{\boldsymbol{u}}_\varepsilon(\boldsymbol{r})$ in terms of the force matrix $\boldsymbol{F}$:

$$\begin{aligned}\widetilde{\boldsymbol{u}}_\varepsilon(\boldsymbol{r}) = &-a\left(\frac{1}{r_\varepsilon^3}+\frac{3\varepsilon^2}{2r_\varepsilon^5}\right)\boldsymbol{F}\boldsymbol{r} \\ &+b\left[\frac{1}{r_\varepsilon^3}(\boldsymbol{F}+\boldsymbol{F}^t+\mathrm{tr}(\boldsymbol{F})\boldsymbol{I})-\frac{3}{r_\varepsilon^5}\left(\boldsymbol{r}^t\boldsymbol{F}\boldsymbol{r}\right)\boldsymbol{I}\right]\boldsymbol{r}.\end{aligned} \qquad (14)$$

[De Goes and James 2017]

Formative Study

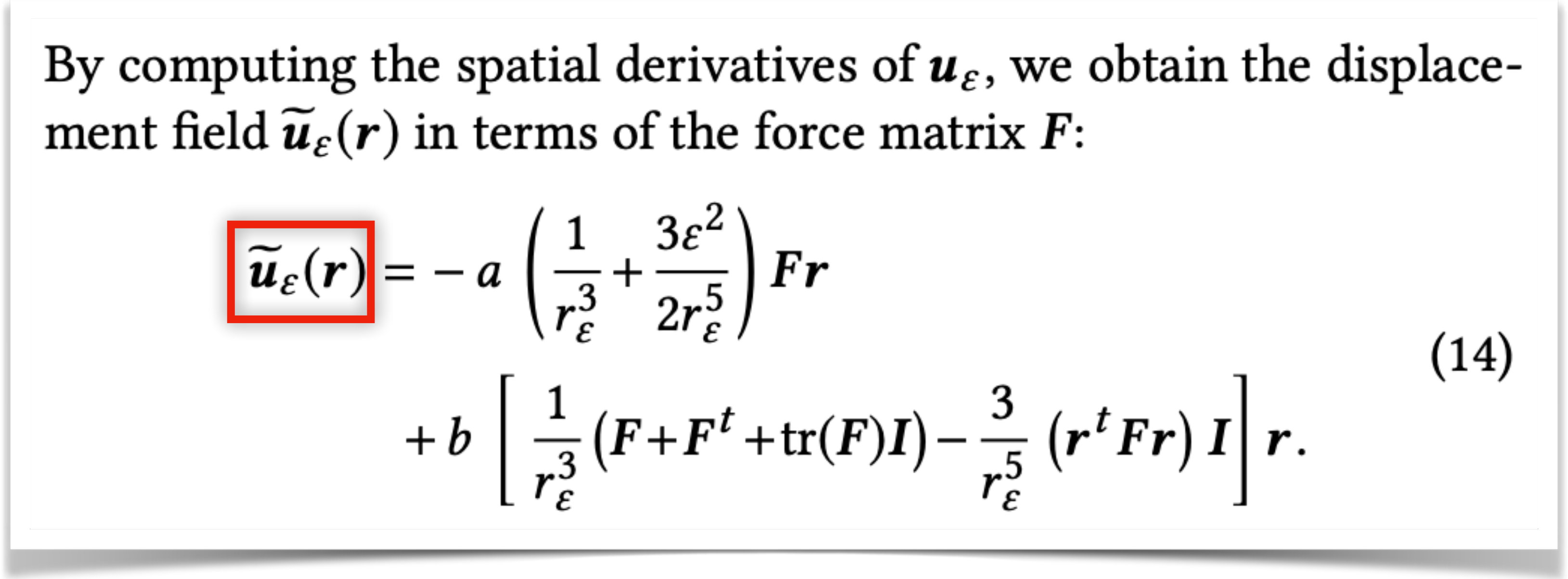
- All appear to be written using LaTeX.
- Observations:
  1. Prose organizes the document, interleaved with math.
  2. Math appears out of order. Symbols used before defined.
  3. Symbols re-used in different contexts.
  4. Symbol appears in executable formulas and non-executable derivations.



We now consider the nine possible deformations $\widetilde{\boldsymbol{u}}_\varepsilon^{ij}$ generated by setting $\boldsymbol{f}=\boldsymbol{e}_i$ and $\boldsymbol{g}=\boldsymbol{e}_j$ for every pair (i,j), where the vectors $\{\boldsymbol{e}_1,\boldsymbol{e}_2,\boldsymbol{e}_3\}$ form an orthonormal bases spanning $\mathbb{R}^3$. Due to superposition, we can linearly combine $\widetilde{\boldsymbol{u}}_\varepsilon^{ij}$ with scalar coefficients F_{ij}, and obtain a matrix-driven solution of (2) of the form

$$\boxed{\widetilde{\boldsymbol{u}}_\varepsilon(\boldsymbol{r})} = \sum_{ij} F_{ij}\,\boldsymbol{e}_j \cdot \nabla(\boldsymbol{\mathcal{K}}_\varepsilon(\boldsymbol{r})\,\boldsymbol{e}_i) = \nabla\boldsymbol{\mathcal{K}}_\varepsilon(\boldsymbol{r}) : \boldsymbol{F}, \qquad (12)$$

where $\boldsymbol{F}=\left[F_{ij}\right]$ is a 3×3 force matrix, and the symbol : indicates the double contraction of $\boldsymbol{F}$ to the third-order tensor $\nabla\boldsymbol{\mathcal{K}}_\varepsilon(\boldsymbol{r})$, thus returning a vector. Similarly, we can write the body load that generates



By computing the spatial derivatives of $\boldsymbol{u}_\varepsilon$, we obtain the displacement field $\widetilde{\boldsymbol{u}}_\varepsilon(\boldsymbol{r})$ in terms of the force matrix $\boldsymbol{F}$:

$$\begin{aligned}\boxed{\widetilde{\boldsymbol{u}}_\varepsilon(\boldsymbol{r})} = &-a\left(\frac{1}{r_\varepsilon^3}+\frac{3\varepsilon^2}{2r_\varepsilon^5}\right)\boldsymbol{F}\boldsymbol{r} \\ &+b\left[\frac{1}{r_\varepsilon^3}(\boldsymbol{F}+\boldsymbol{F}^t+\mathrm{tr}(\boldsymbol{F})\boldsymbol{I})-\frac{3}{r_\varepsilon^5}\left(\boldsymbol{r}^t\boldsymbol{F}\boldsymbol{r}\right)\boldsymbol{I}\right]\boldsymbol{r}.\end{aligned} \qquad (14)$$

[De Goes and James 2017]

Formative Study

- All appear to be written using LaTeX.
- Observations:
  - I. Prose organizes the document, interleaved with math.
  - II. Math appears out of order. Symbols used before defined.
  - III. Symbols re-used in different contexts.
  - IV. Symbol appears in executable formulas and non-executable derivations.

Formative Study

- All appear to be written using LaTeX.
- Observations:
    I. Prose organizes the document, interleaved with math.
    II. Math appears out of order. Symbols used before defined.
    III. Symbols re-used in different contexts.
    IV. Symbol appears in executable formulas and non-executable derivations.
    V. Symbols and functions appear with conditional assignment.

Formative Study

- All appear to be written using LaTeX.
- Observations:
    I. Prose organizes the document, interleaved with math.
    II. Math appears out of order. Symbols used before defined.
    III. Symbols re-used in different contexts.
    IV. Symbol appears in executable formulas and non-executable derivations.
    V. Symbols and functions appear with conditional assignment.
    VI. Functions have a variety of implied semantics for parameters and pre-computed symbols.

$$f_1(y) = \begin{cases} -\dfrac{y^2}{\epsilon_v^2 h^2} + \dfrac{2y}{\epsilon_v h}, & y \in (0, h\epsilon_v) \\ 1, & y \geq h\epsilon_v, \end{cases} \tag{13}$$

$$\Gamma_{l_0}(d) = \int_{\Omega^0} \gamma_{l_0}(d, \nabla d) \mathrm{dV}, \tag{1}$$

$$G^l_{mn}(I) = \sum_i^{H_l \times W_l} \hat{\mathcal{F}}^l_{mi}(I)\, \hat{\mathcal{F}}^l_{ni}(I). \tag{3}$$

$$\overline{b}(f,g) := \begin{cases} \frac{\lambda}{2} ||(\mathbf{x}_g - \mathbf{x}_f) - (\mathbf{r}_{\ell_g} - \mathbf{r}_{\ell_f})||^2_{\mathbf{W}} & \text{if } |f-g| = 1, \\ 0 & \text{otherwise.} \end{cases}$$

$$\pi(\mathbf{a}|\mathbf{s}) = \sum_{i \in \mathcal{E}} w_i(\mathbf{s}) \pi_i(\mathbf{a}|\mathbf{s}),\ w_i(\mathbf{s}) = \frac{exp\big(g_i(\mathbf{s})\big)}{\sum_{i \in \mathcal{E}} exp\big(g_i(\mathbf{s})\big)} \tag{8}$$

$$\begin{gathered} D(n,m) = \sum_{k \in \kappa} d(n+k, m+k) \\ d(n,m) = w_1 \frac{d_{\mathrm{q}}}{|\mathcal{J}|} + w_2 \frac{d_{\mathrm{v}}}{|\mathcal{J}|} + w_3 \frac{d_{\mathrm{ee}}}{|\mathcal{F}|} + w_4 d_{\mathrm{root}} \end{gathered} \tag{1}$$

$$\Psi(\mathbf{F}^s, J^g) = \Psi^s(\mathbf{F}^s) + \Psi^g(J^g), \tag{9}$$

$$JSD(P||Q) = \frac{1}{2} D(P||M) + \frac{1}{2} D(Q||M) \tag{8}$$

$$\pi(a_{t+1}|s_t, c_t) = \frac{1}{Z(s_t, c_t)} \Pi_{i=1}^k \phi_i^{w_i}, \tag{3}$$

$$d(\mathcal{L}, \ell) = \frac{1}{|\ell|} \int_\ell \min_{\ell_i \in \mathcal{L}} \mathrm{dist}(\ell_i, p_\theta) dp_\theta, \tag{2}$$

$$\mathbf{q}^f(\mathbf{x}) = \sum_c \mathbf{q}^f_c N_c^{f,1}(\mathbf{x}). \tag{28}$$

$$\mathbf{Q}(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos^2 2\theta & \sin 2\theta \cos 2\theta & -\sin 2\theta \\ 0 & \sin 2\theta \cos 2\theta & \sin^2 2\theta & \cos 2\theta \\ 0 & \sin 2\theta & -\cos 2\theta & 0 \end{bmatrix}, \tag{2}$$

$$\begin{aligned} i_3(x_3, y_3, \lambda) &= \widehat{i}_3(x_3, y_3, \lambda) d(x_3, y_3) \\ &= \frac{h(x_0, y_0, \lambda)}{(j\lambda f)^2} A\left(\frac{x_0 + x_3}{\lambda f}, \frac{y_0 + y_3}{\lambda f}\right) \frac{1}{v_0} \sum_{k=-\infty}^{\infty} \delta\left(x_3 - \frac{k}{v_0}\right). \end{aligned} \tag{19}$$

$$\begin{aligned} V[\hat{I}_t] = {} & \frac{1}{M} \sum_{s \geq 1} V\left[\frac{w_t f}{p}\right] - \frac{1}{MN} \sum_{s \geq 1} V\left[\frac{w_t f}{q_s}\right] + \frac{1}{N} V\left[\frac{w_t f}{q}\right] \\ & + \left(1 - \frac{1}{N}\right) V\left[\frac{1}{p(\bar{Z}_t)} \int_{\mathcal{A}} w_t(\bar{y}\bar{Z}_t) f(\bar{y}\bar{Z}_t) d\mu(\bar{y})\right] \\ & - \frac{1}{M}\left(1 - \frac{1}{N}\right) \sum_{s \geq 1} V\left[\frac{1}{p(\bar{Z}_t)} \int_{A^s} w_t(\bar{y}\bar{Z}_t) f(\bar{y}\bar{Z}_t) d\mu(\bar{y})\right]. \end{aligned} \tag{9}$$

$$\begin{aligned} R(\lambda) &= \left| r_{as} + \sum_{k=0}^{\infty} t_{as} r_{sa} \left(r_{sa}^2 e^{i\Delta\phi}\right)^k e^{i\Delta\phi} t_{sa} \right|^2 \\ &= \left| r_{as} + \frac{t_{as} r_{sa} t_{sa} e^{i\Delta\phi}}{1 - r_{sa}^2 e^{i\Delta\phi}} \right|^2, \end{aligned} \tag{27}$$

$$\begin{aligned} E[L_{\boldsymbol{x}_p}] &= \int_0^\pi |\cos\phi_1| \frac{\sin^{n-2}\phi_1 d\phi_1}{I_{n-2}} \\ &= \frac{2}{I_{n-2}} \int_0^{\frac{\pi}{2}} \cos\phi_1 \sin^{n-2}\phi_1 d\phi_1 = \frac{2}{(n-1)I_{n-2}}. \end{aligned} \tag{24}$$

$$J_{RL}(\theta) = E\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right].$$

$$\mathbf{F}_{s \to f}(\mathbf{X}_p) = \frac{\rho\big(\mathbf{u}_b - \mathbf{u}_s\big) \cdot \mathbf{n}}{\Delta t} \mathbf{n}.$$

$$\boldsymbol{a}^{(n)}(\boldsymbol{x}, t) = \int \frac{(2\pi)^{\frac{D}{2}} f(\boldsymbol{x}, \boldsymbol{v}, t)}{e^{-\|\boldsymbol{v}\|^2/2}} \boldsymbol{H}^{(n)}(\boldsymbol{v}) \mathrm{d}\boldsymbol{v} \approx \sum_{i=0}^{q-1} f_i(\boldsymbol{x}, t) \boldsymbol{H}^{(n)}(\boldsymbol{c}_i), \tag{7}$$

$$\begin{aligned} E_{\widetilde{t},\alpha}(t) &= \tfrac{1}{d!} \|Det(Y)\| \\ &= \tfrac{1}{d!} \sqrt{Det(Y^T Y)} \\ &= \tfrac{1}{d!} \sqrt{Det(X^T X + \alpha \widetilde{X}^T \widetilde{X})} \end{aligned} \tag{2}$$

$$\begin{aligned} & \operatorname{corr}\left(x; \mathcal{T}^c, \mathcal{P}^c\right) \\ & = \int_\theta |E(\theta)|^2 \exp\left(-i\kappa\Phi(\theta)\right) \int_\epsilon \mathcal{T}^c(x, x+\epsilon) \exp\left(-i\kappa\epsilon\theta\right) \,\mathrm{d}\epsilon \,\mathrm{d}\theta \\ & = \int_\epsilon \mathcal{T}^c(x, x+\epsilon) \underbrace{\int_\theta A(\theta) \exp\left(-i\kappa\Phi(\theta) - i\kappa\epsilon\theta\right) \,\mathrm{d}\theta}_{\equiv \mathcal{P}^c(\epsilon)} \,\mathrm{d}\epsilon, \end{aligned} \tag{31}$$

$$\mathcal{L}^{spatial}_{i \to j}(x) = \left\| p_{i \to j}(x) - f_{i \to j}(x) \right\|_2, \tag{11}$$

$$\widehat{u}_f(x_k) := \begin{cases} g(\overline{x}_k), & x_k \in \partial\Omega_\epsilon \\ \widehat{u}_f(x_{k+1}) + |B(x_k)| f(y_k) G(x_k, y_k), & \text{otherwise.} \end{cases} \tag{8}$$

$$f_x(S_n(\boldsymbol{y}), \boldsymbol{y}) = -\frac{1}{(n+1)!} \langle \boldsymbol{x}^{(n+1)}, \otimes^{n+1} \boldsymbol{y} \rangle + o(|\boldsymbol{y}|^{n+1}),$$

$$\phi_H(\nabla u) = \begin{cases} \dfrac{1}{2\alpha} (\nabla u)^2, & |\nabla u| \leq \alpha \\ |\nabla u| - \dfrac{\alpha}{2}, & |\nabla u| > \alpha \end{cases}, \tag{16}$$

$$\begin{aligned} \mathrm{Var}[\boldsymbol{x}^\star(\boldsymbol{y})] &= \left(\frac{\partial^2 f}{\partial \boldsymbol{x}^2}\right)^{-1} \cdot \frac{\partial^2 f}{\partial \boldsymbol{x} \partial \boldsymbol{y}} \cdot \mathrm{Var}[\boldsymbol{y}] \cdot \frac{\partial^2 f}{\partial \boldsymbol{y} \partial \boldsymbol{x}} \cdot \left(\frac{\partial^2 f}{\partial \boldsymbol{x}^2}\right)^{-1} \\ &= \sigma^2 \left(\frac{\partial^2 f}{\partial \boldsymbol{x}^2}\right)^{-1} \cdot \frac{\partial^2 f}{\partial \boldsymbol{x} \partial \boldsymbol{y}} \cdot \frac{\partial^2 f}{\partial \boldsymbol{y} \partial \boldsymbol{x}} \cdot \left(\frac{\partial^2 f}{\partial \boldsymbol{x}^2}\right)^{-1}, \end{aligned} \tag{56}$$

$$\Psi(d) = \begin{cases} 1, & d = 0 \\ 1/n, & d > 0 \text{ and } d \leq h. \\ 0, & d > h \end{cases} \tag{32}$$

$$\log(\alpha_t) = \lambda^{-1} \int_{V(t)} w(\mathbf{r}) \log(\alpha_{\mathbf{r}})^2 \,\mathrm{d}\mathbf{r},$$

$$W_{\mathrm{cloth}}(\lambda_1, \lambda_2) = \begin{cases} 0 & \lambda_1 < 1, \lambda_2 < 1 \\ W_{\mathrm{StVK}}(\lambda_1, \bar{\lambda}_2(\lambda_1)) & \lambda_1 \geq 1, \lambda_2 < \bar{\lambda}_2(\lambda_1) \\ W_{\mathrm{StVK}}(\lambda_1, \lambda_2) & \text{otherwise}. \end{cases}$$

$$\Gamma_{a,b}(z) := \left\{ S(a,b) + z\boldsymbol{n}(a,b) : z \in \left[-\frac{h(a,b)}{2}, \frac{h(a,b)}{2}\right] \right\}.$$

$$\mathcal{L}\left(I(\vec{x}), \hat{I}(\vec{x})\right) = \tfrac{1}{2} \sum_{\vec{x}} \left(I(\vec{x}) - \hat{I}(\vec{x})\right)^2. \tag{16}$$

$$F(x) := \begin{pmatrix} \varphi_1(x) \\ \vdots \\ \varphi_m(x) \end{pmatrix}, \quad F'(x) := \begin{pmatrix} \nabla\varphi_1(x)^T \\ \vdots \\ \nabla\varphi_m(x)^T \end{pmatrix}. \tag{11}$$

$$\phi_H(\nabla u) = \begin{cases} \dfrac{1}{2\alpha} (\nabla u)^2, & |\nabla u| \leq \alpha \\ |\nabla u| - \dfrac{\alpha}{2}, & |\nabla u| > \alpha \end{cases},$$

$$\begin{aligned} C(P) = {} & \sum_{ij} f_{\mathrm{acc}}(e_{ij}) + \sum_{ijk} f_{\mathrm{cont}}(e_{ij}, e_{jk}) \\ & + \omega_{\mathrm{cv}} \sum_{ijkl} f_{\mathrm{cv}}(e_{ij}, e_{jk}, e_{kl}) + \sum_{ij} f_{\mathrm{simp}}(e_{ij}) \end{aligned} \tag{5}$$

$$c_i(x) = D_i(x) K_i^{-1} \tilde{x}, \tag{8}$$

$$p(y_d = 1 | x(b,k)), \hat{\alpha}(x(b,k)) = D(x(b,k)),$$

$$\kappa_2(x, x') = \exp\left(-\frac{1}{2} \sum_{j=1}^K \frac{1}{\sigma_j^2} (x_j - x'_j)^2\right), \tag{4}$$

$$E(\Phi) = \int_{\mathcal{B}} \|\mathrm{J}\|_{\mathrm{F}}^2 \,\mathrm{d}A_{\mathcal{B}} + \int_{\mathcal{A}} \|\mathrm{J}^{-1}\|_{\mathrm{F}}^2 \,\mathrm{d}A_{\mathcal{A}} \tag{8}$$

$$= \sum_{\tau \in \mathcal{T}} \|\mathrm{J}(\tau)\|_{\mathrm{F}}^2 \,\mathrm{d}A_{\mathcal{B}}(\tau) + \|\mathrm{J}^{-1}(\tau)\|_{\mathrm{F}}^2 \,\mathrm{d}A_{\mathcal{A}}(\tau) \tag{9}$$

$$\begin{aligned} E(\mathbf{X}) &= \sum_{i=1}^d \sum_{j=0}^{N-1} \lambda_j \sum_{m=0}^K \sum_v \gamma_{ij}(t_m, v) \omega_{ij} \\ &= \sum_{m=0}^K \sum_v \sum_{j=0}^{N-1} \lambda_j \sum_{i=1}^d \gamma_{ij}(t_m, v) \omega_{ij}, \end{aligned} \tag{14}$$

$$\mathrm{E}[\langle I \rangle_{\mathrm{SMIS}}] = \mathrm{E}\big[\langle I \rangle_{\mathrm{SMIS}}\big]_{(t_1,x_1),\ldots,(t_n,x_n)} \tag{33a}$$

$$= \mathrm{E}\big[\mathrm{E}[\langle I \rangle_{\mathrm{SMIS}}]_{x_1,\ldots,x_n}\big]_{t_1,\ldots,t_n} \tag{33b}$$

$$= \mathrm{E}\left[\mathrm{E}\left[\sum_{i=1}^n \dot{w}(x_i, t_i) \frac{f(x_i)}{p(x_i|t_i)}\right]_{x_1,\ldots,x_n}\right]_{t_1,\ldots,t_n} \tag{33c}$$

$$= \mathrm{E}\Bigg[\underbrace{\sum_{i=1}^n \int_{\mathcal{X}} \dot{w}(x, t_i) \frac{f(x)}{\cancel{p(x|t_i)}} \cancel{p(x|t_i)} \,\mathrm{d}x}_{= \sum_{i=1}^n I_{t_i} = I;\ \text{see Eq. (2)}}\Bigg]_{t_1,\ldots,t_n} = I. \tag{33d}$$

$$I_\Omega(\boldsymbol{r}) = \begin{cases} 1, & \boldsymbol{r} \in \Omega, \\ 1/2, & \boldsymbol{r} \in \Sigma, \\ 0, & \boldsymbol{r} \in \Gamma \setminus (\Omega \cup \Sigma), \end{cases} \tag{9}$$

$$b(\alpha, \beta) := \begin{cases} \sum\limits_{f=1}^n \|\mathbf{e}_{f\alpha\beta}\| \left(1 + \sum\limits_{i \in \alpha\cap\beta} \|\mathbf{x}_{fi} - \tilde{\mathbf{x}}_i\|\right) & \text{if } q_\alpha \neq q_\beta, \\ 0 & \text{otherwise,} \end{cases} \tag{3}$$

$$c^{im}(\mathbf{v}^q, \mathbf{v}^h, \omega^q_\psi, \omega^h) = ||\mathbf{v}^q - \mathbf{v}^h||^2 + ||\omega^q_\psi - \omega^h||^2. \tag{5}$$

$$\begin{aligned} \mathcal{L}_{\mathrm{feat}}(\theta) &= \sum_d \lambda_d \left\| \Phi_d\left(I^*\right) - \Phi_d\left(f\left(I_{in}; \theta\right)\right) \right\|_1 \\ \mathcal{L}_{\mathrm{pix}}(\theta) &= \left\| I^* - f(I_{in}; \theta) \right\|_1 \\ \mathcal{L}(\theta) &= 0.01 \times \mathcal{L}_{\mathrm{feat}}(\theta) + \mathcal{L}_{\mathrm{pix}}(\theta), \end{aligned} \tag{9}$$

$$\begin{aligned} C_d(x) &= \frac{1}{V_d(1)} \int_{-\infty}^x Ru(t) \,\mathrm{d}t \\ &= \frac{K_{d-1}}{K_d} \int_{-1}^x \left(1 - t^2\right)^{\frac{d-1}{2}} \mathrm{d}t. \end{aligned}$$

$$VTV[h] = \sup_{\phi \in C^1_c, \forall_x \|\phi(x)\|_F \leq 1} \left(\sum_{i=1}^m \int_\Omega h_i \nabla \cdot \phi_i \right),$$

$$E^k(s) \equiv \frac{\int_{\omega_k/\sqrt{2}}^{\omega_k\sqrt{2}} |\mathcal{F}\{S\}(s,\omega)|^2 \,d\omega}{\omega_k(\sqrt{2} - 1/\sqrt{2})} \text{ and } S^k(s) \equiv 10 \log_{10} E^k(s). \tag{20}$$

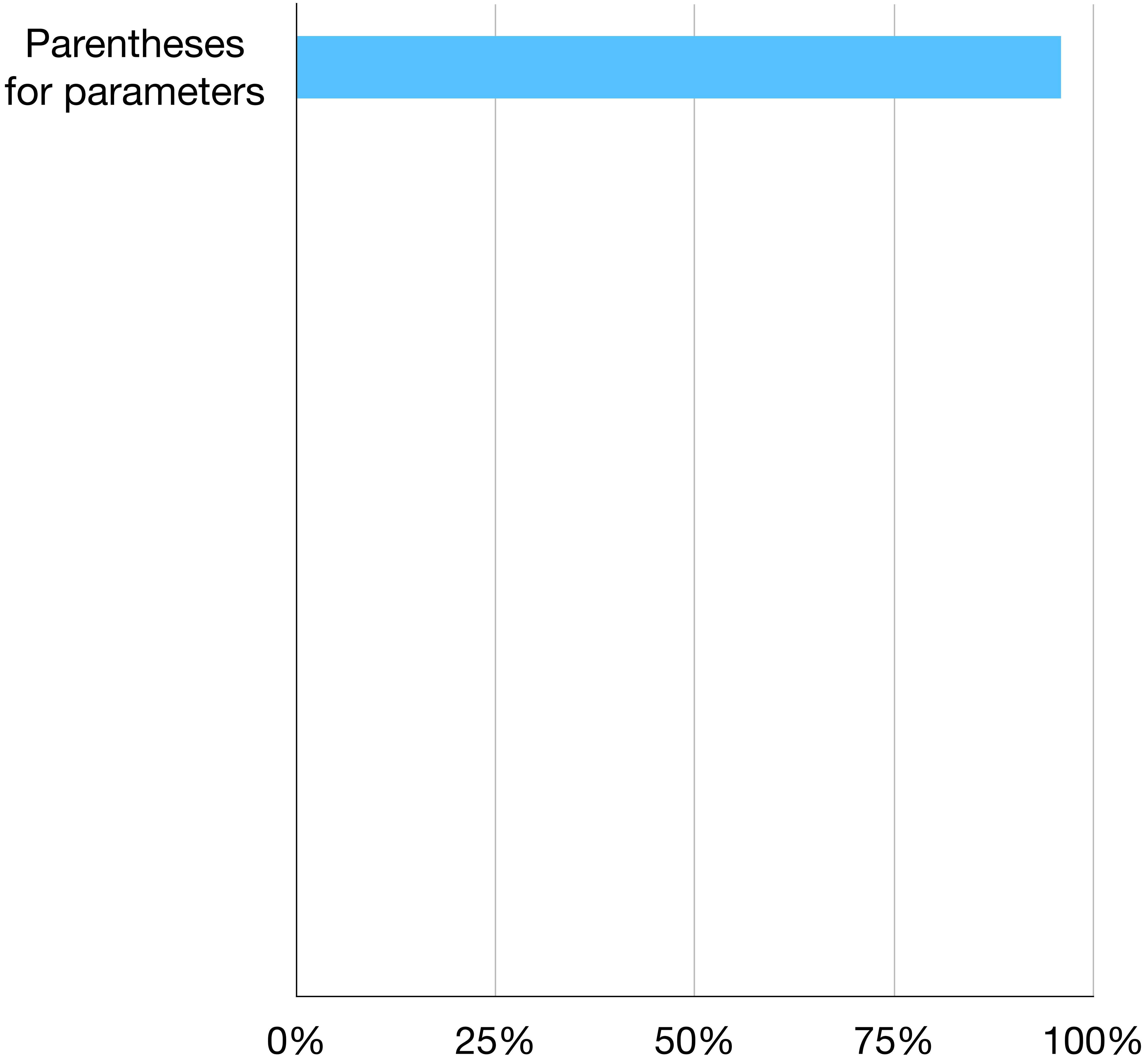
$$\begin{gathered} W_f^{(\cdot)}(x) = \max_{\eta \subset H_f^{(\cdot)}} K_\eta(x), \\ K_\eta(x) = \alpha e^{\left(-\left(\frac{(\theta - \theta_\eta^e)^2}{\left(\sigma_\eta^\theta\right)^2} + \frac{(\phi - \phi_\eta^e)^2}{\left(\sigma_\eta^\phi\right)^2}\right)\right)}, \end{gathered} \tag{4}$$

$$D_{iso}(\theta) = \begin{cases} a_1\theta + b_1 & \theta_0 = 0 \leq \theta \leq \theta_1 \\ a_2\theta + b_2 & \theta_1 \leq \theta \leq \theta_2 \\ \ldots \\ a_n\theta + b_n & \theta_{n-1} \leq \theta \leq \theta_n = \frac{\pi}{2}, \end{cases} \tag{11}$$

$$t(j) = \begin{cases} p_k, & \text{if } \exists k : j = \Delta_\Sigma(k) \\ t_{[\Psi_k, \Psi_{k+1}]}(\psi(j)), & \text{else } \exists k : \Delta_\Sigma(k) < j < \Delta_\Sigma(k+1) \end{cases} \tag{14}$$

Analysis of all 916 function definitions at SIGGRAPH 2020

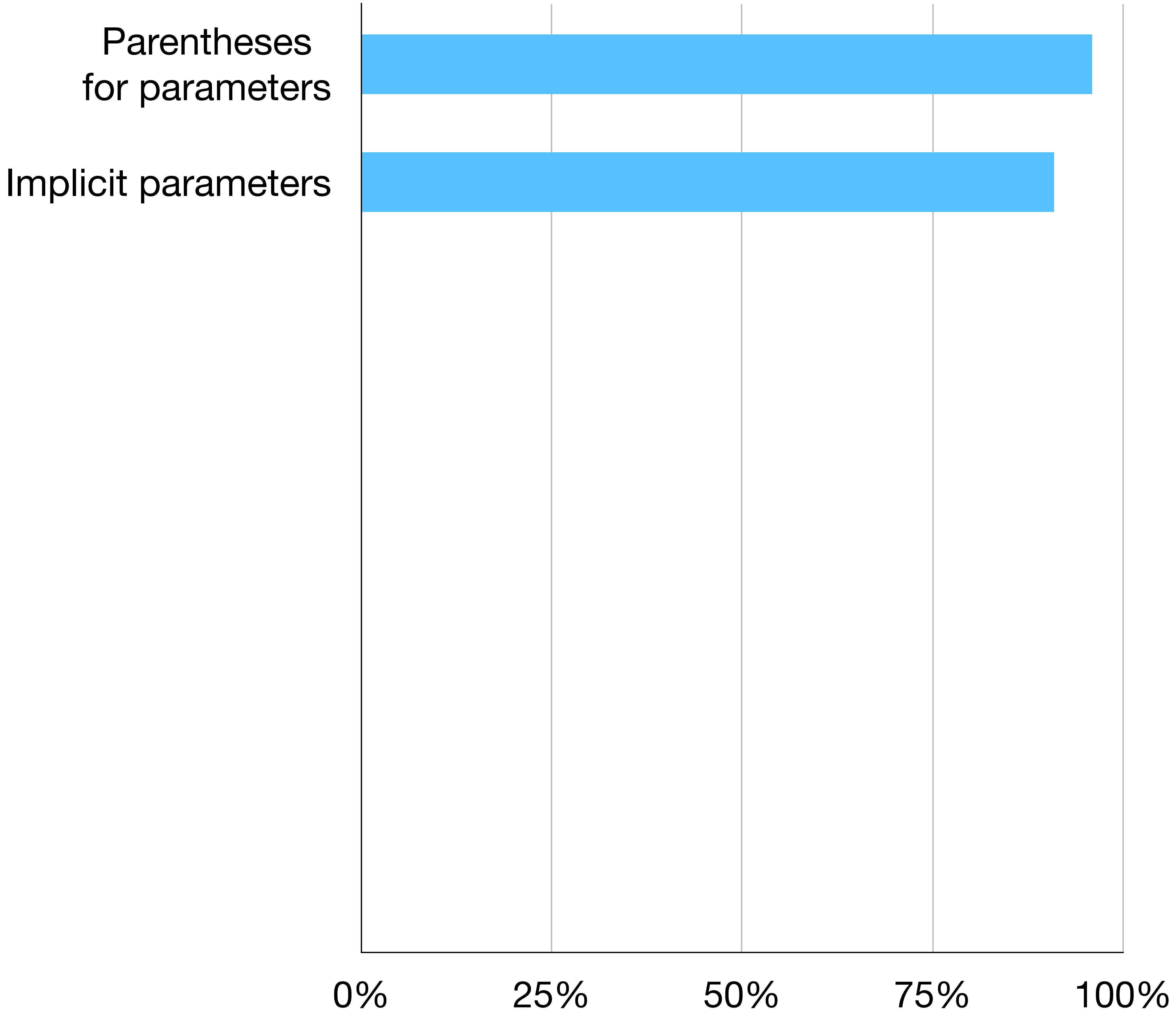
Analysis of all 916 function definitions at SIGGRAPH 2020



$$L(\alpha) = \coth\alpha - \frac{1}{\alpha}$$

[Ni et al. 2020]

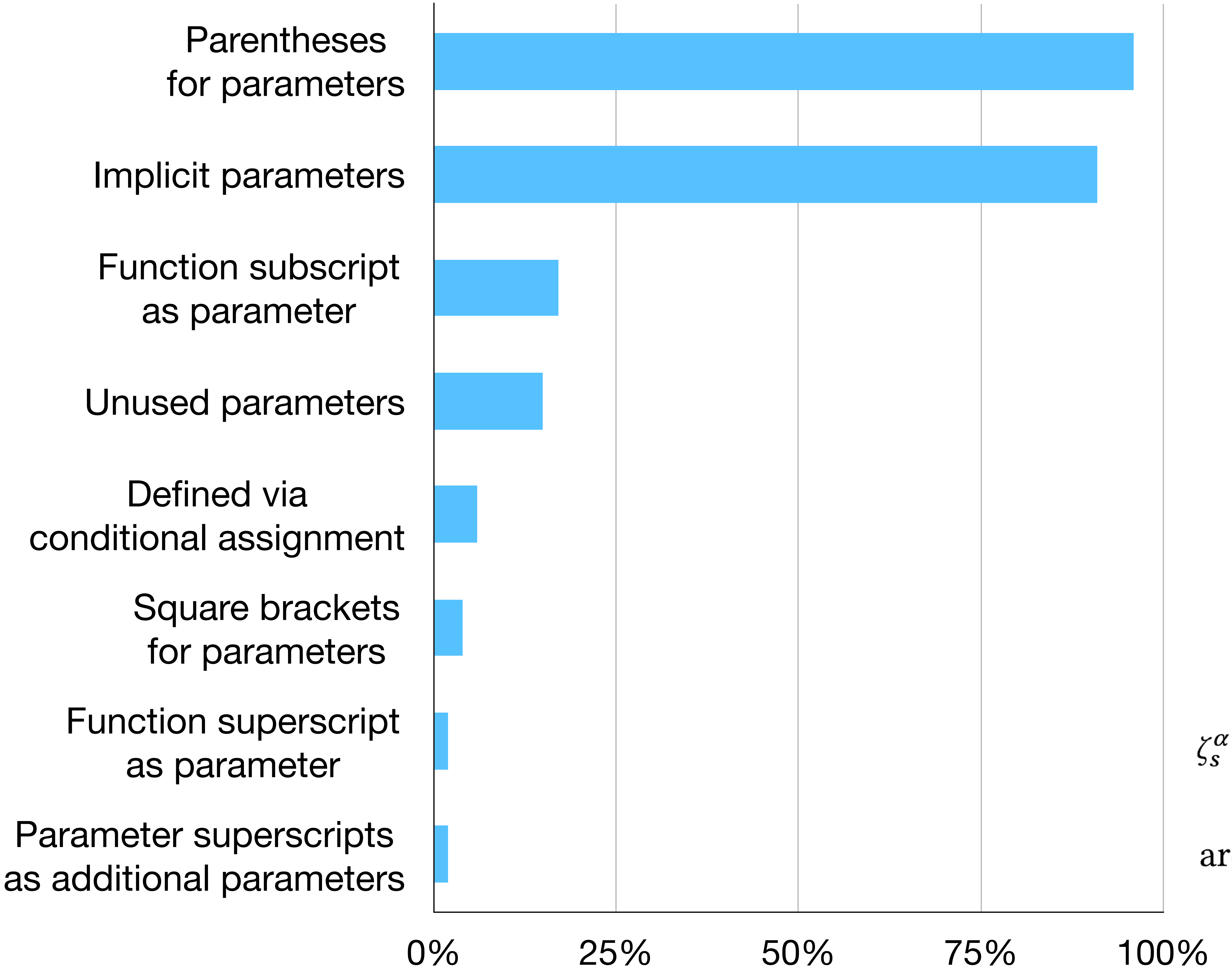
Analysis of all 916 function definitions at SIGGRAPH 2020



$$L(\alpha) = \coth \alpha - \frac{1}{\alpha}$$ [Ni et al. 2020]

$$E(\mathbf{u}) = \frac{1}{2h^2} \left\| \mathbf{M}^{\frac{1}{2}} (\mathbf{u} - \mathbf{u}^*) \right\|^2 + \sum W(\mathbf{u})$$ [Liu et al. 2020]

Analysis of all 916 function definitions at SIGGRAPH 2020



$$L(\alpha) = \coth \alpha - \frac{1}{\alpha}$$ [Ni et al. 2020]

$$E(\mathbf{u}) = \frac{1}{2h^2} \left\| \mathbf{M}^{\frac{1}{2}} (\mathbf{u} - \mathbf{u}^*) \right\|^2 + \sum W(\mathbf{u})$$ [Liu et al. 2020]

$$\varphi_p(x) = \frac{p}{2} (x^2 + \epsilon)^{\frac{p}{2} - 1}$$ [Lan et al. 2020]

$$S_{SR}(x, y) = \frac{\sum_{i \in \mathcal{N}} w_i \cdot S_i}{\sum_{i \in \mathcal{N}} w_i}$$ [Ma et al. 2020]

$$W(r, h)_{\text{cubic}} = \begin{cases} \frac{2}{3} - r^2 + \frac{1}{2} r^3, & 0 \leq r \leq 1, \\ \frac{1}{6} (2 - r)^3, & 1 \leq r \leq 2, \\ 0, & r > 2. \end{cases}$$ [Kim et al. 2020]

$$E[L_{\mathbf{x}_p}] = \frac{2}{(n-1)\sqrt{\pi}} \frac{\Gamma\left(\frac{n}{2}\right)}{\Gamma\left(\frac{n-1}{2}\right)} = \frac{2}{n\sqrt{\pi}} \frac{\Gamma\left(\frac{n+2}{2}\right)}{\Gamma\left(\frac{n+1}{2}\right)}$$ [Chiu et al. 2020]

$$\zeta_s^{\alpha}(x) \equiv \frac{2^j \, \alpha^{-1}}{(2\pi)^{3/2}} \sum_n \beta_{j,n}^t \, \zeta_s^{\alpha,n}(x) = \int_{\mathbb{R}_u} \psi_s\left(S_\alpha(x,u)^T\right) du$$ [Lessig 2020]

$$\text{area}(f^\delta) = \text{area}(f)(1 - 2\delta H(f) + \delta^2 K(f))$$ [Jiang et al. 2020]

Formative Study

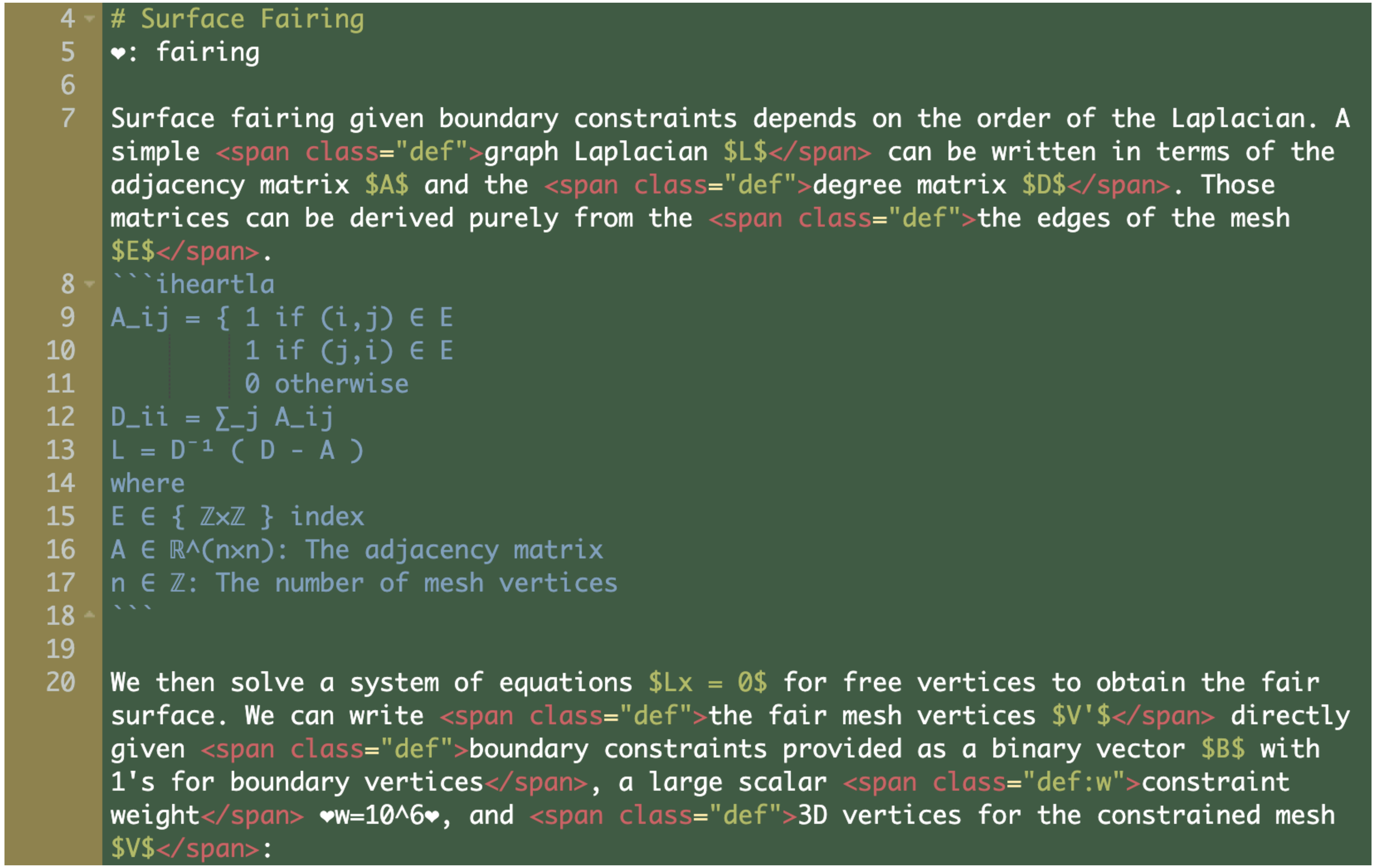
- All appear to be written using LaTeX.
- Observations:
  - I. Prose organizes the document, interleaved with math.
  - II. Math appears out of order. Symbols used before defined.
  - III. Symbols re-used in different contexts.
  - IV. Symbol appears in executable formulas and non-executable derivations.
  - V. Symbols and functions appear with conditional assignment.
  - VI. Functions have a variety of implied semantics for parameters and pre-computed symbols.

Formative Study

- All appear to be written using LaTeX.
- Observations:
  - I. Prose organizes the document, interleaved with math.
  - II. Math appears out of order. Symbols used before defined.
  - III. Symbols re-used in different contexts.
  - IV. Symbol appears in executable formulas and non-executable derivations.
  - V. Symbols and functions appear with conditional assignment.
  - VI. Functions have a variety of implied semantics for parameters and pre-computed symbols.
- Pseudocode sometimes present, compilable code isn’t. No literate programs.

H❤️rtDown Design: Authoring

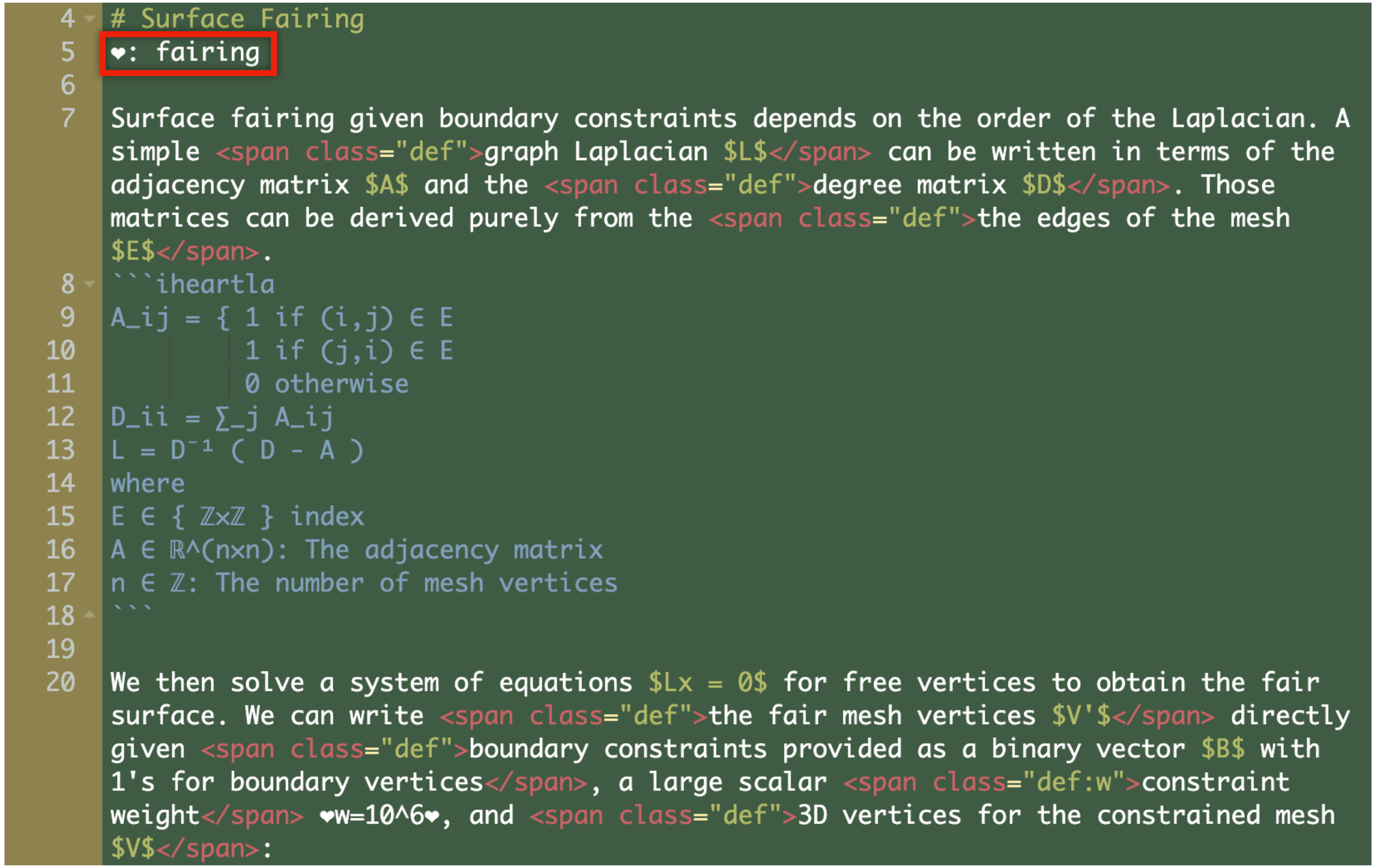
- Context definition



~~~
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A
simple <span class="def">graph Laplacian $L$</span> can be written in terms of the
adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those
matrices can be derived purely from the <span class="def">the edges of the mesh
$E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤxℤ } index
A ∈ ℝ^(nxn): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
```

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair
surface. We can write <span class="def">the fair mesh vertices $V'$</span> directly
given <span class="def">boundary constraints provided as a binary vector $B$ with
1's for boundary vertices</span>, a large scalar <span class="def:w">constraint
weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh
$V$</span>:
~~~

H❤️rtDown Design: Authoring

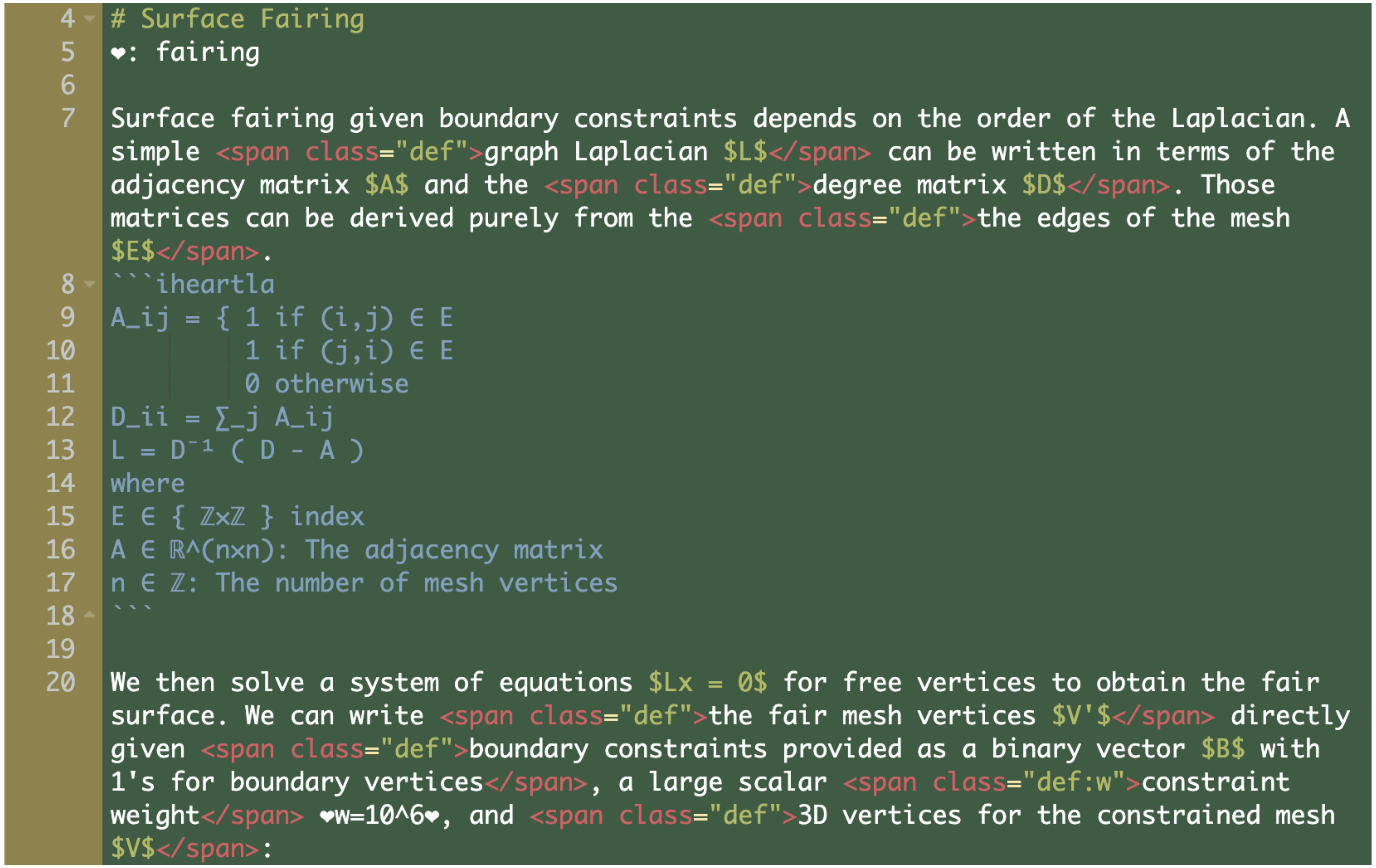
- Context definition



~~~
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A
simple <span class="def">graph Laplacian $L$</span> can be written in terms of the
adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those
matrices can be derived purely from the <span class="def">the edges of the mesh
$E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤ×ℤ } index
A ∈ ℝ^(n×n): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
```


We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair
surface. We can write <span class="def">the fair mesh vertices $V'$</span> directly
given <span class="def">boundary constraints provided as a binary vector $B$ with
1's for boundary vertices</span>, a large scalar <span class="def:w">constraint
weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh
$V$</span>:
~~~

H❤️rtDown Design: Authoring

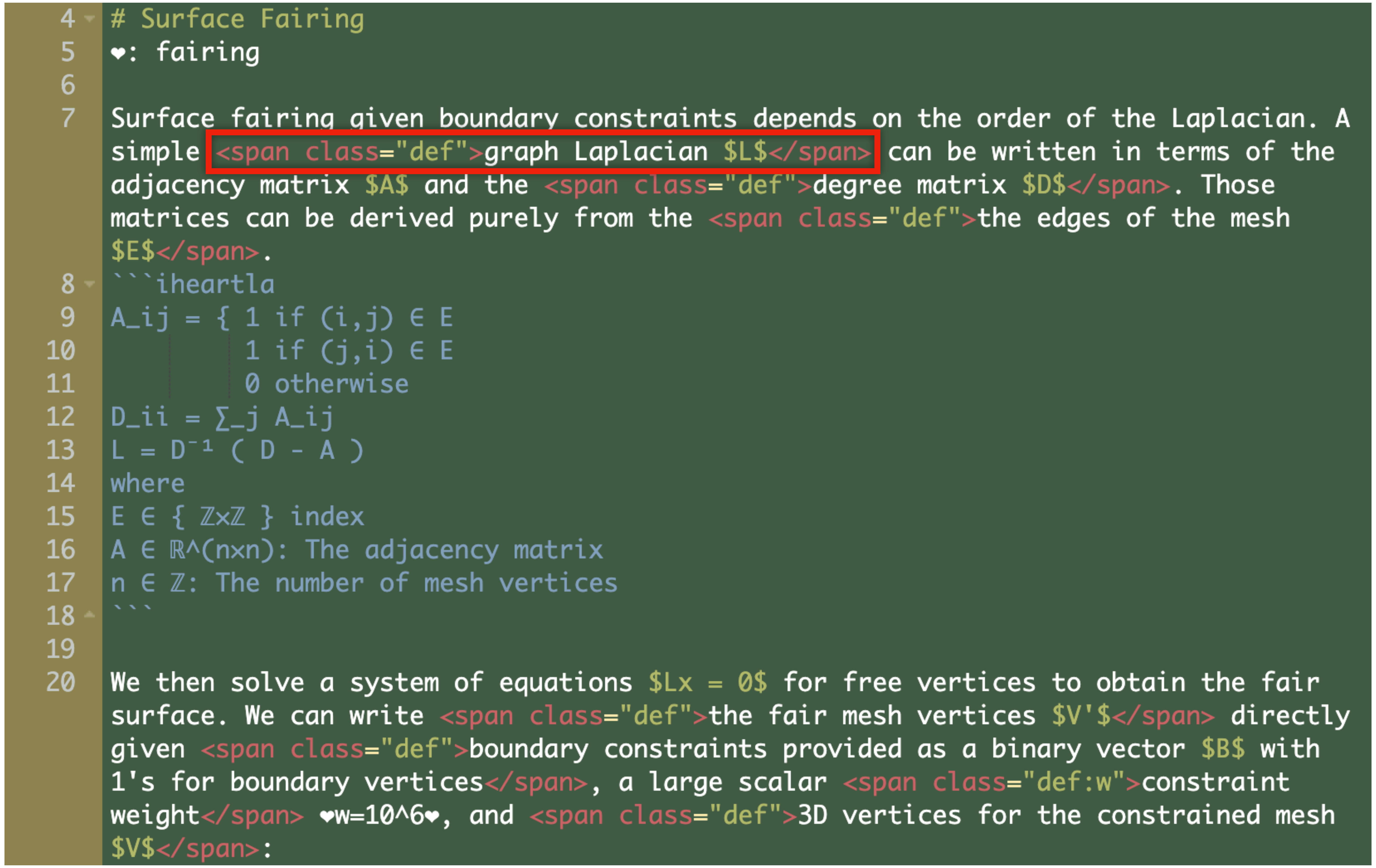
- Prose descriptions



~~~
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A
simple <span class="def">graph Laplacian $L$</span> can be written in terms of the
adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those
matrices can be derived purely from the <span class="def">the edges of the mesh
$E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤ×ℤ } index
A ∈ ℝ^(nxn): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
```


We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair
surface. We can write <span class="def">the fair mesh vertices $V'$</span> directly
given <span class="def">boundary constraints provided as a binary vector $B$ with
1's for boundary vertices</span>, a large scalar <span class="def:w">constraint
weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh
$V$</span>:
~~~

H❤️rtDown Design: Authoring

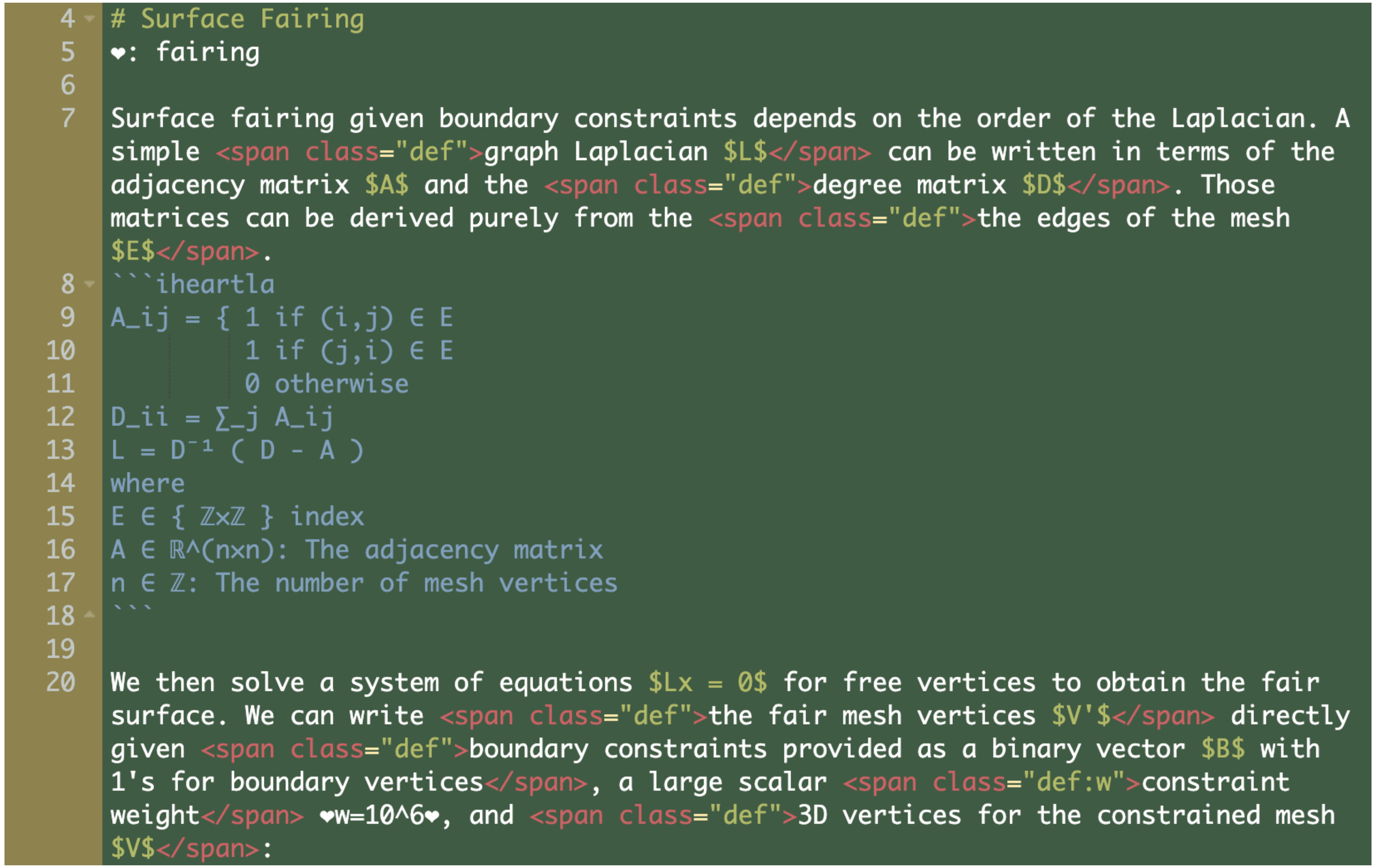
- Prose descriptions



~~~
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those matrices can be derived purely from the <span class="def">the edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤ×ℤ } index
A ∈ ℝ^(n×n): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
```


We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write <span class="def">the fair mesh vertices $V'$</span> directly given <span class="def">boundary constraints provided as a binary vector $B$ with 1's for boundary vertices</span>, a large scalar <span class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh $V$</span>:
~~~

H❤️rtDown Design: Authoring

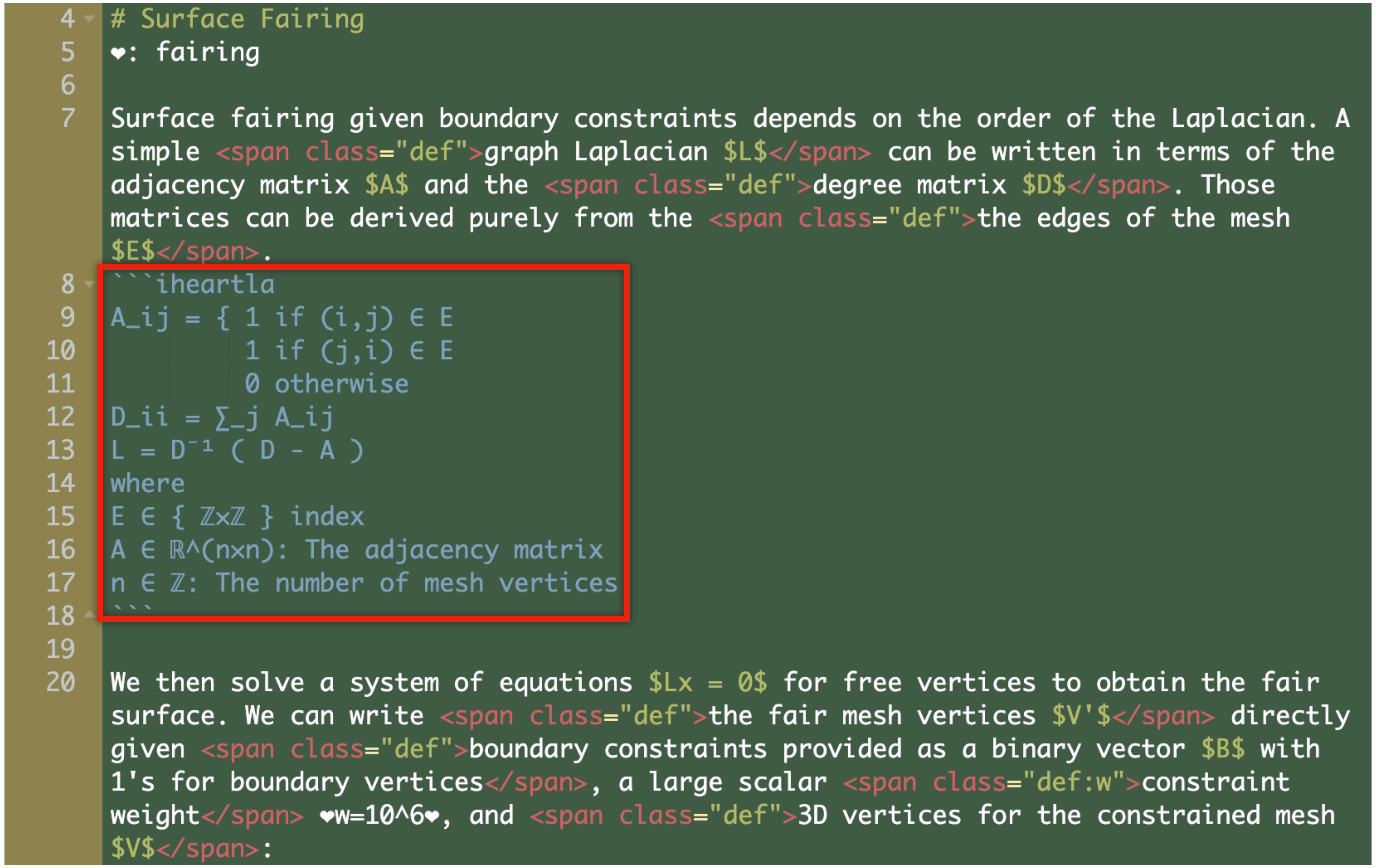
- Executable mathematical expressions



~~~
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those matrices can be derived purely from the <span class="def">the edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤ×ℤ } index
A ∈ ℝ^(n×n): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
```


We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write <span class="def">the fair mesh vertices $V'$</span> directly given <span class="def">boundary constraints provided as a binary vector $B$ with 1's for boundary vertices</span>, a large scalar <span class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh $V$</span>:
~~~

H❤️rtDown Design: Authoring

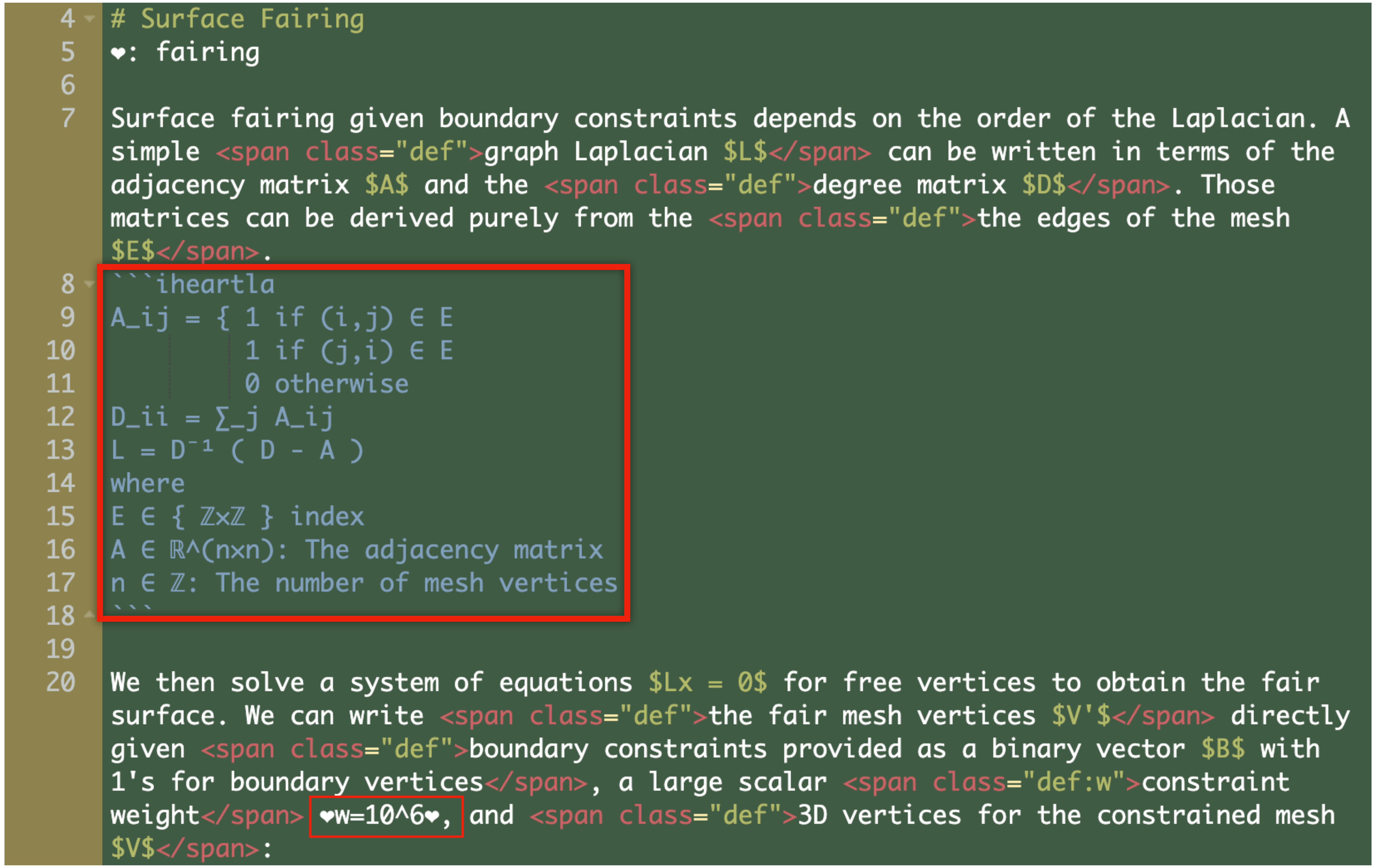
- Executable mathematical expressions



~~~
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A
simple <span class="def">graph Laplacian $L$</span> can be written in terms of the
adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those
matrices can be derived purely from the <span class="def">the edges of the mesh
$E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤ×ℤ } index
A ∈ ℝ^(n×n): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
```


We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair
surface. We can write <span class="def">the fair mesh vertices $V'$</span> directly
given <span class="def">boundary constraints provided as a binary vector $B$ with
1's for boundary vertices</span>, a large scalar <span class="def:w">constraint
weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh
$V$</span>:
~~~

H❤️rtDown Design: Authoring

- Executable mathematical expressions



~~~~
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A
simple <span class="def">graph Laplacian $L$</span> can be written in terms of the
adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those
matrices can be derived purely from the <span class="def">the edges of the mesh
$E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤxℤ } index
A ∈ ℝ^(nxn): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
```


We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair
surface. We can write <span class="def">the fair mesh vertices $V'$</span> directly
given <span class="def">boundary constraints provided as a binary vector $B$ with
1's for boundary vertices</span>, a large scalar <span class="def:w">constraint
weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh
$V$</span>:
~~~~

H❤️rtDown Design: Authoring

- I❤️LA extensions

H❤️rtDown Design: Authoring

- I❤️LA extensions
    - Local function support

H❤️rtDown Design: Authoring

- I❤️LA extensions
    - Local function support
    - Symbol def-use analysis

H❤️rtDown Design: Authoring

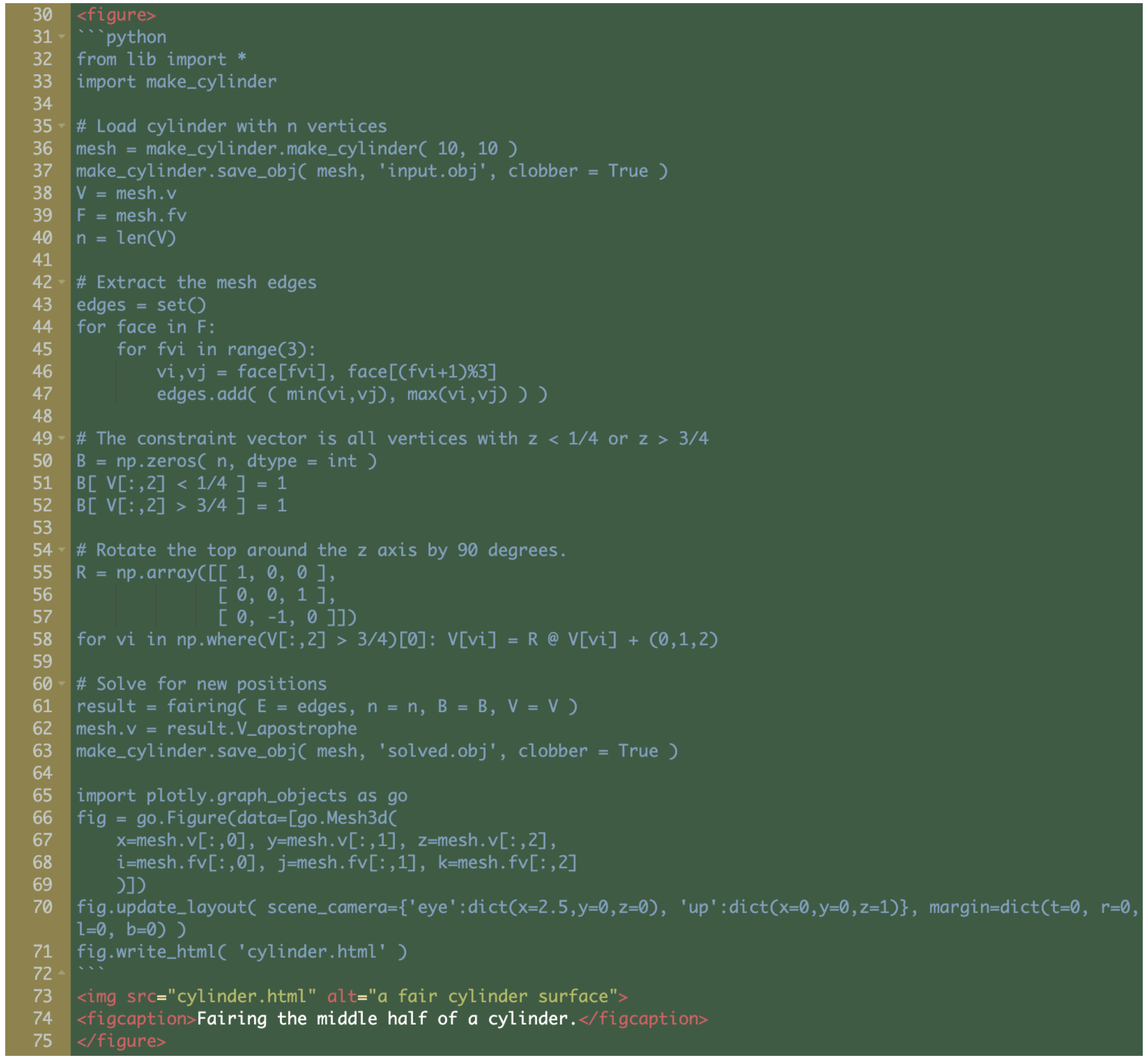
- I❤️LA extensions
  - Local function support
  - Symbol def-use analysis
  - Modules

H❤️rtDown Design: Authoring

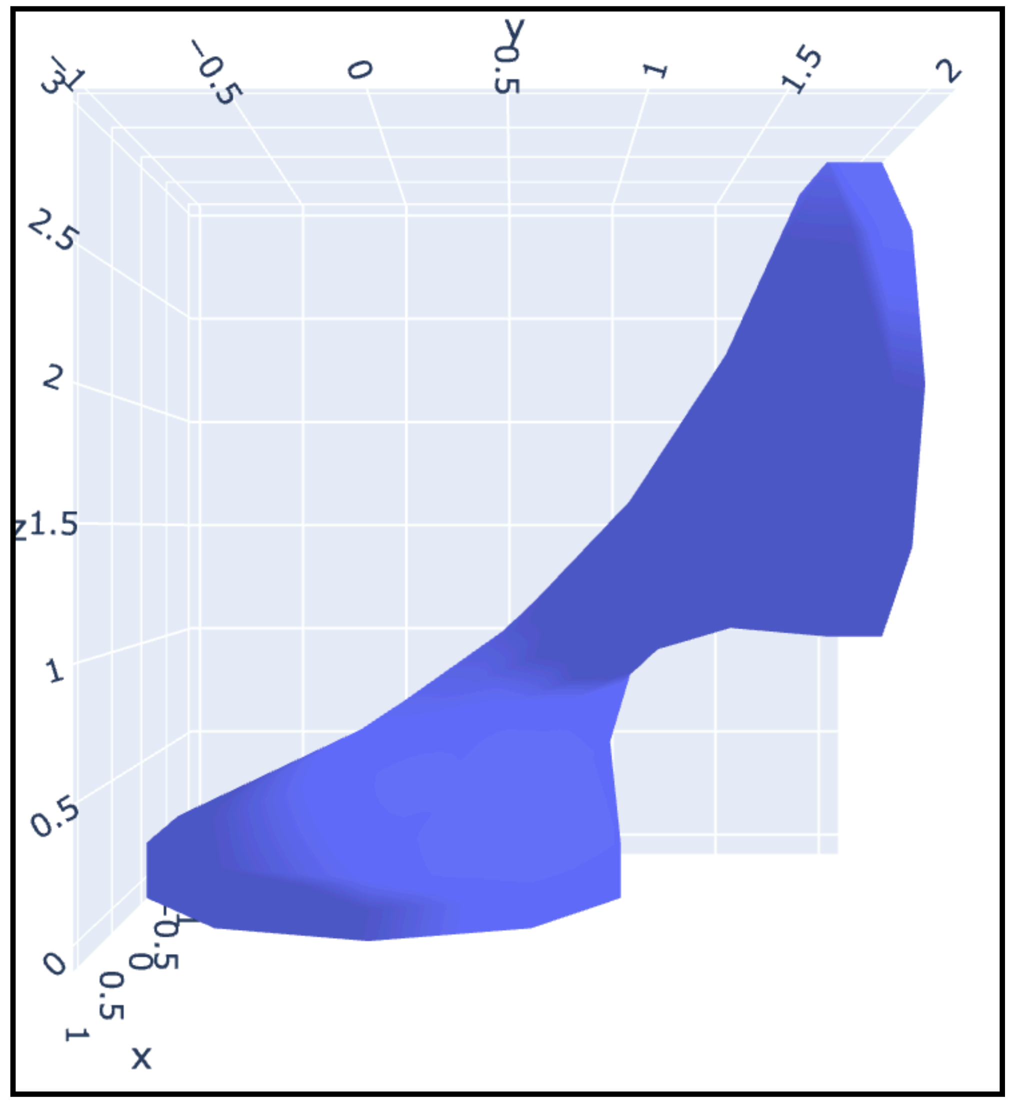
- I❤️LA extensions
  - Local function support
  - Symbol def-use analysis
  - Modules
  - MathJax output includes metadata

H❤️rtDown Design: Authoring

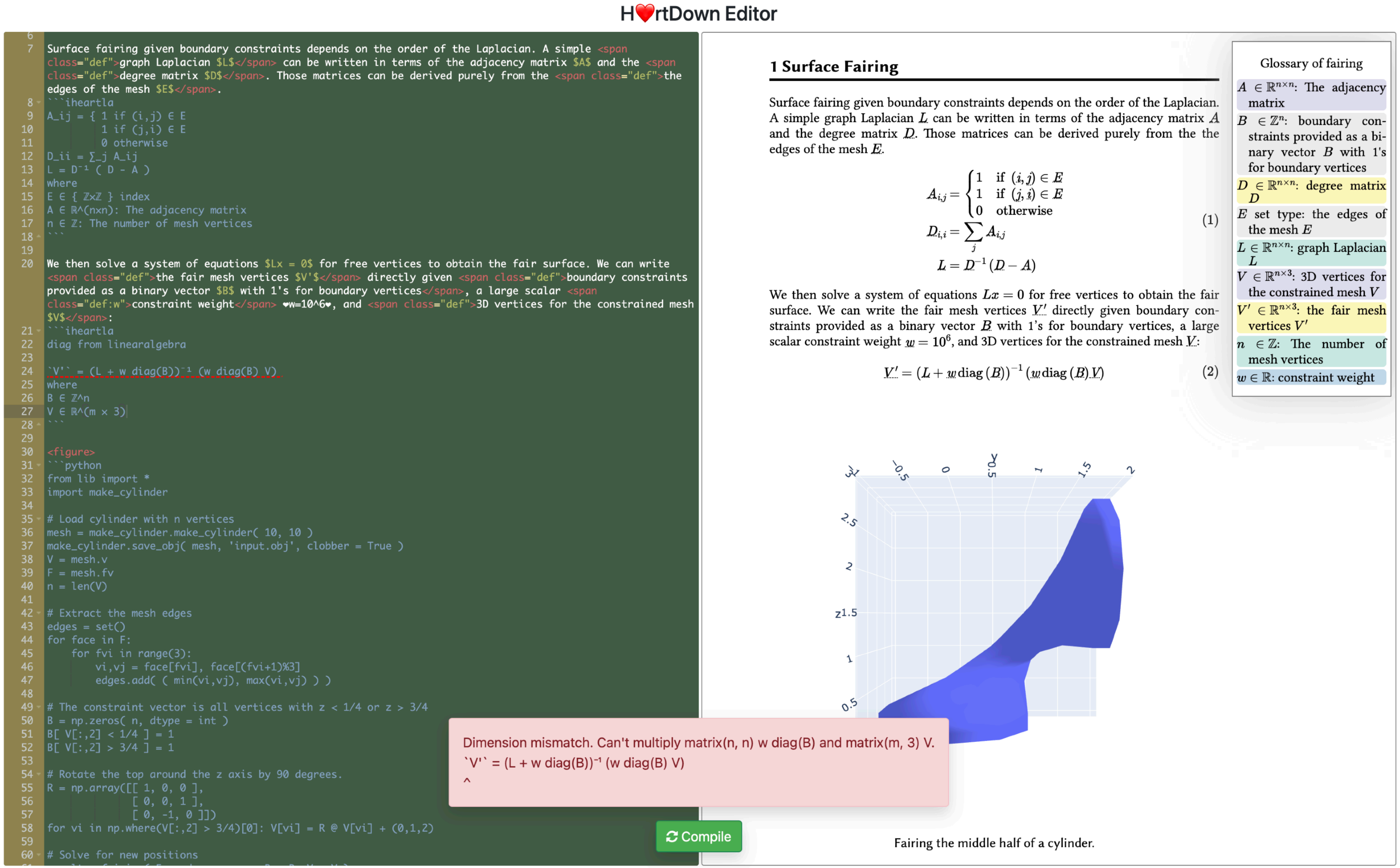
- Figures



~~~
<figure>
```python
from lib import *
import make_cylinder

# Load cylinder with n vertices
mesh = make_cylinder.make_cylinder( 10, 10 )
make_cylinder.save_obj( mesh, 'input.obj', clobber = True )
V = mesh.v
F = mesh.fv
n = len(V)

# Extract the mesh edges
edges = set()
for face in F:
    for fvi in range(3):
        vi,vj = face[fvi], face[(fvi+1)%3]
        edges.add( ( min(vi,vj), max(vi,vj) ) )

# The constraint vector is all vertices with z < 1/4 or z > 3/4
B = np.zeros( n, dtype = int )
B[ V[:,2] < 1/4 ] = 1
B[ V[:,2] > 3/4 ] = 1

# Rotate the top around the z axis by 90 degrees.
R = np.array([[ 1, 0, 0 ],
              [ 0, 0, 1 ],
              [ 0, -1, 0 ]])
for vi in np.where(V[:,2] > 3/4)[0]: V[vi] = R @ V[vi] + (0,1,2)

# Solve for new positions
result = fairing( E = edges, n = n, B = B, V = V )
mesh.v = result.V_apostrophe
make_cylinder.save_obj( mesh, 'solved.obj', clobber = True )

import plotly.graph_objects as go
fig = go.Figure(data=[go.Mesh3d(
    x=mesh.v[:,0], y=mesh.v[:,1], z=mesh.v[:,2],
    i=mesh.fv[:,0], j=mesh.fv[:,1], k=mesh.fv[:,2]
    )])
fig.update_layout( scene_camera={'eye':dict(x=2.5,y=0,z=0), 'up':dict(x=0,y=0,z=1)}, margin=dict(t=0, r=0, l=0, b=0) )
fig.write_html( 'cylinder.html' )
```
<img src="cylinder.html" alt="a fair cylinder surface">
<figcaption>Fairing the middle half of a cylinder.</figcaption>
</figure>
~~~



H❤rtDown Design: Author support



H❤rtDown Editor

~~~
Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span
class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the <span
class="def">degree matrix $D$</span>. Those matrices can be derived purely from the <span class="def">the
edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤ×ℤ } index
A ∈ ℝ^(nxn): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
~~~

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write
<span class="def">the fair mesh vertices V'</span> directly given <span class="def">boundary constraints
provided as a binary vector B with 1's for boundary vertices</span>, a large scalar <span
class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh
V</span>:
```iheartla
diag from linearalgebra

`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
where
B ∈ ℤ^n
V ∈ ℝ^(m × 3)
```

<figure>
```python
from lib import *
import make_cylinder

# Load cylinder with n vertices
mesh = make_cylinder.make_cylinder( 10, 10 )
make_cylinder.save_obj( mesh, 'input.obj', clobber = True )
V = mesh.v
F = mesh.fv
n = len(V)

# Extract the mesh edges
edges = set()
for face in F:
    for fvi in range(3):
        vi,vj = face[fvi], face[(fvi+1)%3]
        edges.add( ( min(vi,vj), max(vi,vj) ) )

# The constraint vector is all vertices with z < 1/4 or z > 3/4
B = np.zeros( n, dtype = int )
B[ V[:,2] < 1/4 ] = 1
B[ V[:,2] > 3/4 ] = 1

# Rotate the top around the z axis by 90 degrees.
R = np.array([[ 1, 0, 0 ],
              [ 0, 0, 1 ],
              [ 0, -1, 0 ]])
for vi in np.where(V[:,2] > 3/4)[0]: V[vi] = R @ V[vi] + (0,1,2)

# Solve for new positions
```

1 Surface Fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple graph Laplacian L can be written in terms of the adjacency matrix A and the degree matrix D. Those matrices can be derived purely from the the edges of the mesh E.

$$\begin{aligned} A_{i,j} &= \begin{cases} 1 & \text{if } (i,j) \in E \\ 1 & \text{if } (j,i) \in E \\ 0 & \text{otherwise} \end{cases} \\ D_{i,i} &= \sum_j A_{i,j} \\ L &= D^{-1}(D - A) \end{aligned} \tag{1}$$

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write the fair mesh vertices V' directly given boundary constraints provided as a binary vector B with 1's for boundary vertices, a large scalar constraint weight $w = 10^6$, and 3D vertices for the constrained mesh V:

$$V' = (L + w\,\mathrm{diag}(B))^{-1}(w\,\mathrm{diag}(B)V) \tag{2}$$

Glossary of fairing

$A \in \mathbb{R}^{n\times n}$: The adjacency matrix

$B \in \mathbb{Z}^{n}$: boundary constraints provided as a binary vector B with 1's for boundary vertices

$D \in \mathbb{R}^{n\times n}$: degree matrix D

E set type: the edges of the mesh E

$L \in \mathbb{R}^{n\times n}$: graph Laplacian L

$V \in \mathbb{R}^{n\times 3}$: 3D vertices for the constrained mesh V

$V' \in \mathbb{R}^{n\times 3}$: the fair mesh vertices V'

$n \in \mathbb{Z}$: The number of mesh vertices

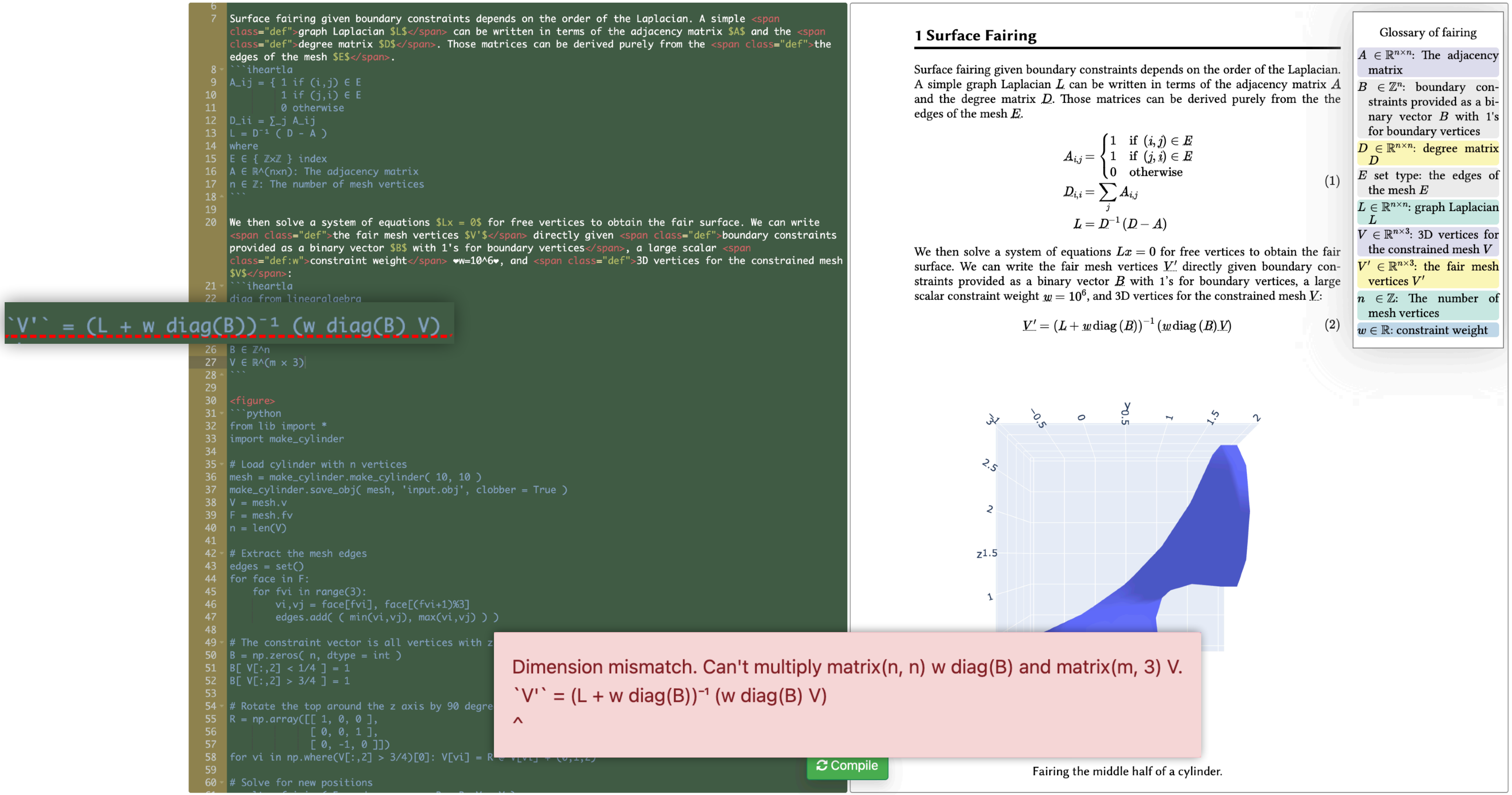
$w \in \mathbb{R}$: constraint weight

Fairing the middle half of a cylinder.

Dimension mismatch. Can't multiply matrix(n, n) w diag(B) and matrix(m, 3) V.
`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
^

H❤️rtDown Design: Author support

H❤️rtDown Editor



~~~
Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the <span class="def">degree matrix $D$</span>. Those matrices can be derived purely from the <span class="def">the edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤxℤ } index
A ∈ ℝ^(nxn): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
~~~

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write <span class="def">the fair mesh vertices V'</span> directly given <span class="def">boundary constraints provided as a binary vector B with 1's for boundary vertices</span>, a large scalar <span class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh V</span>:
```iheartla
diag from linearalgebra
`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
B ∈ ℤ^n
V ∈ ℝ^(m x 3)
```

<figure>
```python
from lib import *
import make_cylinder

# Load cylinder with n vertices
mesh = make_cylinder.make_cylinder( 10, 10 )
make_cylinder.save_obj( mesh, 'input.obj', clobber = True )
V = mesh.v
F = mesh.fv
n = len(V)

# Extract the mesh edges
edges = set()
for face in F:
    for fvi in range(3):
        vi,vj = face[fvi], face[(fvi+1)%3]
        edges.add( ( min(vi,vj), max(vi,vj) ) )

# The constraint vector is all vertices with z
B = np.zeros( n, dtype = int )
B[ V[:,2] < 1/4 ] = 1
B[ V[:,2] > 3/4 ] = 1

# Rotate the top around the z axis by 90 degre
R = np.array([[ 1, 0, 0 ],
              [ 0, 0, 1 ],
              [ 0, -1, 0 ]])
for vi in np.where(V[:,2] > 3/4)[0]: V[vi] = R

# Solve for new positions
```

Dimension mismatch. Can't multiply matrix(n, n) w diag(B) and matrix(m, 3) V.

```
`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
^
```

1 Surface Fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple graph Laplacian L can be written in terms of the adjacency matrix A and the degree matrix D. Those matrices can be derived purely from the the edges of the mesh E.

$$\begin{aligned} A_{i,j} &= \begin{cases} 1 & \text{if } (i,j) \in E \\ 1 & \text{if } (j,i) \in E \\ 0 & \text{otherwise} \end{cases} \\ D_{i,i} &= \sum_j A_{i,j} \\ L &= D^{-1}(D - A) \end{aligned} \tag{1}$$

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write the fair mesh vertices V' directly given boundary constraints provided as a binary vector B with 1's for boundary vertices, a large scalar constraint weight $w = 10^6$, and 3D vertices for the constrained mesh V:

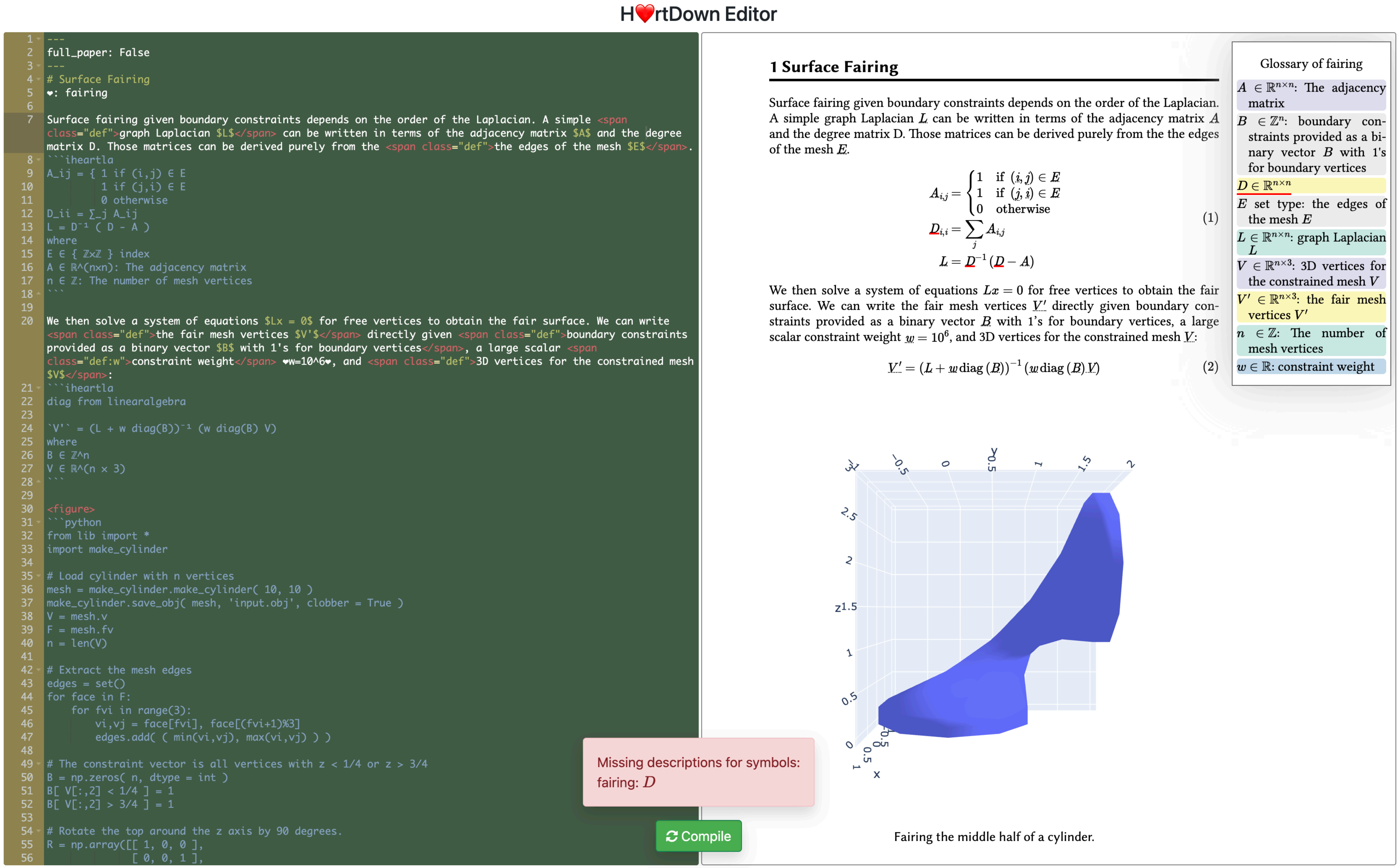
$$V' = (L + w\,\mathrm{diag}(B))^{-1}(w\,\mathrm{diag}(B)V) \tag{2}$$

Glossary of fairing

- $A \in \mathbb{R}^{n\times n}$: The adjacency matrix
- $B \in \mathbb{Z}^n$: boundary constraints provided as a binary vector B with 1's for boundary vertices
- $D \in \mathbb{R}^{n\times n}$: degree matrix D
- E set type: the edges of the mesh E
- $L \in \mathbb{R}^{n\times n}$: graph Laplacian L
- $V \in \mathbb{R}^{n\times 3}$: 3D vertices for the constrained mesh V
- $V' \in \mathbb{R}^{n\times 3}$: the fair mesh vertices V'
- $n \in \mathbb{Z}$: The number of mesh vertices
- $w \in \mathbb{R}$: constraint weight

Fairing the middle half of a cylinder.

H❤️rtDown Design: Author support



H❤️rtDown Editor

~~~
---
full_paper: False
---
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the degree matrix D. Those matrices can be derived purely from the <span class="def">the edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤxℤ } index
A ∈ ℝ^(nxn): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
```

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write <span class="def">the fair mesh vertices $V'$</span> directly given <span class="def">boundary constraints provided as a binary vector $B$ with 1's for boundary vertices</span>, a large scalar <span class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh $V$</span>:
```iheartla
diag from linearalgebra

`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
where
B ∈ ℤ^n
V ∈ ℝ^(n x 3)
```


<figure>
```python
from lib import *
import make_cylinder

# Load cylinder with n vertices
mesh = make_cylinder.make_cylinder( 10, 10 )
make_cylinder.save_obj( mesh, 'input.obj', clobber = True )
V = mesh.v
F = mesh.fv
n = len(V)

# Extract the mesh edges
edges = set()
for face in F:
    for fvi in range(3):
        vi,vj = face[fvi], face[(fvi+1)%3]
        edges.add( ( min(vi,vj), max(vi,vj) ) )

# The constraint vector is all vertices with z < 1/4 or z > 3/4
B = np.zeros( n, dtype = int )
B[ V[:,2] < 1/4 ] = 1
B[ V[:,2] > 3/4 ] = 1

# Rotate the top around the z axis by 90 degrees.
R = np.array([[ 1, 0, 0 ],
              [ 0, 0, 1 ],
~~~

Missing descriptions for symbols:
fairing: D

Compile

1 Surface Fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple graph Laplacian L can be written in terms of the adjacency matrix A and the degree matrix D. Those matrices can be derived purely from the the edges of the mesh E.

$$A_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 1 & \text{if } (j,i) \in E \\ 0 & \text{otherwise} \end{cases}$$
$$D_{i,i} = \sum_j A_{i,j}$$
$$L = D^{-1}(D - A) \tag{1}$$

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write the fair mesh vertices V' directly given boundary constraints provided as a binary vector B with 1's for boundary vertices, a large scalar constraint weight $w = 10^6$, and 3D vertices for the constrained mesh V:

$$V' = (L + w\,\mathrm{diag}(B))^{-1}(w\,\mathrm{diag}(B)V) \tag{2}$$

Glossary of fairing

$A \in \mathbb{R}^{n\times n}$: The adjacency matrix

$B \in \mathbb{Z}^{n}$: boundary constraints provided as a binary vector B with 1's for boundary vertices

$D \in \mathbb{R}^{n\times n}$

E set type: the edges of the mesh E

$L \in \mathbb{R}^{n\times n}$: graph Laplacian L

$V \in \mathbb{R}^{n\times 3}$: 3D vertices for the constrained mesh V

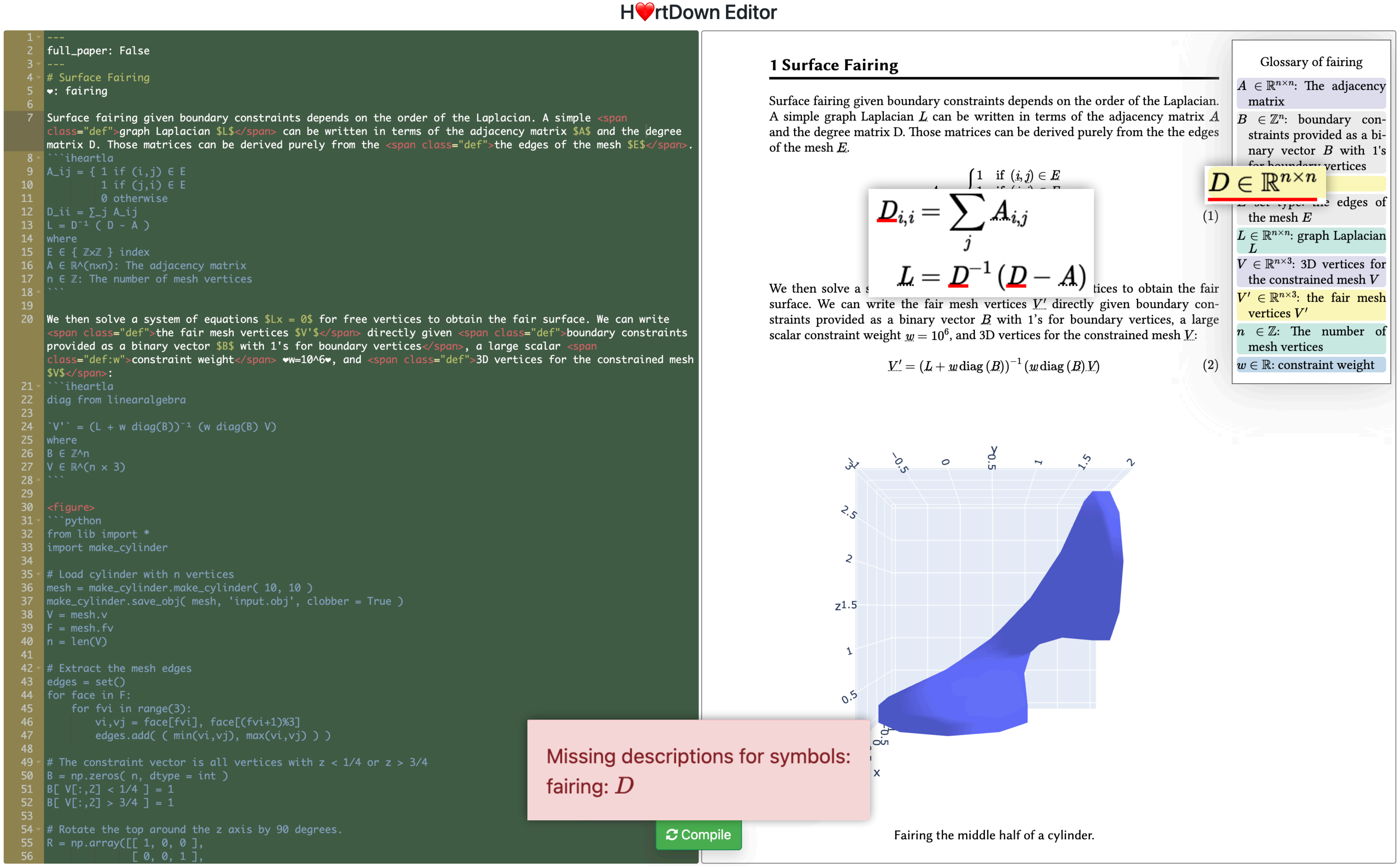
$V' \in \mathbb{R}^{n\times 3}$: the fair mesh vertices V'

$n \in \mathbb{Z}$: The number of mesh vertices

$w \in \mathbb{R}$: constraint weight

Fairing the middle half of a cylinder.

H❤️rtDown Design: Author support



H❤️rtDown Editor

~~~
---
full_paper: False
---
# Surface Fairing
❤: fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple <span
class="def">graph Laplacian $L$</span> can be written in terms of the adjacency matrix $A$ and the degree
matrix D. Those matrices can be derived purely from the <span class="def">the edges of the mesh $E$</span>.
```iheartla
A_ij = { 1 if (i,j) ∈ E
         1 if (j,i) ∈ E
         0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤxℤ } index
A ∈ ℝ^(nxn): The adjacency matrix
n ∈ ℤ: The number of mesh vertices
~~~

We then solve a system of equations $Lx = 0$ for free vertices to obtain the fair surface. We can write
<span class="def">the fair mesh vertices V'</span> directly given <span class="def">boundary constraints
provided as a binary vector B with 1's for boundary vertices</span>, a large scalar <span
class="def:w">constraint weight</span> ❤w=10^6❤, and <span class="def">3D vertices for the constrained mesh
V</span>:
```iheartla
diag from linearalgebra

`V'` = (L + w diag(B))⁻¹ (w diag(B) V)
where
B ∈ ℤ^n
V ∈ ℝ^(n x 3)
```

<figure>
```python
from lib import *
import make_cylinder

# Load cylinder with n vertices
mesh = make_cylinder.make_cylinder( 10, 10 )
make_cylinder.save_obj( mesh, 'input.obj', clobber = True )
V = mesh.v
F = mesh.fv
n = len(V)

# Extract the mesh edges
edges = set()
for face in F:
    for fvi in range(3):
        vi,vj = face[fvi], face[(fvi+1)%3]
        edges.add( ( min(vi,vj), max(vi,vj) ) )

# The constraint vector is all vertices with z < 1/4 or z > 3/4
B = np.zeros( n, dtype = int )
B[ V[:,2] < 1/4 ] = 1
B[ V[:,2] > 3/4 ] = 1

# Rotate the top around the z axis by 90 degrees.
R = np.array([[ 1, 0, 0 ],
              [ 0, 0, 1 ],
```

Missing descriptions for symbols:
fairing: D

1 Surface Fairing

Surface fairing given boundary constraints depends on the order of the Laplacian. A simple graph Laplacian L can be written in terms of the adjacency matrix A and the degree matrix D. Those matrices can be derived purely from the the edges of the mesh E.

$$\begin{cases}1 & \text{if } (i,j) \in E\end{cases}$$

$$D_{i,i} = \sum_j A_{i,j} \qquad (1)$$

$$L = D^{-1}(D - A)$$

We then solve a s tices to obtain the fair surface. We can write the fair mesh vertices V' directly given boundary constraints provided as a binary vector B with 1's for boundary vertices, a large scalar constraint weight $w = 10^6$, and 3D vertices for the constrained mesh V:

$$V' = (L + w\,\mathrm{diag}(B))^{-1}(w\,\mathrm{diag}(B)V) \qquad (2)$$

Fairing the middle half of a cylinder.

Glossary of fairing

$A \in \mathbb{R}^{n\times n}$: The adjacency matrix

$B \in \mathbb{Z}^{n}$: boundary constraints provided as a binary vector B with 1's for boundary vertices

$D \in \mathbb{R}^{n\times n}$

the edges of the mesh E

$L \in \mathbb{R}^{n\times n}$: graph Laplacian L

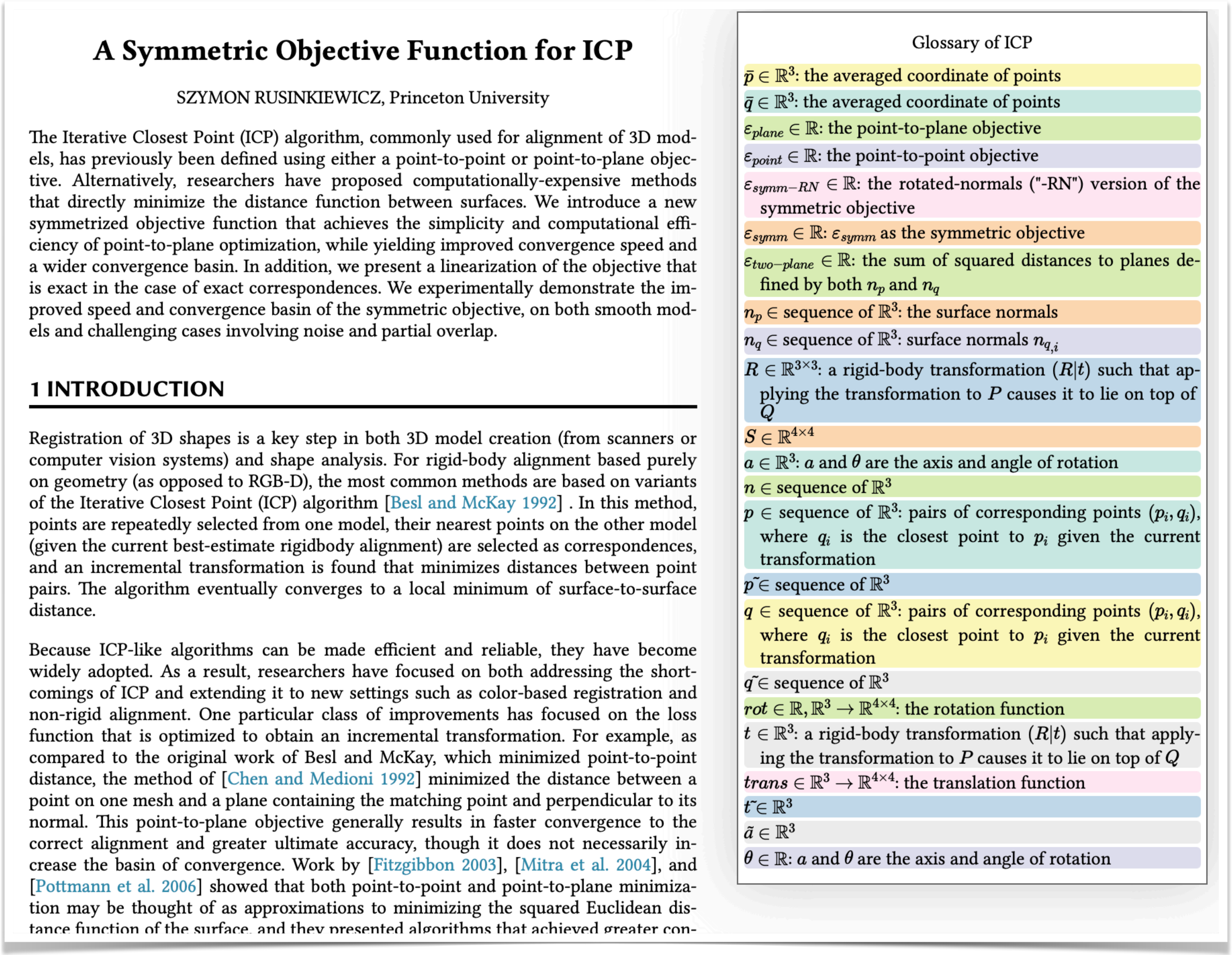
$V \in \mathbb{R}^{n\times 3}$: 3D vertices for the constrained mesh V

$V' \in \mathbb{R}^{n\times 3}$: the fair mesh vertices V'

$n \in \mathbb{Z}$: The number of mesh vertices

$w \in \mathbb{R}$: constraint weight

H❤rtDown Design: Reading Environment



A Symmetric Objective Function for ICP

SZYMON RUSINKIEWICZ, Princeton University

The Iterative Closest Point (ICP) algorithm, commonly used for alignment of 3D models, has previously been defined using either a point-to-point or point-to-plane objective. Alternatively, researchers have proposed computationally-expensive methods that directly minimize the distance function between surfaces. We introduce a new symmetrized objective function that achieves the simplicity and computational efficiency of point-to-plane optimization, while yielding improved convergence speed and a wider convergence basin. In addition, we present a linearization of the objective that is exact in the case of exact correspondences. We experimentally demonstrate the improved speed and convergence basin of the symmetric objective, on both smooth models and challenging cases involving noise and partial overlap.

1 INTRODUCTION

Registration of 3D shapes is a key step in both 3D model creation (from scanners or computer vision systems) and shape analysis. For rigid-body alignment based purely on geometry (as opposed to RGB-D), the most common methods are based on variants of the Iterative Closest Point (ICP) algorithm [Besl and McKay 1992] . In this method, points are repeatedly selected from one model, their nearest points on the other model (given the current best-estimate rigidbody alignment) are selected as correspondences, and an incremental transformation is found that minimizes distances between point pairs. The algorithm eventually converges to a local minimum of surface-to-surface distance.

Because ICP-like algorithms can be made efficient and reliable, they have become widely adopted. As a result, researchers have focused on both addressing the shortcomings of ICP and extending it to new settings such as color-based registration and non-rigid alignment. One particular class of improvements has focused on the loss function that is optimized to obtain an incremental transformation. For example, as compared to the original work of Besl and McKay, which minimized point-to-point distance, the method of [Chen and Medioni 1992] minimized the distance between a point on one mesh and a plane containing the matching point and perpendicular to its normal. This point-to-plane objective generally results in faster convergence to the correct alignment and greater ultimate accuracy, though it does not necessarily increase the basin of convergence. Work by [Fitzgibbon 2003], [Mitra et al. 2004], and [Pottmann et al. 2006] showed that both point-to-point and point-to-plane minimization may be thought of as approximations to minimizing the squared Euclidean distance function of the surface, and they presented algorithms that achieved greater con-

Glossary of ICP

$\bar{p} \in \mathbb{R}^3$: the averaged coordinate of points

$\bar{q} \in \mathbb{R}^3$: the averaged coordinate of points

$\varepsilon_{plane} \in \mathbb{R}$: the point-to-plane objective

$\varepsilon_{point} \in \mathbb{R}$: the point-to-point objective

$\varepsilon_{symm-RN} \in \mathbb{R}$: the rotated-normals ("-RN") version of the symmetric objective

$\varepsilon_{symm} \in \mathbb{R}$: ε_{symm} as the symmetric objective

$\varepsilon_{two-plane} \in \mathbb{R}$: the sum of squared distances to planes defined by both n_p and n_q

$n_p \in$ sequence of $\mathbb{R}^3$: the surface normals

$n_q \in$ sequence of $\mathbb{R}^3$: surface normals $n_{q,i}$

$R \in \mathbb{R}^{3\times3}$: a rigid-body transformation $(R|t)$ such that applying the transformation to P causes it to lie on top of Q

$S \in \mathbb{R}^{4\times4}$

$a \in \mathbb{R}^3$: a and θ are the axis and angle of rotation

$n \in$ sequence of $\mathbb{R}^3$

$p \in$ sequence of $\mathbb{R}^3$: pairs of corresponding points (p_i, q_i), where q_i is the closest point to p_i given the current transformation

$\tilde{p} \in$ sequence of $\mathbb{R}^3$

$q \in$ sequence of $\mathbb{R}^3$: pairs of corresponding points (p_i, q_i), where q_i is the closest point to p_i given the current transformation

$\tilde{q} \in$ sequence of $\mathbb{R}^3$

$rot \in \mathbb{R}, \mathbb{R}^3 \rightarrow \mathbb{R}^{4\times4}$: the rotation function

$t \in \mathbb{R}^3$: a rigid-body transformation $(R|t)$ such that applying the transformation to P causes it to lie on top of Q

$trans \in \mathbb{R}^3 \rightarrow \mathbb{R}^{4\times4}$: the translation function

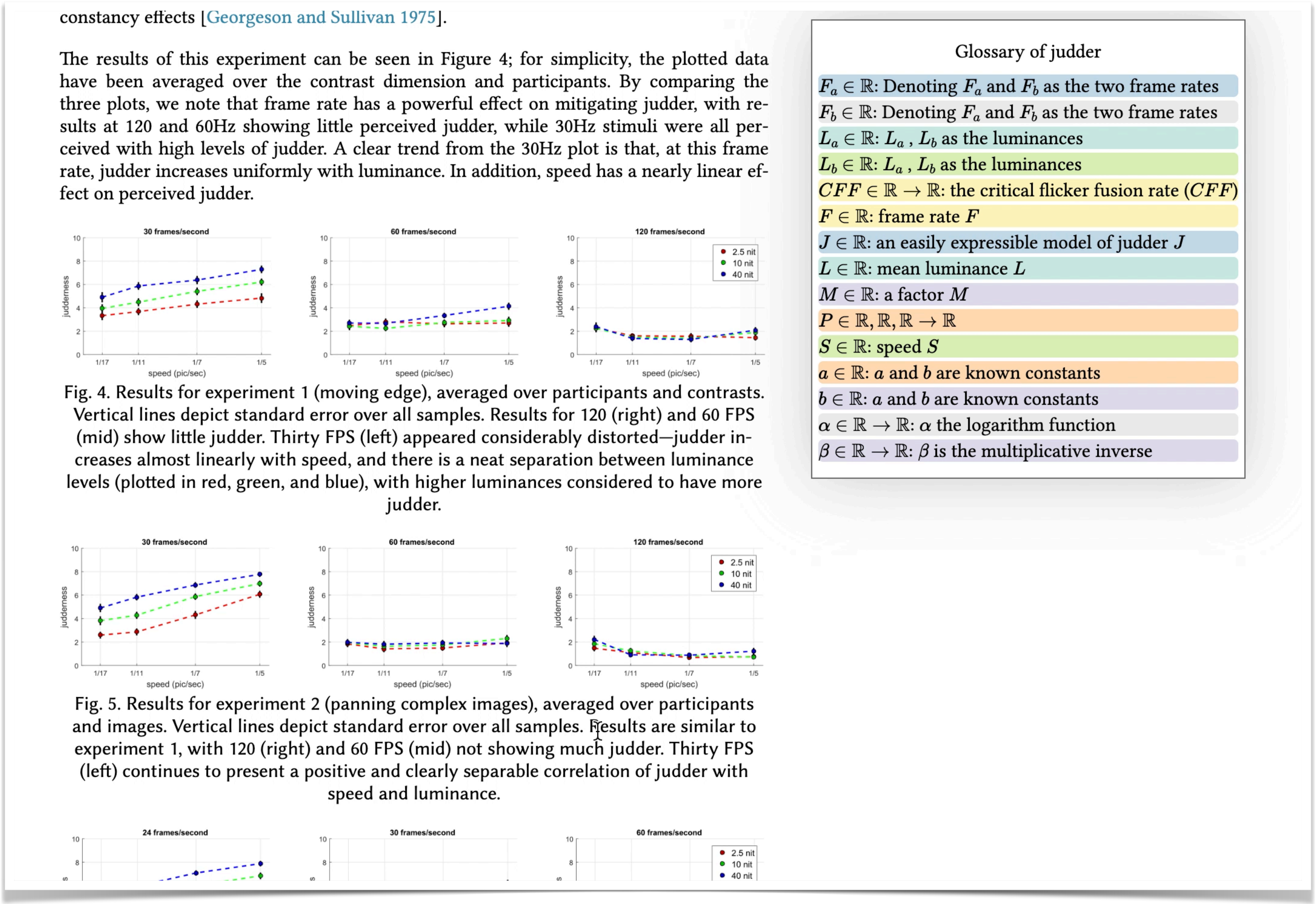
$\tilde{t} \in \mathbb{R}^3$

$\tilde{a} \in \mathbb{R}^3$

$\theta \in \mathbb{R}$: a and θ are the axis and angle of rotation

H❤️rtDown Design: Reading Environment

- Glossary



constancy effects [Georgeson and Sullivan 1975].

The results of this experiment can be seen in Figure 4; for simplicity, the plotted data have been averaged over the contrast dimension and participants. By comparing the three plots, we note that frame rate has a powerful effect on mitigating judder, with results at 120 and 60Hz showing little perceived judder, while 30Hz stimuli were all perceived with high levels of judder. A clear trend from the 30Hz plot is that, at this frame rate, judder increases uniformly with luminance. In addition, speed has a nearly linear effect on perceived judder.

Fig. 4. Results for experiment 1 (moving edge), averaged over participants and contrasts. Vertical lines depict standard error over all samples. Results for 120 (right) and 60 FPS (mid) show little judder. Thirty FPS (left) appeared considerably distorted—judder increases almost linearly with speed, and there is a neat separation between luminance levels (plotted in red, green, and blue), with higher luminances considered to have more judder.

Fig. 5. Results for experiment 2 (panning complex images), averaged over participants and images. Vertical lines depict standard error over all samples. Results are similar to experiment 1, with 120 (right) and 60 FPS (mid) not showing much judder. Thirty FPS (left) continues to present a positive and clearly separable correlation of judder with speed and luminance.

Glossary of judder

$F_a \in \mathbb{R}$: Denoting F_a and F_b as the two frame rates

$F_b \in \mathbb{R}$: Denoting F_a and F_b as the two frame rates

$L_a \in \mathbb{R}$: L_a , L_b as the luminances

$L_b \in \mathbb{R}$: L_a , L_b as the luminances

$CFF \in \mathbb{R} \to \mathbb{R}$: the critical flicker fusion rate (CFF)

$F \in \mathbb{R}$: frame rate F

$J \in \mathbb{R}$: an easily expressible model of judder J

$L \in \mathbb{R}$: mean luminance L

$M \in \mathbb{R}$: a factor M

$P \in \mathbb{R}, \mathbb{R}, \mathbb{R} \to \mathbb{R}$

$S \in \mathbb{R}$: speed S

$a \in \mathbb{R}$: a and b are known constants

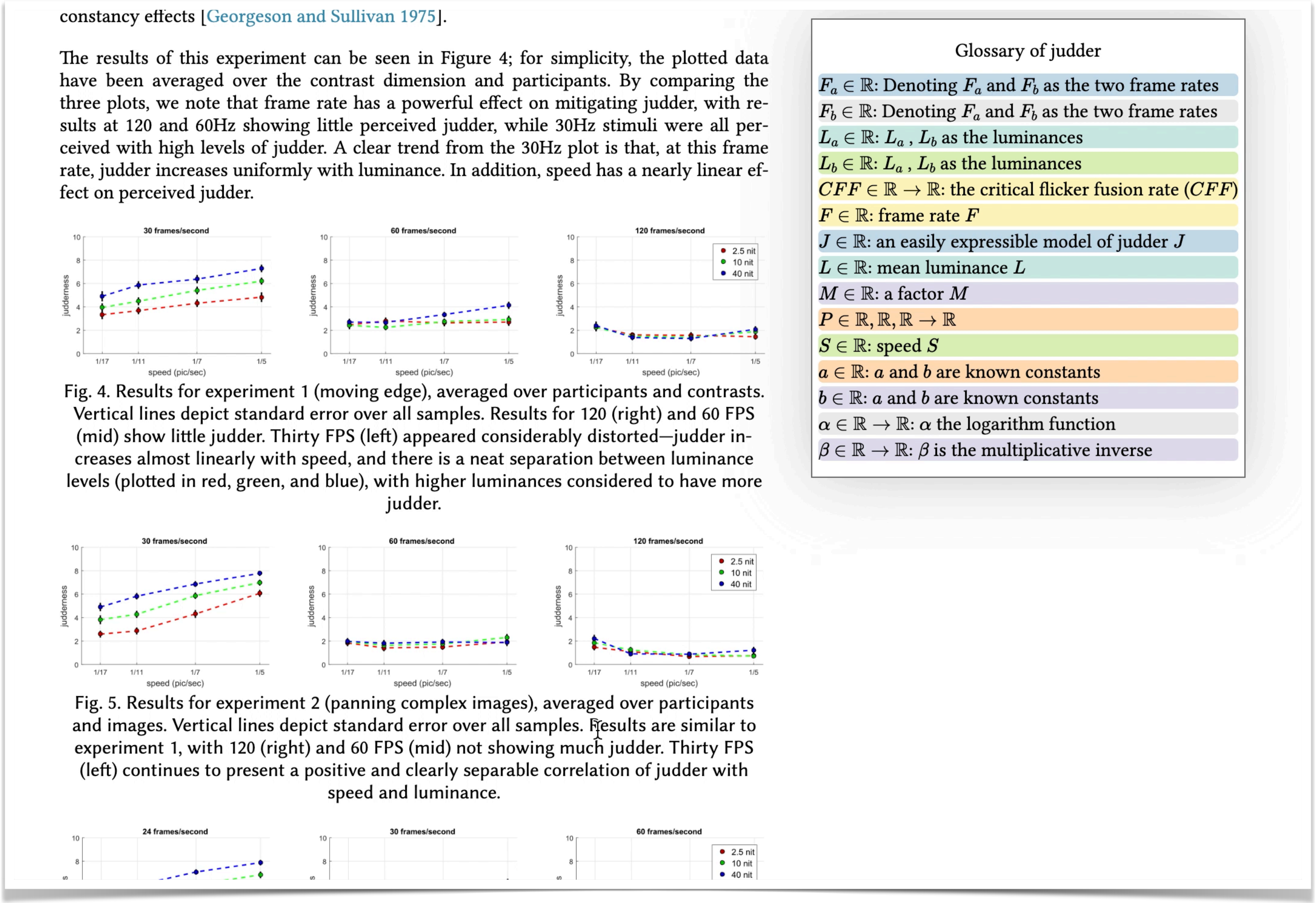
$b \in \mathbb{R}$: a and b are known constants

$\alpha \in \mathbb{R} \to \mathbb{R}$: α the logarithm function

$\beta \in \mathbb{R} \to \mathbb{R}$: β is the multiplicative inverse

H❤️rtDown Design: Reading Environment

- Glossary



constancy effects [Georgeson and Sullivan 1975].

The results of this experiment can be seen in Figure 4; for simplicity, the plotted data have been averaged over the contrast dimension and participants. By comparing the three plots, we note that frame rate has a powerful effect on mitigating judder, with results at 120 and 60Hz showing little perceived judder, while 30Hz stimuli were all perceived with high levels of judder. A clear trend from the 30Hz plot is that, at this frame rate, judder increases uniformly with luminance. In addition, speed has a nearly linear effect on perceived judder.

Fig. 4. Results for experiment 1 (moving edge), averaged over participants and contrasts. Vertical lines depict standard error over all samples. Results for 120 (right) and 60 FPS (mid) show little judder. Thirty FPS (left) appeared considerably distorted—judder increases almost linearly with speed, and there is a neat separation between luminance levels (plotted in red, green, and blue), with higher luminances considered to have more judder.

Fig. 5. Results for experiment 2 (panning complex images), averaged over participants and images. Vertical lines depict standard error over all samples. Results are similar to experiment 1, with 120 (right) and 60 FPS (mid) not showing much judder. Thirty FPS (left) continues to present a positive and clearly separable correlation of judder with speed and luminance.

Glossary of judder

$F_a \in \mathbb{R}$: Denoting F_a and F_b as the two frame rates

$F_b \in \mathbb{R}$: Denoting F_a and F_b as the two frame rates

$L_a \in \mathbb{R}$: L_a , L_b as the luminances

$L_b \in \mathbb{R}$: L_a , L_b as the luminances

$CFF \in \mathbb{R} \to \mathbb{R}$: the critical flicker fusion rate (CFF)

$F \in \mathbb{R}$: frame rate F

$J \in \mathbb{R}$: an easily expressible model of judder J

$L \in \mathbb{R}$: mean luminance L

$M \in \mathbb{R}$: a factor M

$P \in \mathbb{R}, \mathbb{R}, \mathbb{R} \to \mathbb{R}$

$S \in \mathbb{R}$: speed S

$a \in \mathbb{R}$: a and b are known constants

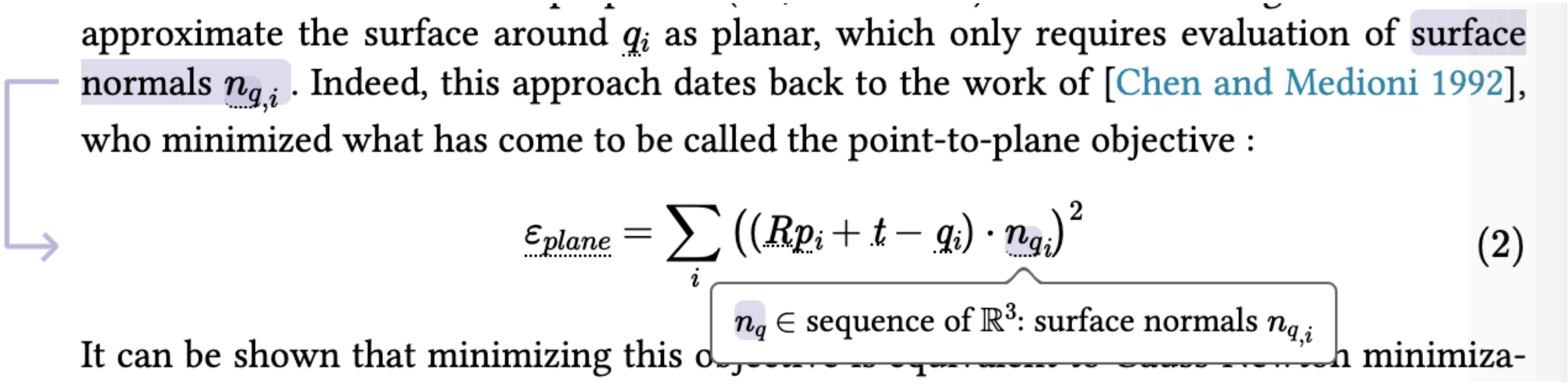
$b \in \mathbb{R}$: a and b are known constants

$\alpha \in \mathbb{R} \to \mathbb{R}$: α the logarithm function

$\beta \in \mathbb{R} \to \mathbb{R}$: β is the multiplicative inverse

H❤️rtDown Design: Reading Environment

- Symbol definitions



approximate the surface around q_i as planar, which only requires evaluation of surface normals $n_{q,i}$. Indeed, this approach dates back to the work of [Chen and Medioni 1992], who minimized what has come to be called the point-to-plane objective :

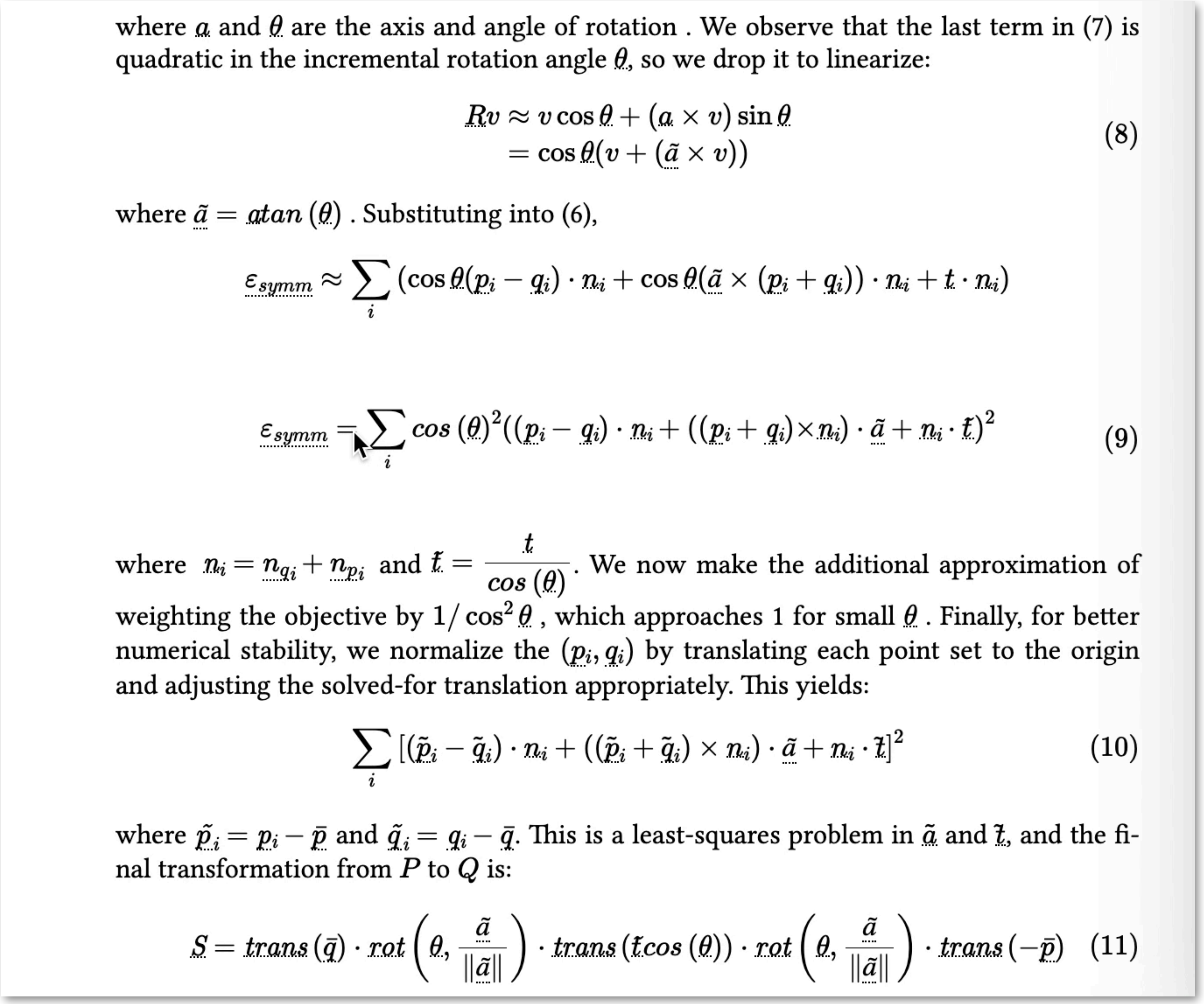
$$\varepsilon_{plane} = \sum_i \left((Rp_i + t - q_i) \cdot n_{q_i}\right)^2 \tag{2}$$

$n_q \in$ sequence of $\mathbb{R}^3$: surface normals $n_{q,i}$

It can be shown that minimizing this objective is equivalent to Gauss-Newton minimiza-

H❤️rtDown Design: Reading Environment

- Equation relationships



where a and θ are the axis and angle of rotation . We observe that the last term in (7) is quadratic in the incremental rotation angle θ, so we drop it to linearize:

$$\begin{aligned} Rv &\approx v\cos\theta + (a \times v)\sin\theta \\ &= \cos\theta(v + (\tilde{a} \times v)) \end{aligned} \tag{8}$$

where $\tilde{a} = atan\,(\theta)$. Substituting into (6),

$$\varepsilon_{symm} \approx \sum_i (\cos\theta(p_i - q_i) \cdot n_i + \cos\theta(\tilde{a} \times (p_i + q_i)) \cdot n_i + t \cdot n_i)$$

$$\varepsilon_{symm} = \sum_i cos\,(\theta)^2((p_i - q_i) \cdot n_i + ((p_i + q_i) \times n_i) \cdot \tilde{a} + n_i \cdot \tilde{t})^2 \tag{9}$$

where $n_i = n_{q_i} + n_{p_i}$ and $\tilde{t} = \dfrac{t}{cos\,(\theta)}$. We now make the additional approximation of weighting the objective by $1/\cos^2\theta$, which approaches 1 for small θ . Finally, for better numerical stability, we normalize the (p_i, q_i) by translating each point set to the origin and adjusting the solved-for translation appropriately. This yields:

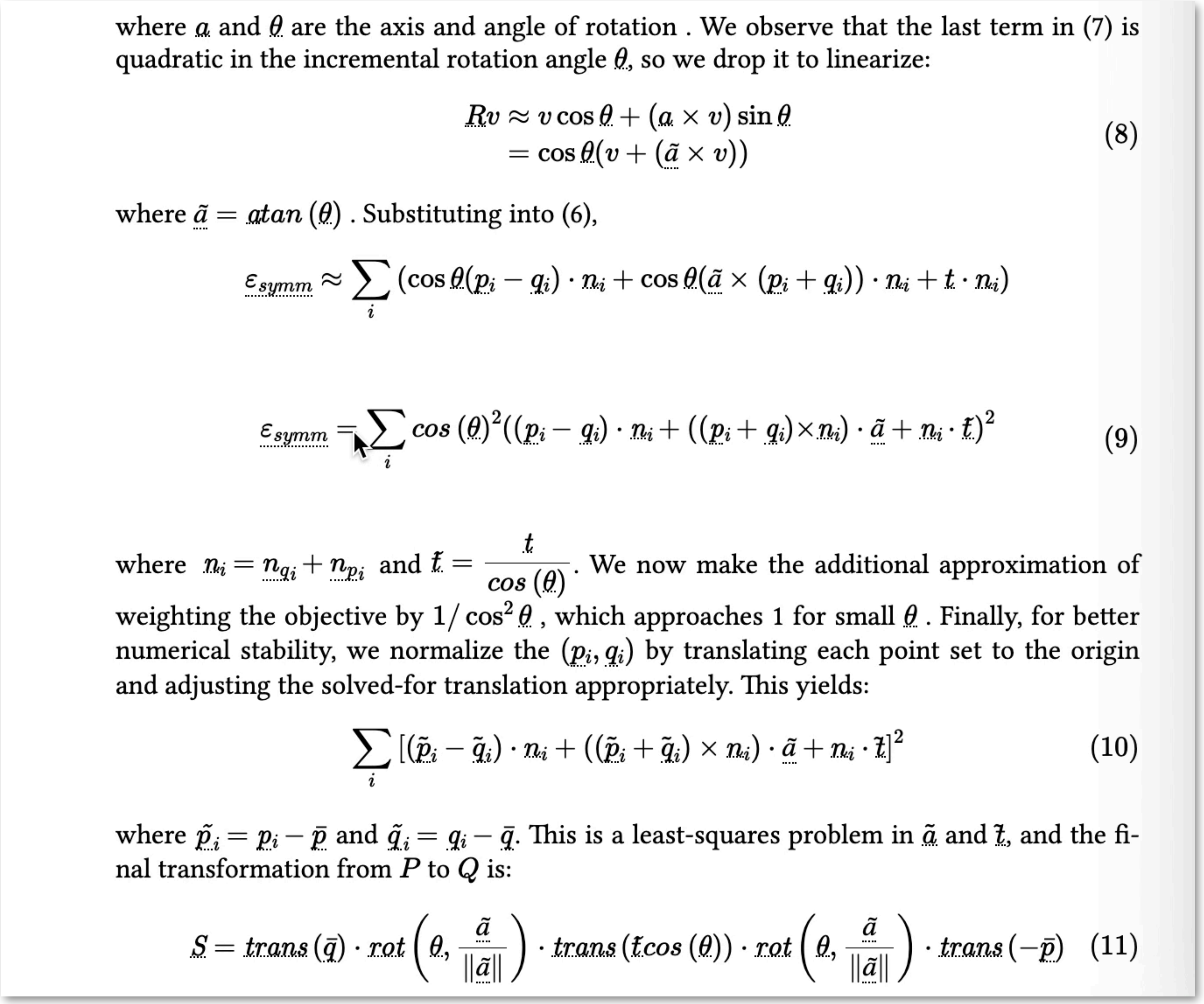
$$\sum_i [(\tilde{p}_i - \tilde{q}_i) \cdot n_i + ((\tilde{p}_i + \tilde{q}_i) \times n_i) \cdot \tilde{a} + n_i \cdot \tilde{t}]^2 \tag{10}$$

where $\tilde{p}_i = p_i - \bar{p}$ and $\tilde{q}_i = q_i - \bar{q}$. This is a least-squares problem in $\tilde{a}$ and $\tilde{t}$, and the final transformation from P to Q is:

$$S = trans\,(\bar{q}) \cdot rot\left(\theta, \frac{\tilde{a}}{\|\tilde{a}\|}\right) \cdot trans\,(\tilde{t}cos\,(\theta)) \cdot rot\left(\theta, \frac{\tilde{a}}{\|\tilde{a}\|}\right) \cdot trans\,(-\bar{p}) \tag{11}$$

H❤️rtDown Design: Reading Environment

- Equation relationships



where a and θ are the axis and angle of rotation . We observe that the last term in (7) is quadratic in the incremental rotation angle θ, so we drop it to linearize:

$$\begin{aligned} Rv &\approx v\cos\theta + (a \times v)\sin\theta \\ &= \cos\theta(v + (\tilde{a} \times v)) \end{aligned} \tag{8}$$

where $\tilde{a} = atan\,(\theta)$. Substituting into (6),

$$\varepsilon_{symm} \approx \sum_i (\cos\theta(p_i - q_i) \cdot n_i + \cos\theta(\tilde{a} \times (p_i + q_i)) \cdot n_i + t \cdot n_i)$$

$$\varepsilon_{symm} = \sum_i cos\,(\theta)^2((p_i - q_i) \cdot n_i + ((p_i + q_i) \times n_i) \cdot \tilde{a} + n_i \cdot \tilde{t})^2 \tag{9}$$

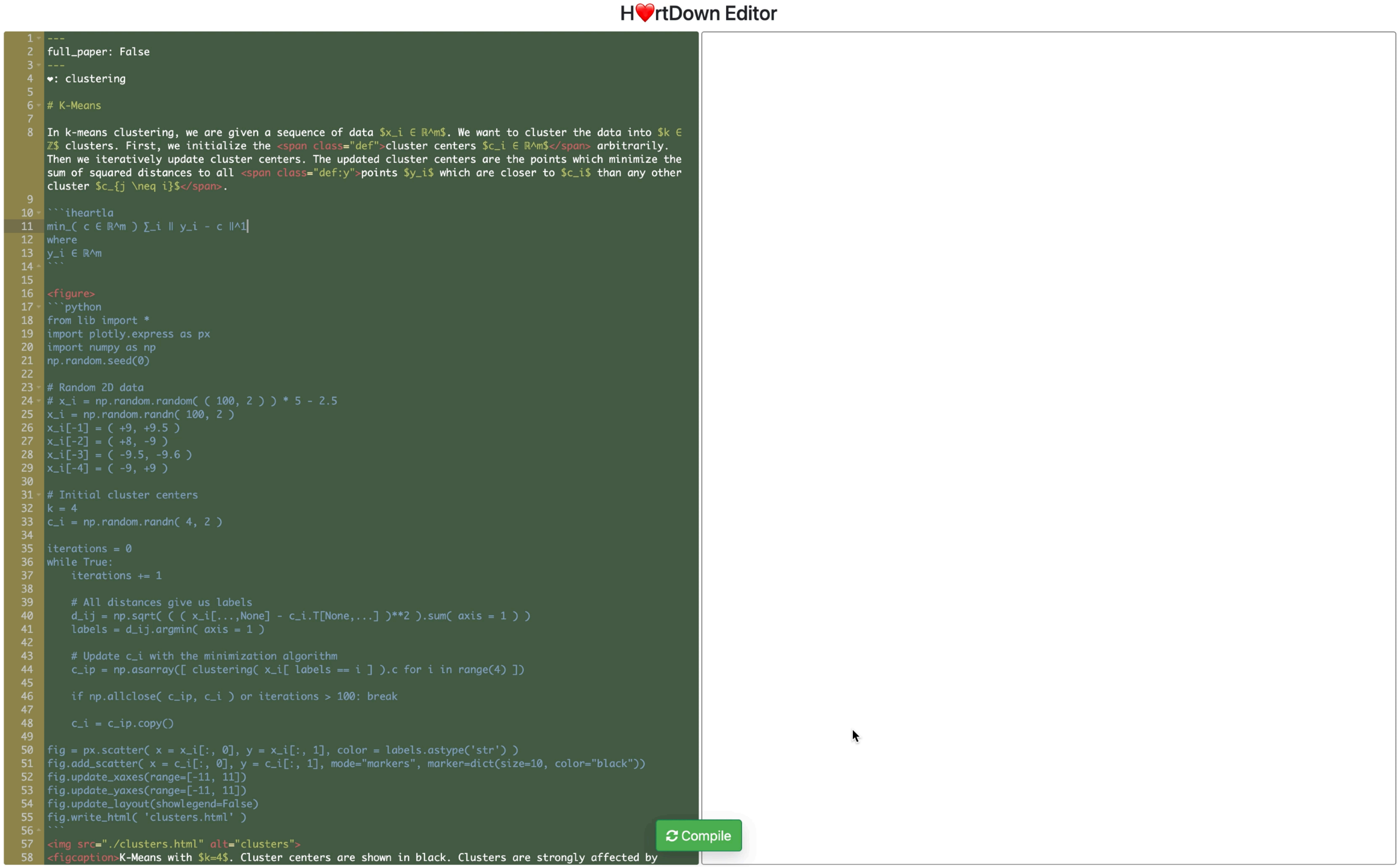
where $n_i = n_{q_i} + n_{p_i}$ and $\tilde{t} = \frac{t}{cos\,(\theta)}$. We now make the additional approximation of weighting the objective by $1/\cos^2\theta$, which approaches 1 for small θ . Finally, for better numerical stability, we normalize the (p_i, q_i) by translating each point set to the origin and adjusting the solved-for translation appropriately. This yields:

$$\sum_i [(\tilde{p}_i - \tilde{q}_i) \cdot n_i + ((\tilde{p}_i + \tilde{q}_i) \times n_i) \cdot \tilde{a} + n_i \cdot \tilde{t}]^2 \tag{10}$$

where $\tilde{p}_i = p_i - \bar{p}$ and $\tilde{q}_i = q_i - \bar{q}$. This is a least-squares problem in $\tilde{a}$ and $\tilde{t}$, and the final transformation from P to Q is:

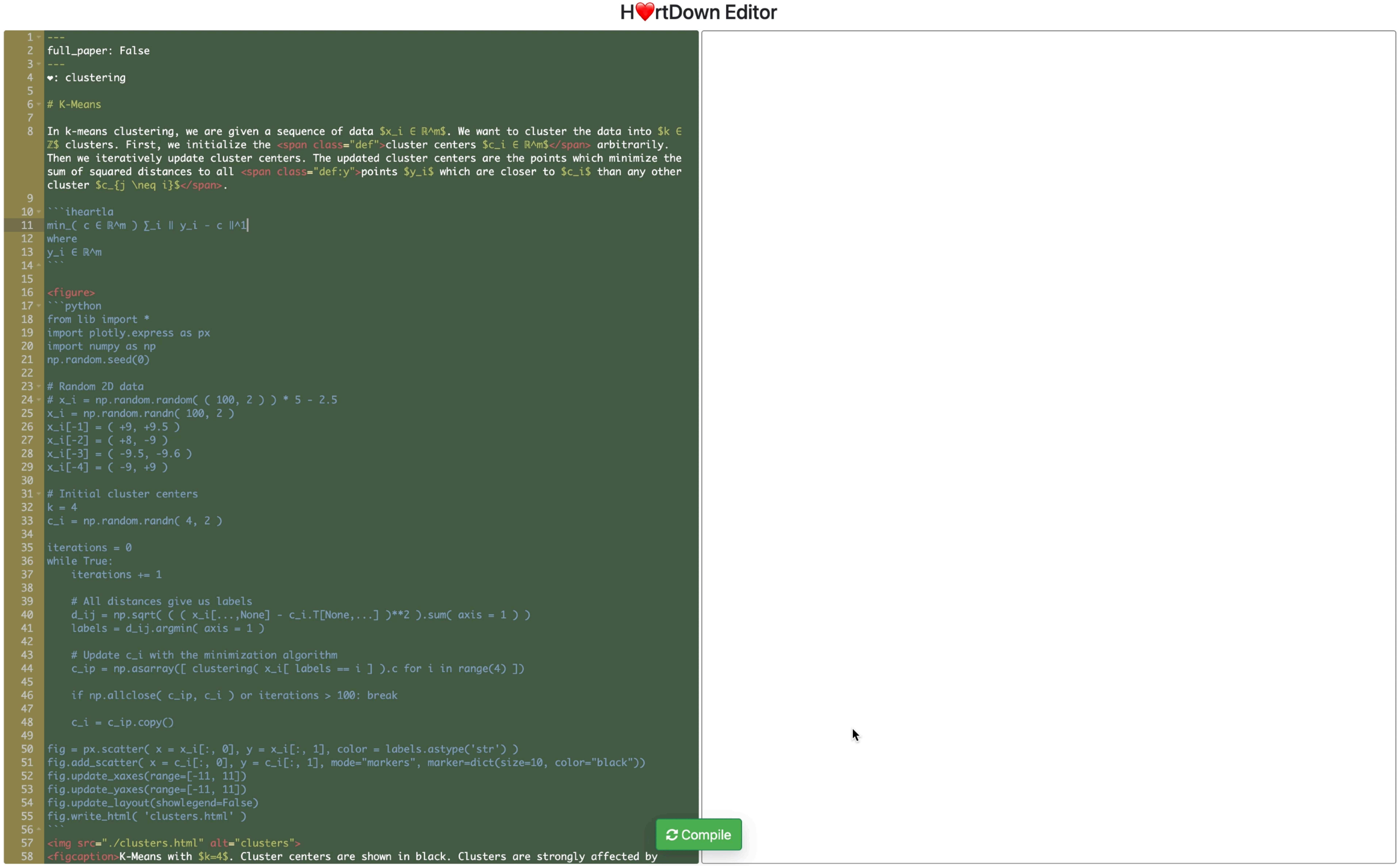
$$S = trans\,(\bar{q}) \cdot rot\left(\theta, \frac{\tilde{a}}{\|\tilde{a}\|}\right) \cdot trans\,(\tilde{t}cos\,(\theta)) \cdot rot\left(\theta, \frac{\tilde{a}}{\|\tilde{a}\|}\right) \cdot trans\,(-\bar{p}) \tag{11}$$

H❤️rtDown Design: Experimenter (making use of)



~~~
---
full_paper: False
---
❤: clustering

# K-Means

In k-means clustering, we are given a sequence of data $x_i ∈ ℝ^m$. We want to cluster the data into $k ∈ ℤ$ clusters. First, we initialize the <span class="def">cluster centers $c_i ∈ ℝ^m$</span> arbitrarily. Then we iteratively update cluster centers. The updated cluster centers are the points which minimize the sum of squared distances to all <span class="def:y">points $y_i$ which are closer to $c_i$ than any other cluster $c_{j \neq i}$</span>.

```iheartla
min_( c ∈ ℝ^m ) ∑_i ‖ y_i - c ‖^1
where
y_i ∈ ℝ^m
```


<figure>
```python
from lib import *
import plotly.express as px
import numpy as np
np.random.seed(0)

# Random 2D data
# x_i = np.random.random( ( 100, 2 ) ) * 5 - 2.5
x_i = np.random.randn( 100, 2 )
x_i[-1] = ( +9, +9.5 )
x_i[-2] = ( +8, -9 )
x_i[-3] = ( -9.5, -9.6 )
x_i[-4] = ( -9, +9 )

# Initial cluster centers
k = 4
c_i = np.random.randn( 4, 2 )

iterations = 0
while True:
    iterations += 1

    # All distances give us labels
    d_ij = np.sqrt( ( ( x_i[...,None] - c_i.T[None,...] )**2 ).sum( axis = 1 ) )
    labels = d_ij.argmin( axis = 1 )

    # Update c_i with the minimization algorithm
    c_ip = np.asarray([ clustering( x_i[ labels == i ] ).c for i in range(4) ])

    if np.allclose( c_ip, c_i ) or iterations > 100: break

    c_i = c_ip.copy()

fig = px.scatter( x = x_i[:, 0], y = x_i[:, 1], color = labels.astype('str') )
fig.add_scatter( x = c_i[:, 0], y = c_i[:, 1], mode="markers", marker=dict(size=10, color="black"))
fig.update_xaxes(range=[-11, 11])
fig.update_yaxes(range=[-11, 11])
fig.update_layout(showlegend=False)
fig.write_html( 'clusters.html' )
```
<img src="./clusters.html" alt="clusters">
<figcaption>K-Means with $k=4$. Cluster centers are shown in black. Clusters are strongly affected by
~~~

H❤️rtDown Design: Experimenter (making use of)



~~~~
---
full_paper: False
---
❤: clustering

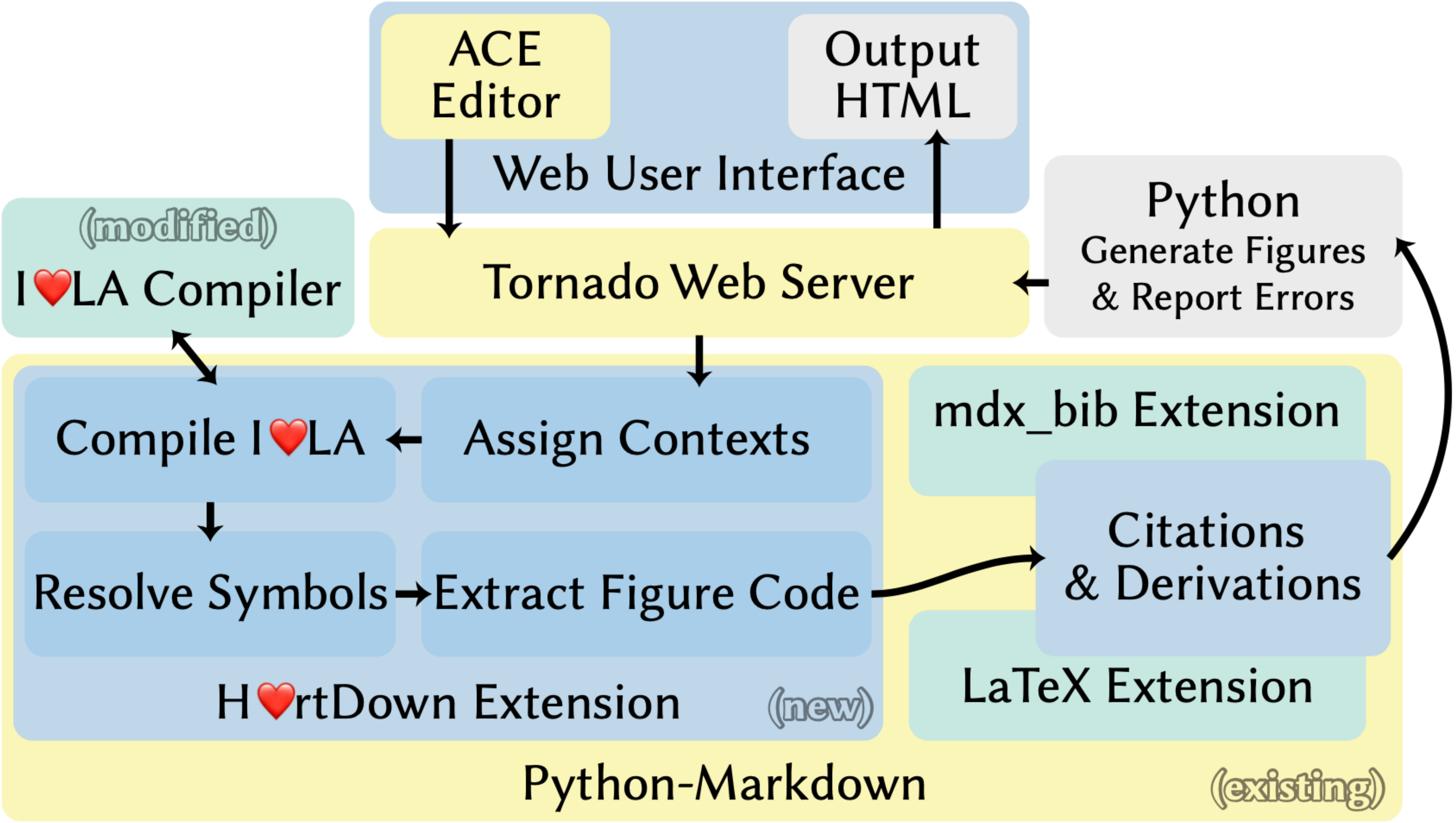
# K-Means

In k-means clustering, we are given a sequence of data $x_i ∈ ℝ^m$. We want to cluster the data into $k ∈ ℤ$ clusters. First, we initialize the <span class="def">cluster centers $c_i ∈ ℝ^m$</span> arbitrarily. Then we iteratively update cluster centers. The updated cluster centers are the points which minimize the sum of squared distances to all <span class="def:y">points $y_i$ which are closer to $c_i$ than any other cluster $c_{j \neq i}$</span>.

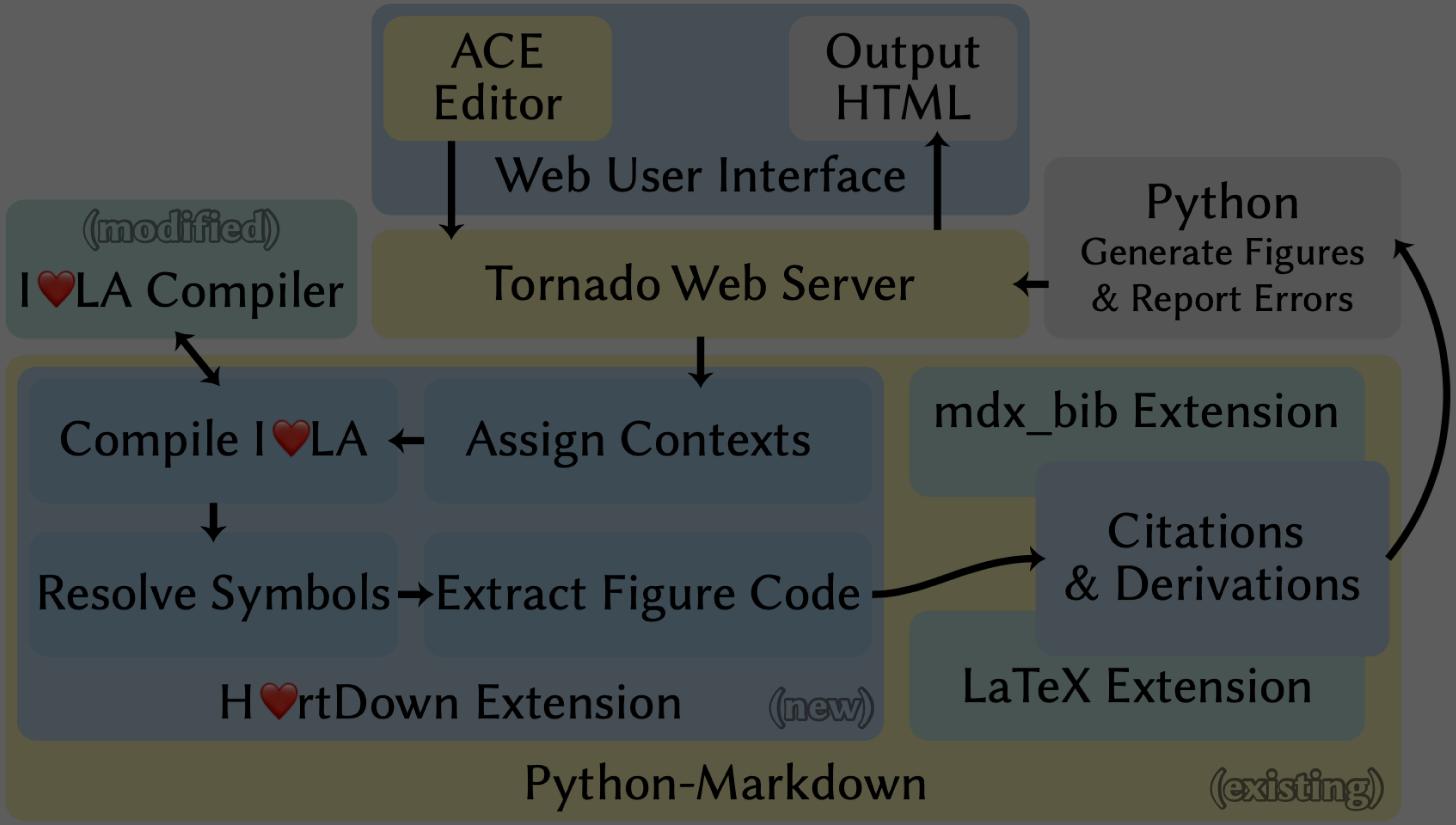
```iheartla
min_( c ∈ ℝ^m ) ∑_i ‖ y_i - c ‖^1
where
y_i ∈ ℝ^m
```


<figure>
```python
from lib import *
import plotly.express as px
import numpy as np
np.random.seed(0)

# Random 2D data
# x_i = np.random.random( ( 100, 2 ) ) * 5 - 2.5
x_i = np.random.randn( 100, 2 )
x_i[-1] = ( +9, +9.5 )
x_i[-2] = ( +8, -9 )
x_i[-3] = ( -9.5, -9.6 )
x_i[-4] = ( -9, +9 )

# Initial cluster centers
k = 4
c_i = np.random.randn( 4, 2 )

iterations = 0
while True:
    iterations += 1

    # All distances give us labels
    d_ij = np.sqrt( ( ( x_i[...,None] - c_i.T[None,...] )**2 ).sum( axis = 1 ) )
    labels = d_ij.argmin( axis = 1 )

    # Update c_i with the minimization algorithm
    c_ip = np.asarray([ clustering( x_i[ labels == i ] ).c for i in range(4) ])

    if np.allclose( c_ip, c_i ) or iterations > 100: break

    c_i = c_ip.copy()

fig = px.scatter( x = x_i[:, 0], y = x_i[:, 1], color = labels.astype('str') )
fig.add_scatter( x = c_i[:, 0], y = c_i[:, 1], mode="markers", marker=dict(size=10, color="black"))
fig.update_xaxes(range=[-11, 11])
fig.update_yaxes(range=[-11, 11])
fig.update_layout(showlegend=False)
fig.write_html( 'clusters.html' )
```
<img src="./clusters.html" alt="clusters">
<figcaption>K-Means with $k=4$. Cluster centers are shown in black. Clusters are strongly affected by
~~~~

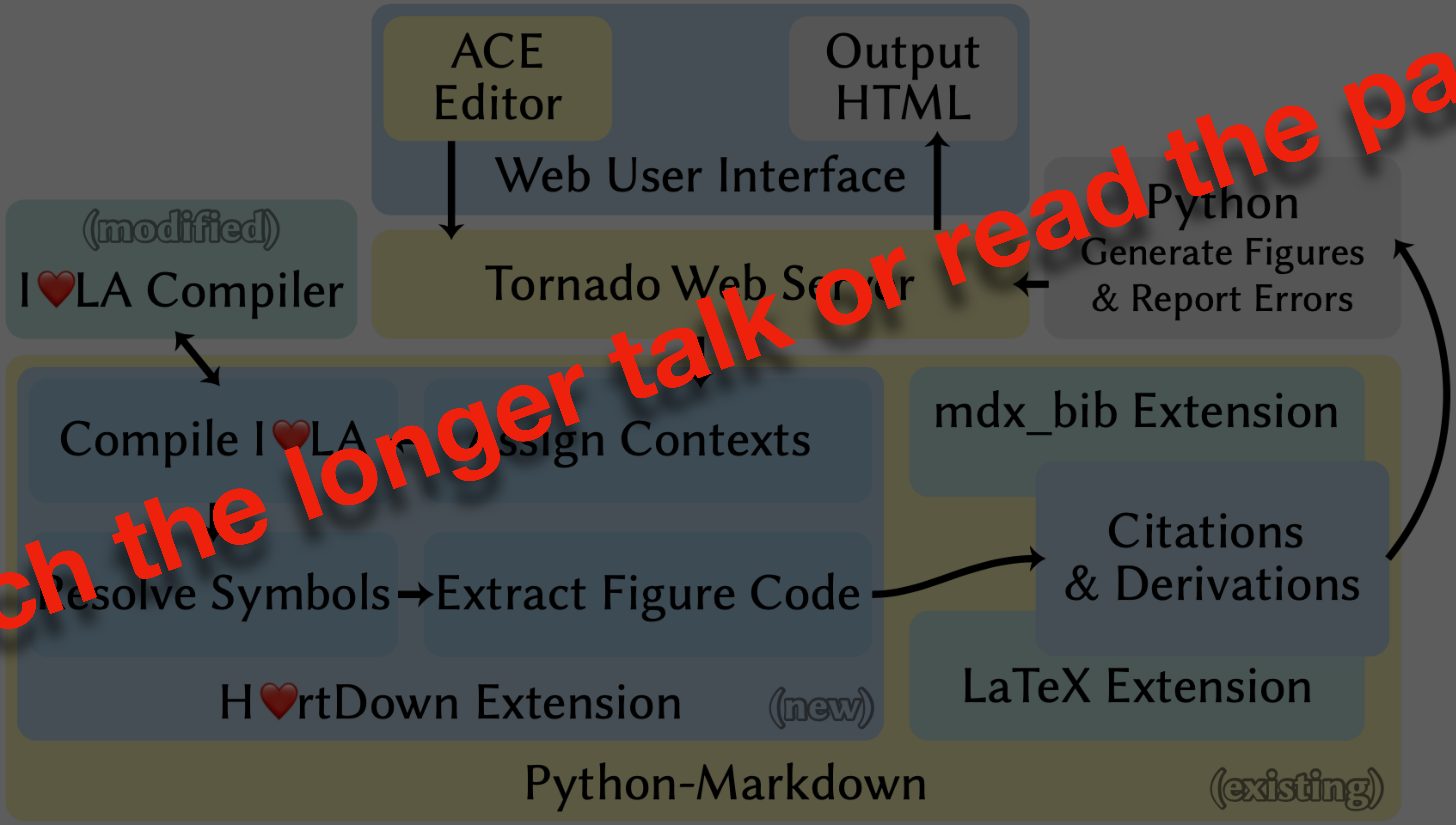
Implementation



Implementation



Implementation



ACE Editor

Output HTML

Web User Interface

(modified)

I❤LA Compiler

Tornado Web Server

Python
Generate Figures
& Report Errors

Compile I❤LA

Assign Contexts

mdx_bib Extension

Citations
& Derivations

Resolve Symbols → Extract Figure Code

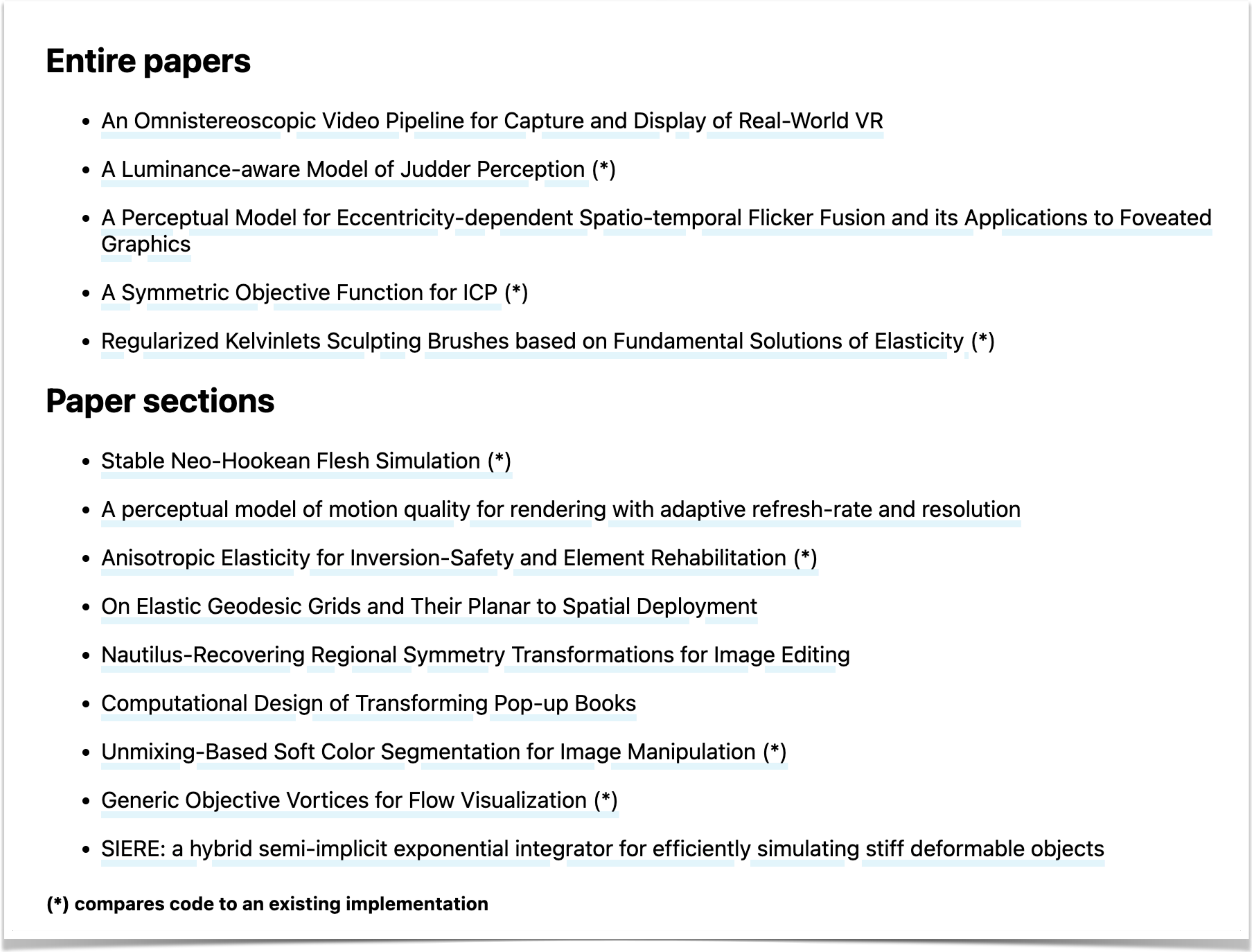
H❤rtDown Extension (new)

LaTeX Extension

Python-Markdown (existing)

Watch the longer talk or read the paper

H❤️rtDown Case Studies



Entire papers

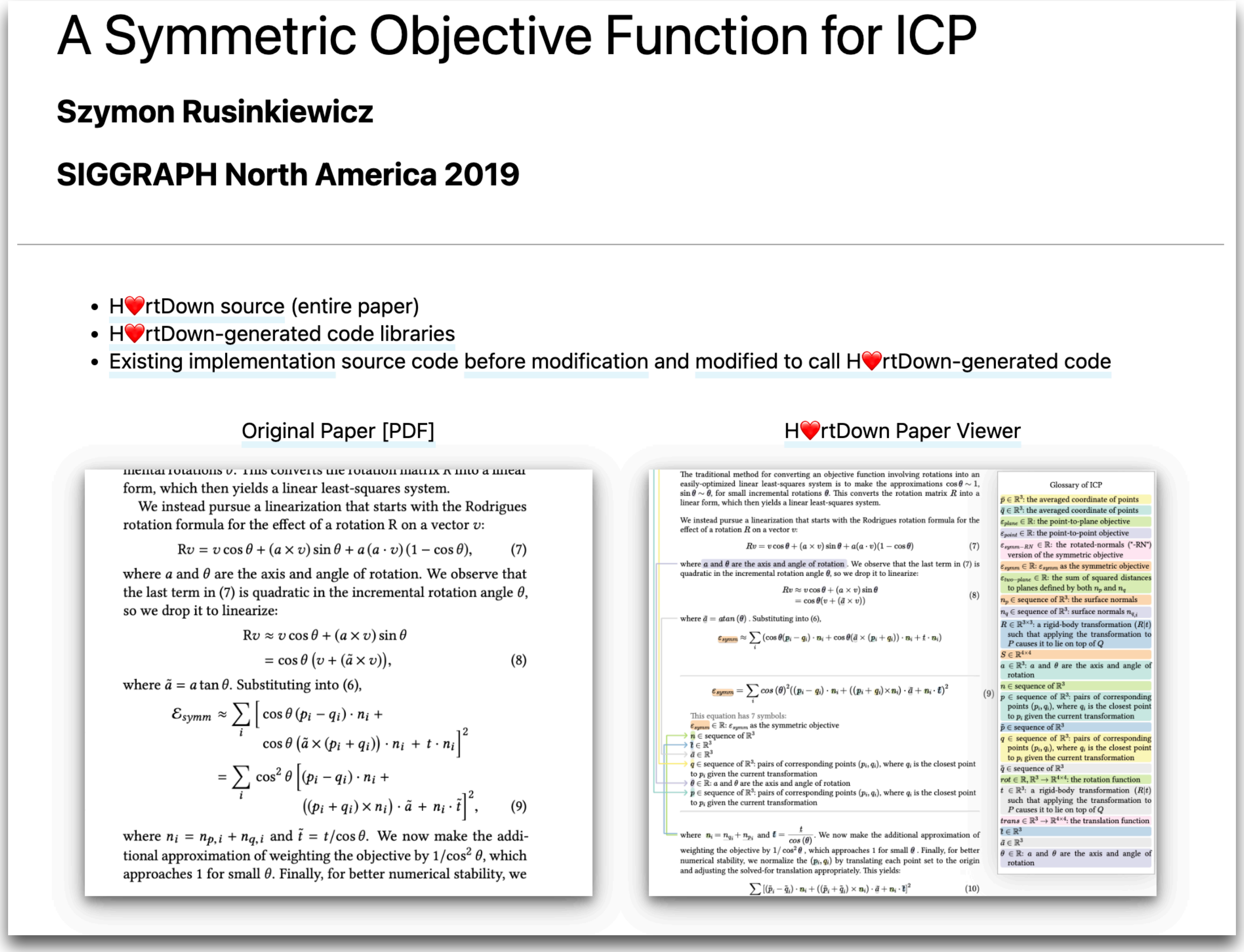
- An Omnistereoscopic Video Pipeline for Capture and Display of Real-World VR
- A Luminance-aware Model of Judder Perception (*)
- A Perceptual Model for Eccentricity-dependent Spatio-temporal Flicker Fusion and its Applications to Foveated Graphics
- A Symmetric Objective Function for ICP (*)
- Regularized Kelvinlets Sculpting Brushes based on Fundamental Solutions of Elasticity (*)

Paper sections

- Stable Neo-Hookean Flesh Simulation (*)
- A perceptual model of motion quality for rendering with adaptive refresh-rate and resolution
- Anisotropic Elasticity for Inversion-Safety and Element Rehabilitation (*)
- On Elastic Geodesic Grids and Their Planar to Spatial Deployment
- Nautilus-Recovering Regional Symmetry Transformations for Image Editing
- Computational Design of Transforming Pop-up Books
- Unmixing-Based Soft Color Segmentation for Image Manipulation (*)
- Generic Objective Vortices for Flow Visualization (*)
- SIERE: a hybrid semi-implicit exponential integrator for efficiently simulating stiff deformable objects

(*) compares code to an existing implementation

H❤rtDown Case Studies



A Symmetric Objective Function for ICP

Szymon Rusinkiewicz

SIGGRAPH North America 2019

- H❤rtDown source (entire paper)
- H❤rtDown-generated code libraries
- Existing implementation source code before modification and modified to call H❤rtDown-generated code

Original Paper [PDF]

mental rotations θ. This converts the rotation matrix R into a linear form, which then yields a linear least-squares system.

We instead pursue a linearization that starts with the Rodrigues rotation formula for the effect of a rotation R on a vector v:

$$Rv = v\cos\theta + (a \times v)\sin\theta + a\,(a \cdot v)\,(1 - \cos\theta), \tag{7}$$

where a and θ are the axis and angle of rotation. We observe that the last term in (7) is quadratic in the incremental rotation angle θ, so we drop it to linearize:

$$\begin{aligned} Rv &\approx v\cos\theta + (a \times v)\sin\theta \\ &= \cos\theta\,(v + (\tilde{a} \times v)), \end{aligned} \tag{8}$$

where $\tilde{a} = a\tan\theta$. Substituting into (6),

$$\begin{aligned} \mathcal{E}_{symm} &\approx \sum_i \Big[\cos\theta\,(p_i - q_i)\cdot n_i + \\ &\qquad \cos\theta\,(\tilde{a} \times (p_i + q_i))\cdot n_i + t \cdot n_i\Big]^2 \\ &= \sum_i \cos^2\theta\,\Big[(p_i - q_i)\cdot n_i + \\ &\qquad ((p_i + q_i) \times n_i)\cdot \tilde{a} + n_i \cdot \tilde{t}\Big]^2, \end{aligned} \tag{9}$$

where $n_i = n_{p,i} + n_{q,i}$ and $\tilde{t} = t/\cos\theta$. We now make the additional approximation of weighting the objective by $1/\cos^2\theta$, which approaches 1 for small θ. Finally, for better numerical stability, we

H❤rtDown Paper Viewer

The traditional method for converting an objective function involving rotations into an easily-optimized linear least-squares system is to make the approximations $\cos\theta \sim 1$, $\sin\theta \sim \theta$, for small incremental rotations θ. This converts the rotation matrix R into a linear form, which then yields a linear least-squares system.

We instead pursue a linearization that starts with the Rodrigues rotation formula for the effect of a rotation R on a vector v:

$$Rv = v\cos\theta + (a \times v)\sin\theta + a(a \cdot v)(1 - \cos\theta) \tag{7}$$

where a and θ are the axis and angle of rotation . We observe that the last term in (7) is quadratic in the incremental rotation angle θ, so we drop it to linearize:

$$\begin{aligned} Rv &\approx v\cos\theta + (a \times v)\sin\theta \\ &= \cos\theta(v + (\tilde{a} \times v)) \end{aligned} \tag{8}$$

where $\tilde{a} = atan(\theta)$. Substituting into (6),

$$\varepsilon_{symm} \approx \sum_i (\cos\theta(p_i - q_i)\cdot n_i + \cos\theta(\tilde{a} \times (p_i + q_i))\cdot n_i + t \cdot n_i)$$

$$\varepsilon_{symm} = \sum_i cos(\theta)^2((p_i - q_i)\cdot n_i + ((p_i + q_i) \times n_i)\cdot \tilde{a} + n_i \cdot \tilde{t})^2 \tag{9}$$

This equation has 7 symbols:
$\varepsilon_{symm} \in \mathbb{R}$: ε_{symm} as the symmetric objective
$n \in$ sequence of $\mathbb{R}^3$
$\tilde{t} \in \mathbb{R}^3$
$\tilde{a} \in \mathbb{R}^3$
$q \in$ sequence of $\mathbb{R}^3$: pairs of corresponding points (p_i, q_i), where q_i is the closest point to p_i given the current transformation
$\theta \in \mathbb{R}$: a and θ are the axis and angle of rotation
$p \in$ sequence of $\mathbb{R}^3$: pairs of corresponding points (p_i, q_i), where q_i is the closest point to p_i given the current transformation

where $n_i = n_{q_i} + n_{p_i}$ and $\tilde{t} = \frac{t}{cos(\theta)}$. We now make the additional approximation of weighting the objective by $1/\cos^2\theta$, which approaches 1 for small θ . Finally, for better numerical stability, we normalize the (p_i, q_i) by translating each point set to the origin and adjusting the solved-for translation appropriately. This yields:

$$\sum [(\tilde{p}_i - \tilde{q}_i)\cdot n_i + ((\tilde{p}_i + \tilde{q}_i) \times n_i)\cdot \tilde{a} + n_i \cdot \tilde{t}]^2 \tag{10}$$

Glossary of ICP

$\bar{p} \in \mathbb{R}^3$: the averaged coordinate of points
$\bar{q} \in \mathbb{R}^3$: the averaged coordinate of points
$\varepsilon_{plane} \in \mathbb{R}$: the point-to-plane objective
$\varepsilon_{point} \in \mathbb{R}$: the point-to-point objective
$\varepsilon_{symm-RN} \in \mathbb{R}$: the rotated-normals ("-RN") version of the symmetric objective
$\varepsilon_{symm} \in \mathbb{R}$: ε_{symm} as the symmetric objective
$\varepsilon_{two-plane} \in \mathbb{R}$: the sum of squared distances to planes defined by both n_p and n_q
$n_p \in$ sequence of $\mathbb{R}^3$: the surface normals
$n_q \in$ sequence of $\mathbb{R}^3$: surface normals $n_{q,i}$
$R \in \mathbb{R}^{3\times3}$: a rigid-body transformation $(R|t)$ such that applying the transformation to P causes it to lie on top of Q
$S \in \mathbb{R}^{4\times4}$
$a \in \mathbb{R}^3$: a and θ are the axis and angle of rotation
$n \in$ sequence of $\mathbb{R}^3$
$p \in$ sequence of $\mathbb{R}^3$: pairs of corresponding points (p_i, q_i), where q_i is the closest point to p_i given the current transformation
$\tilde{p} \in$ sequence of $\mathbb{R}^3$
$q \in$ sequence of $\mathbb{R}^3$: pairs of corresponding points (p_i, q_i), where q_i is the closest point to p_i given the current transformation
$\tilde{q} \in$ sequence of $\mathbb{R}^3$
$rot \in \mathbb{R}, \mathbb{R}^3 \to \mathbb{R}^{4\times4}$: the rotation function
$t \in \mathbb{R}^3$: a rigid-body transformation $(R|t)$ such that applying the transformation to P causes it to lie on top of Q
$trans \in \mathbb{R}^3 \to \mathbb{R}^{4\times4}$: the translation function
$\tilde{t} \in \mathbb{R}^3$
$\tilde{a} \in \mathbb{R}^3$
$\theta \in \mathbb{R}$: a and θ are the axis and angle of rotation

Expert Study

Expert Study

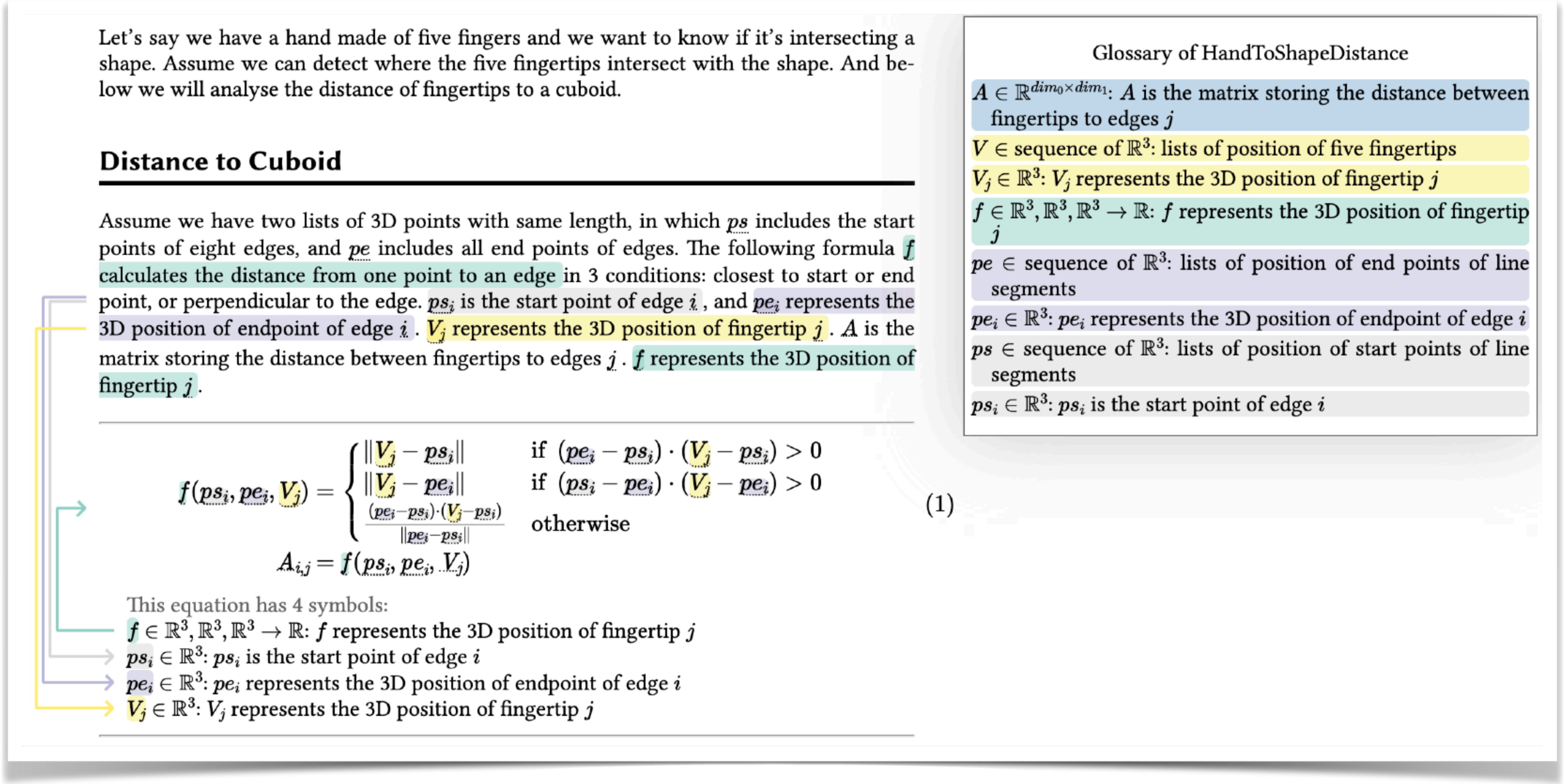
- 3 CS PhD students

Expert Study

- 3 CS PhD students
- Author an original document related to their computer graphics research

Expert Study

- 3 CS PhD students
- Author an original document related to their computer graphics research



Let's say we have a hand made of five fingers and we want to know if it's intersecting a shape. Assume we can detect where the five fingertips intersect with the shape. And below we will analyse the distance of fingertips to a cuboid.

Distance to Cuboid

Assume we have two lists of 3D points with same length, in which ps includes the start points of eight edges, and pe includes all end points of edges. The following formula f calculates the distance from one point to an edge in 3 conditions: closest to start or end point, or perpendicular to the edge. ps_i is the start point of edge i , and pe_i represents the 3D position of endpoint of edge i . V_j represents the 3D position of fingertip j . A is the matrix storing the distance between fingertips to edges j . f represents the 3D position of fingertip j .

$$f(ps_i, pe_i, V_j) = \begin{cases} \|V_j - ps_i\| & \text{if } (pe_i - ps_i)\cdot(V_j - ps_i) > 0 \\ \|V_j - pe_i\| & \text{if } (ps_i - pe_i)\cdot(V_j - pe_i) > 0 \\ \frac{(pe_i - ps_i)\cdot(V_j - ps_i)}{\|pe_i - ps_i\|} & \text{otherwise} \end{cases} \qquad (1)$$

$$A_{i,j} = f(ps_i, pe_i, V_j)$$

This equation has 4 symbols:

$f \in \mathbb{R}^3, \mathbb{R}^3, \mathbb{R}^3 \to \mathbb{R}$: f represents the 3D position of fingertip j

$ps_i \in \mathbb{R}^3$: ps_i is the start point of edge i

$pe_i \in \mathbb{R}^3$: pe_i represents the 3D position of endpoint of edge i

$V_j \in \mathbb{R}^3$: V_j represents the 3D position of fingertip j

Glossary of HandToShapeDistance

$A \in \mathbb{R}^{dim_0 \times dim_1}$: A is the matrix storing the distance between fingertips to edges j

$V \in$ sequence of $\mathbb{R}^3$: lists of position of five fingertips

$V_j \in \mathbb{R}^3$: V_j represents the 3D position of fingertip j

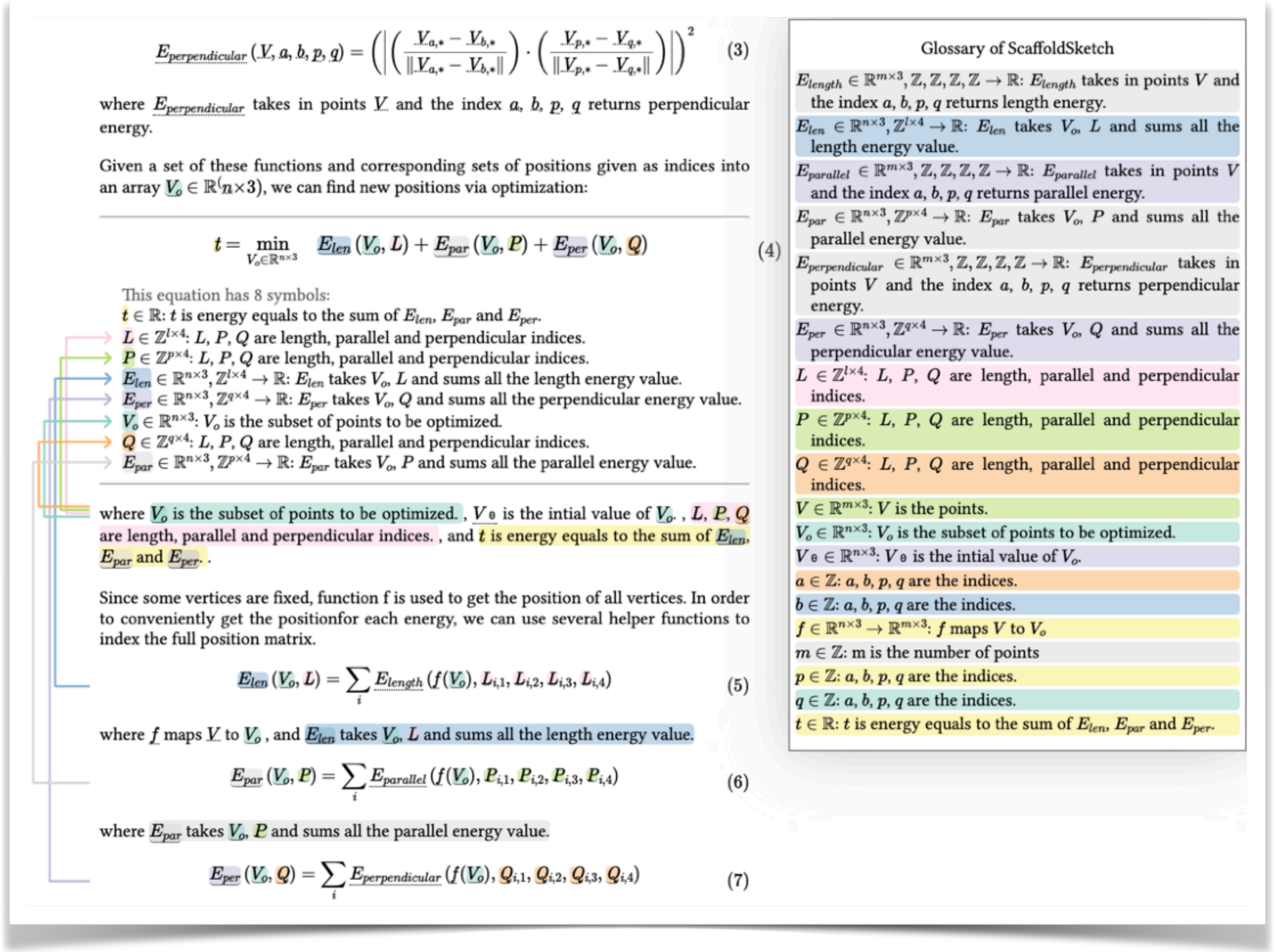
$f \in \mathbb{R}^3, \mathbb{R}^3, \mathbb{R}^3 \to \mathbb{R}$: f represents the 3D position of fingertip j

$pe \in$ sequence of $\mathbb{R}^3$: lists of position of end points of line segments

$pe_i \in \mathbb{R}^3$: pe_i represents the 3D position of endpoint of edge i

$ps \in$ sequence of $\mathbb{R}^3$: lists of position of start points of line segments

$ps_i \in \mathbb{R}^3$: ps_i is the start point of edge i



$$E_{perpendicular}(V, a, b, p, q) = \left(\left|\left(\frac{V_{a,*} - V_{b,*}}{\|V_{a,*} - V_{b,*}\|}\right) \cdot \left(\frac{V_{p,*} - V_{q,*}}{\|V_{p,*} - V_{q,*}\|}\right)\right|\right)^2 \qquad (3)$$

where $E_{perpendicular}$ takes in points V and the index a, b, p, q returns perpendicular energy.

Given a set of these functions and corresponding sets of positions given as indices into an array $V_o \in \mathbb{R}^(n \times 3)$, we can find new positions via optimization:

$$t = \min_{V_o \in \mathbb{R}^{n\times 3}} E_{len}(V_o, L) + E_{par}(V_o, P) + E_{per}(V_o, Q) \qquad (4)$$

This equation has 8 symbols:

$t \in \mathbb{R}$: t is energy equals to the sum of E_{len}, E_{par} and E_{per}.

$L \in \mathbb{Z}^{l\times 4}$: L, P, Q are length, parallel and perpendicular indices.

$P \in \mathbb{Z}^{p\times 4}$: L, P, Q are length, parallel and perpendicular indices.

$E_{len} \in \mathbb{R}^{n\times 3}, \mathbb{Z}^{l\times 4} \to \mathbb{R}$: E_{len} takes V_o, L and sums all the length energy value.

$E_{per} \in \mathbb{R}^{n\times 3}, \mathbb{Z}^{q\times 4} \to \mathbb{R}$: E_{per} takes V_o, Q and sums all the perpendicular energy value.

$V_o \in \mathbb{R}^{n\times 3}$: V_o is the subset of points to be optimized.

$Q \in \mathbb{Z}^{q\times 4}$: L, P, Q are length, parallel and perpendicular indices.

$E_{par} \in \mathbb{R}^{n\times 3}, \mathbb{Z}^{p\times 4} \to \mathbb{R}$: E_{par} takes V_o, P and sums all the parallel energy value.

where V_o is the subset of points to be optimized. , V_θ is the intial value of V_o. , L, P, Q are length, parallel and perpendicular indices. , and t is energy equals to the sum of E_{len}, E_{par} and E_{per}. .

Since some vertices are fixed, function f is used to get the position of all vertices. In order to conveniently get the positionfor each energy, we can use several helper functions to index the full position matrix.

$$E_{len}(V_o, L) = \sum_i E_{length}(f(V_o), L_{i,1}, L_{i,2}, L_{i,3}, L_{i,4}) \qquad (5)$$

where f maps V to V_o , and E_{len} takes V_o, L and sums all the length energy value.

$$E_{par}(V_o, P) = \sum_i E_{parallel}(f(V_o), P_{i,1}, P_{i,2}, P_{i,3}, P_{i,4}) \qquad (6)$$

where E_{par} takes V_o, P and sums all the parallel energy value.

$$E_{per}(V_o, Q) = \sum_i E_{perpendicular}(f(V_o), Q_{i,1}, Q_{i,2}, Q_{i,3}, Q_{i,4}) \qquad (7)$$

Glossary of ScaffoldSketch

$E_{length} \in \mathbb{R}^{m\times 3}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z} \to \mathbb{R}$: E_{length} takes in points V and the index a, b, p, q returns length energy.

$E_{len} \in \mathbb{R}^{n\times 3}, \mathbb{Z}^{l\times 4} \to \mathbb{R}$: E_{len} takes V_o, L and sums all the length energy value.

$E_{parallel} \in \mathbb{R}^{m\times 3}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z} \to \mathbb{R}$: $E_{parallel}$ takes in points V and the index a, b, p, q returns parallel energy.

$E_{par} \in \mathbb{R}^{n\times 3}, \mathbb{Z}^{p\times 4} \to \mathbb{R}$: E_{par} takes V_o, P and sums all the parallel energy value.

$E_{perpendicular} \in \mathbb{R}^{m\times 3}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z} \to \mathbb{R}$: $E_{perpendicular}$ takes in points V and the index a, b, p, q returns perpendicular energy.

$E_{per} \in \mathbb{R}^{n\times 3}, \mathbb{Z}^{q\times 4} \to \mathbb{R}$: E_{per} takes V_o, Q and sums all the perpendicular energy value.

$L \in \mathbb{Z}^{l\times 4}$: L, P, Q are length, parallel and perpendicular indices.

$P \in \mathbb{Z}^{p\times 4}$: L, P, Q are length, parallel and perpendicular indices.

$Q \in \mathbb{Z}^{q\times 4}$: L, P, Q are length, parallel and perpendicular indices.

$V \in \mathbb{R}^{m\times 3}$: V is the points.

$V_o \in \mathbb{R}^{n\times 3}$: V_o is the subset of points to be optimized.

$V_\theta \in \mathbb{R}^{n\times 3}$: V_θ is the intial value of V_o.

$a \in \mathbb{Z}$: a, b, p, q are the indices.

$b \in \mathbb{Z}$: a, b, p, q are the indices.

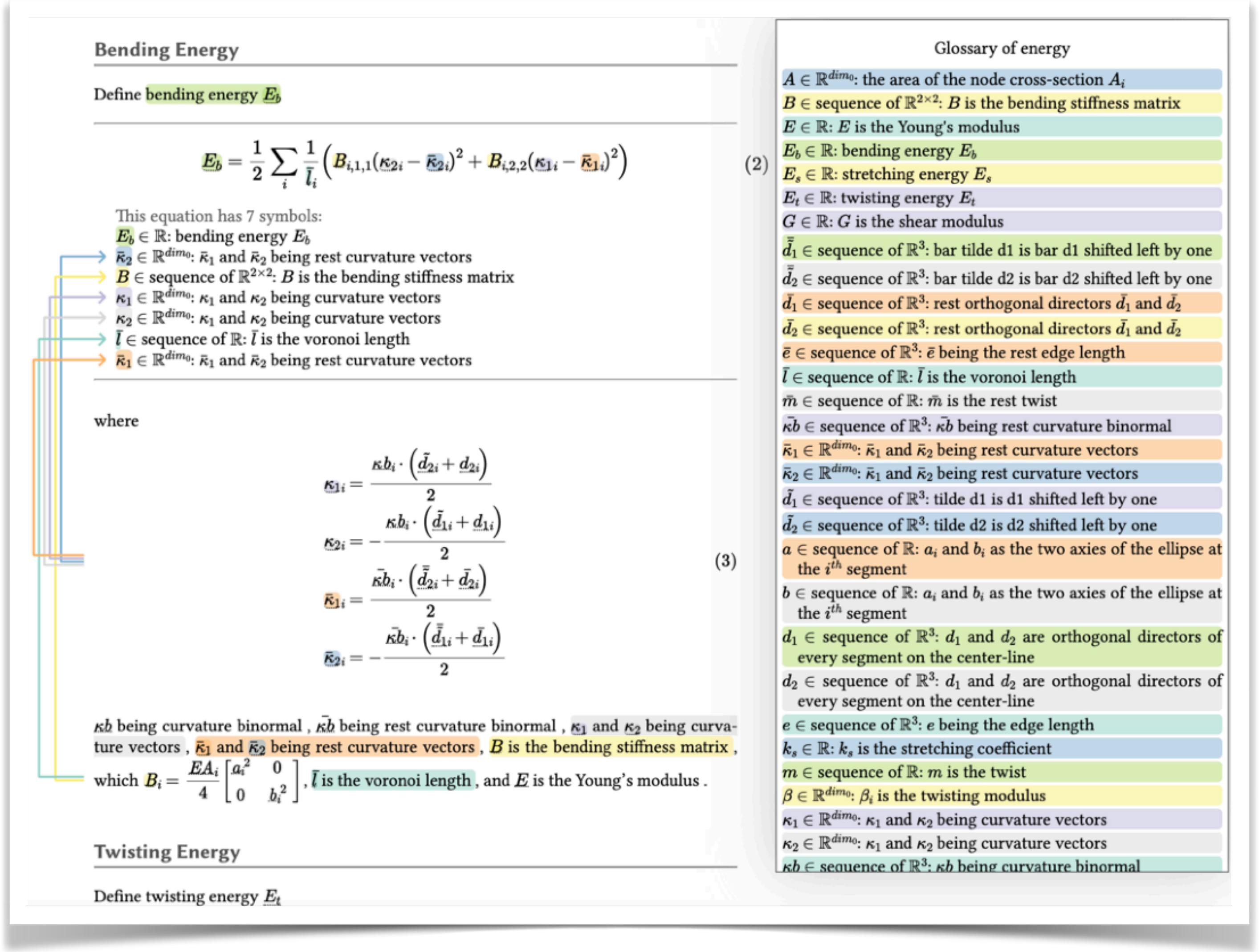
$f \in \mathbb{R}^{n\times 3} \to \mathbb{R}^{m\times 3}$: f maps V to V_o

$m \in \mathbb{Z}$: m is the number of points

$p \in \mathbb{Z}$: a, b, p, q are the indices.

$q \in \mathbb{Z}$: a, b, p, q are the indices.

$t \in \mathbb{R}$: t is energy equals to the sum of E_{len}, E_{par} and E_{per}.



Bending Energy

Define bending energy E_b

$$E_b = \frac{1}{2}\sum_i \frac{1}{\bar{l}_i}\left(B_{i,1,1}(\kappa_{2i} - \bar{\kappa}_{2i})^2 + B_{i,2,2}(\kappa_{1i} - \bar{\kappa}_{1i})^2\right) \qquad (2)$$

This equation has 7 symbols:

$E_b \in \mathbb{R}$: bending energy E_b

$\bar{\kappa}_2 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

$B \in$ sequence of $\mathbb{R}^{2\times 2}$: B is the bending stiffness matrix

$\kappa_1 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors

$\kappa_2 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors

$\bar{l} \in$ sequence of $\mathbb{R}$: $\bar{l}$ is the voronoi length

$\bar{\kappa}_1 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

where

$$\begin{aligned} \kappa_{1i} &= \frac{\kappa b_i \cdot \left(\tilde{d}_{2i} + d_{2i}\right)}{2} \\ \kappa_{2i} &= -\frac{\kappa b_i \cdot \left(\tilde{d}_{1i} + d_{1i}\right)}{2} \\ \bar{\kappa}_{1i} &= \frac{\bar{\kappa b}_i \cdot \left(\bar{\tilde{d}}_{2i} + \bar{d}_{2i}\right)}{2} \\ \bar{\kappa}_{2i} &= -\frac{\bar{\kappa b}_i \cdot \left(\bar{\tilde{d}}_{1i} + \bar{d}_{1i}\right)}{2} \end{aligned} \qquad (3)$$

κb being curvature binormal , $\bar{\kappa b}$ being rest curvature binormal , κ_1 and κ_2 being curvature vectors , $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors , B is the bending stiffness matrix , which $B_i = \frac{EA_i}{4}\begin{bmatrix} a_i^2 & 0 \\ 0 & b_i^2 \end{bmatrix}$, $\bar{l}$ is the voronoi length, and E is the Young's modulus .

Twisting Energy

Define twisting energy E_t

Glossary of energy

$A \in \mathbb{R}^{dim_0}$: the area of the node cross-section A_i

$B \in$ sequence of $\mathbb{R}^{2\times 2}$: B is the bending stiffness matrix

$E \in \mathbb{R}$: E is the Young's modulus

$E_b \in \mathbb{R}$: bending energy E_b

$E_s \in \mathbb{R}$: stretching energy E_s

$E_t \in \mathbb{R}$: twisting energy E_t

$G \in \mathbb{R}$: G is the shear modulus

$\bar{\tilde{d}}_1 \in$ sequence of $\mathbb{R}^3$: bar tilde d1 is bar d1 shifted left by one

$\bar{\tilde{d}}_2 \in$ sequence of $\mathbb{R}^3$: bar tilde d2 is bar d2 shifted left by one

$\bar{d}_1 \in$ sequence of $\mathbb{R}^3$: rest orthogonal directors $\bar{d}_1$ and $\bar{d}_2$

$\bar{d}_2 \in$ sequence of $\mathbb{R}^3$: rest orthogonal directors $\bar{d}_1$ and $\bar{d}_2$

$\bar{e} \in$ sequence of $\mathbb{R}^3$: $\bar{e}$ being the rest edge length

$\bar{l} \in$ sequence of $\mathbb{R}$: $\bar{l}$ is the voronoi length

$\bar{m} \in$ sequence of $\mathbb{R}$: $\bar{m}$ is the rest twist

$\bar{\kappa b} \in$ sequence of $\mathbb{R}^3$: $\bar{\kappa b}$ being rest curvature binormal

$\bar{\kappa}_1 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

$\bar{\kappa}_2 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

$\tilde{d}_1 \in$ sequence of $\mathbb{R}^3$: tilde d1 is d1 shifted left by one

$\tilde{d}_2 \in$ sequence of $\mathbb{R}^3$: tilde d2 is d2 shifted left by one

$a \in$ sequence of $\mathbb{R}$: a_i and b_i as the two axies of the ellipse at the i^{th} segment

$b \in$ sequence of $\mathbb{R}$: a_i and b_i as the two axies of the ellipse at the i^{th} segment

$d_1 \in$ sequence of $\mathbb{R}^3$: d_1 and d_2 are orthogonal directors of every segment on the center-line

$d_2 \in$ sequence of $\mathbb{R}^3$: d_1 and d_2 are orthogonal directors of every segment on the center-line

$e \in$ sequence of $\mathbb{R}^3$: e being the edge length

$k_s \in \mathbb{R}$: k_s is the stretching coefficient

$m \in$ sequence of $\mathbb{R}$: m is the twist

$\beta \in \mathbb{R}^{dim_0}$: β_i is the twisting modulus

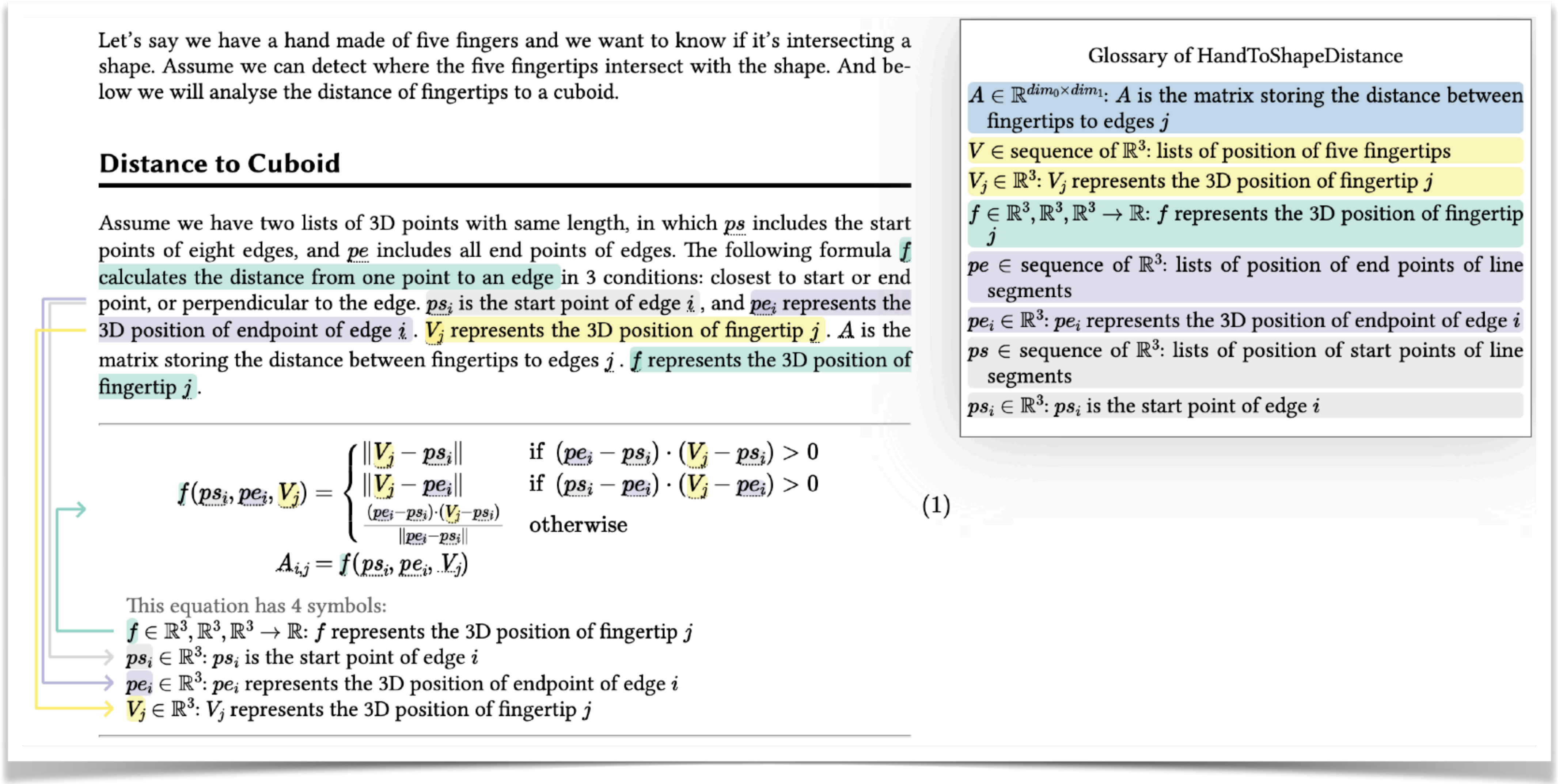
$\kappa_1 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors

$\kappa_2 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors

$\kappa b \in$ sequence of $\mathbb{R}^3$: κb being curvature binormal

Expert Study

- 3 CS PhD students
- Author an original document related to their computer graphics research



Let's say we have a hand made of five fingers and we want to know if it's intersecting a shape. Assume we can detect where the five fingertips intersect with the shape. And below we will analyse the distance of fingertips to a cuboid.

Distance to Cuboid

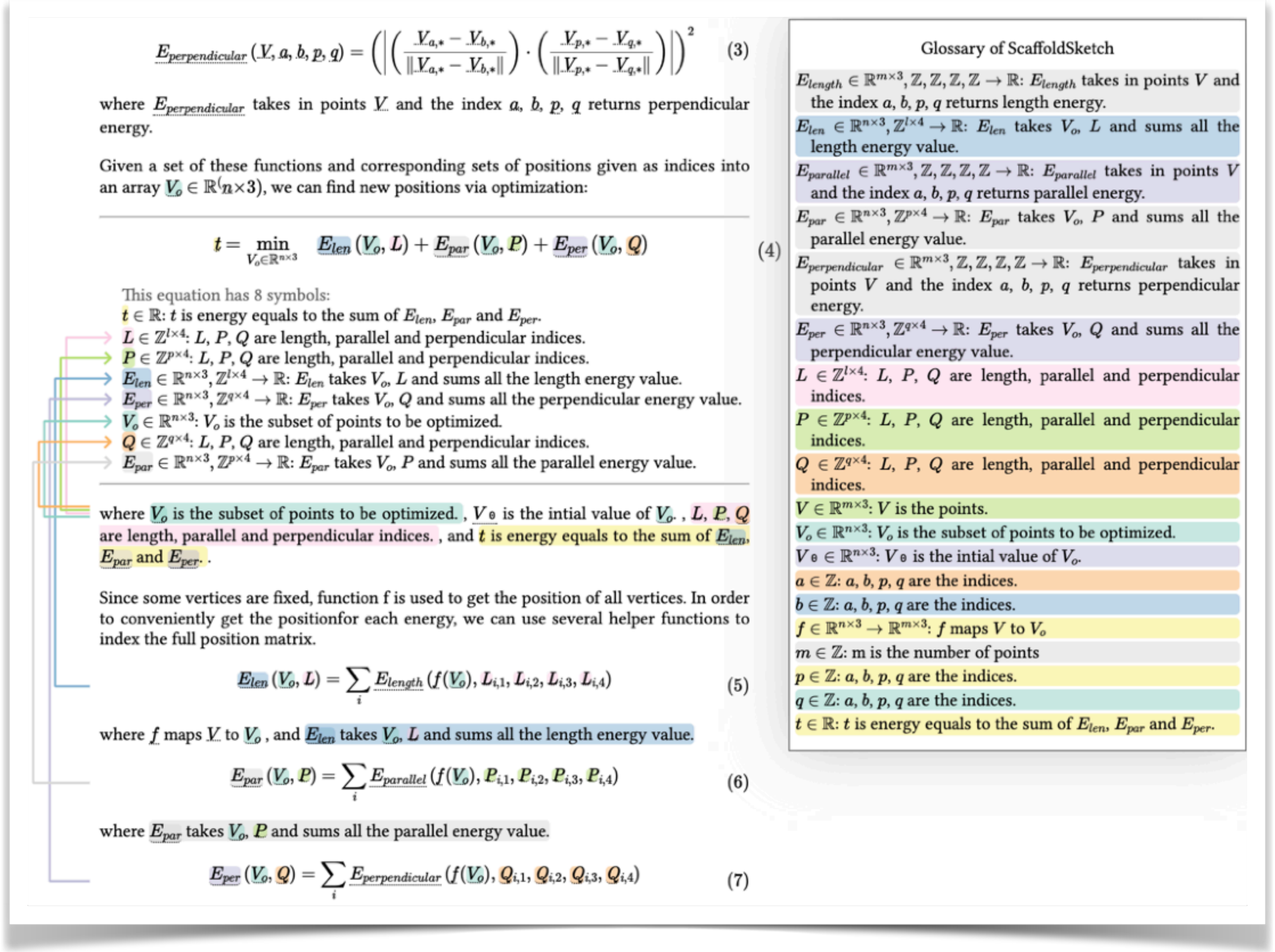
Assume we have two lists of 3D points with same length, in which ps includes the start points of eight edges, and pe includes all end points of edges. The following formula f calculates the distance from one point to an edge in 3 conditions: closest to start or end point, or perpendicular to the edge. ps_i is the start point of edge i , and pe_i represents the 3D position of endpoint of edge i . V_j represents the 3D position of fingertip j . A is the matrix storing the distance between fingertips to edges j . f represents the 3D position of fingertip j .

$$f(ps_i, pe_i, V_j) = \begin{cases} \|V_j - ps_i\| & \text{if } (pe_i - ps_i)\cdot(V_j - ps_i) > 0 \\ \|V_j - pe_i\| & \text{if } (ps_i - pe_i)\cdot(V_j - pe_i) > 0 \\ \frac{(pe_i - ps_i)\cdot(V_j - ps_i)}{\|pe_i - ps_i\|} & \text{otherwise} \end{cases}$$
$$A_{i,j} = f(ps_i, pe_i, V_j) \tag{1}$$

This equation has 4 symbols:
$f \in \mathbb{R}^3, \mathbb{R}^3, \mathbb{R}^3 \to \mathbb{R}$: f represents the 3D position of fingertip j
$ps_i \in \mathbb{R}^3$: ps_i is the start point of edge i
$pe_i \in \mathbb{R}^3$: pe_i represents the 3D position of endpoint of edge i
$V_j \in \mathbb{R}^3$: V_j represents the 3D position of fingertip j

Glossary of HandToShapeDistance

$A \in \mathbb{R}^{dim_0 \times dim_1}$: A is the matrix storing the distance between fingertips to edges j
$V \in$ sequence of $\mathbb{R}^3$: lists of position of five fingertips
$V_j \in \mathbb{R}^3$: V_j represents the 3D position of fingertip j
$f \in \mathbb{R}^3, \mathbb{R}^3, \mathbb{R}^3 \to \mathbb{R}$: f represents the 3D position of fingertip j
$pe \in$ sequence of $\mathbb{R}^3$: lists of position of end points of line segments
$pe_i \in \mathbb{R}^3$: pe_i represents the 3D position of endpoint of edge i
$ps \in$ sequence of $\mathbb{R}^3$: lists of position of start points of line segments
$ps_i \in \mathbb{R}^3$: ps_i is the start point of edge i



$$E_{perpendicular}(V, a, b, p, q) = \left(\left|\left(\frac{V_{a,*} - V_{b,*}}{\|V_{a,*} - V_{b,*}\|}\right) \cdot \left(\frac{V_{p,*} - V_{q,*}}{\|V_{p,*} - V_{q,*}\|}\right)\right|\right)^2 \tag{3}$$

where $E_{perpendicular}$ takes in points V and the index a, b, p, q returns perpendicular energy.

Given a set of these functions and corresponding sets of positions given as indices into an array $V_o \in \mathbb{R}^(n \times 3)$, we can find new positions via optimization:

$$t = \min_{V_o \in \mathbb{R}^{n\times3}} E_{len}(V_o, L) + E_{par}(V_o, P) + E_{per}(V_o, Q) \tag{4}$$

This equation has 8 symbols:
$t \in \mathbb{R}$: t is energy equals to the sum of E_{len}, E_{par} and E_{per}.
$L \in \mathbb{Z}^{l\times4}$: L, P, Q are length, parallel and perpendicular indices.
$P \in \mathbb{Z}^{p\times4}$: L, P, Q are length, parallel and perpendicular indices.
$E_{len} \in \mathbb{R}^{n\times3}, \mathbb{Z}^{l\times4} \to \mathbb{R}$: E_{len} takes V_o, L and sums all the length energy value.
$E_{per} \in \mathbb{R}^{n\times3}, \mathbb{Z}^{q\times4} \to \mathbb{R}$: E_{per} takes V_o, Q and sums all the perpendicular energy value.
$V_o \in \mathbb{R}^{n\times3}$: V_o is the subset of points to be optimized.
$Q \in \mathbb{Z}^{q\times4}$: L, P, Q are length, parallel and perpendicular indices.
$E_{par} \in \mathbb{R}^{n\times3}, \mathbb{Z}^{p\times4} \to \mathbb{R}$: E_{par} takes V_o, P and sums all the parallel energy value.

where V_o is the subset of points to be optimized. , V θ is the intial value of V_o. , L, P, Q are length, parallel and perpendicular indices. , and t is energy equals to the sum of E_{len}, E_{par} and E_{per}. .

Since some vertices are fixed, function f is used to get the position of all vertices. In order to conveniently get the positionfor each energy, we can use several helper functions to index the full position matrix.

$$E_{len}(V_o, L) = \sum_i E_{length}(f(V_o), L_{i,1}, L_{i,2}, L_{i,3}, L_{i,4}) \tag{5}$$

where f maps V to V_o , and E_{len} takes V_o, L and sums all the length energy value.

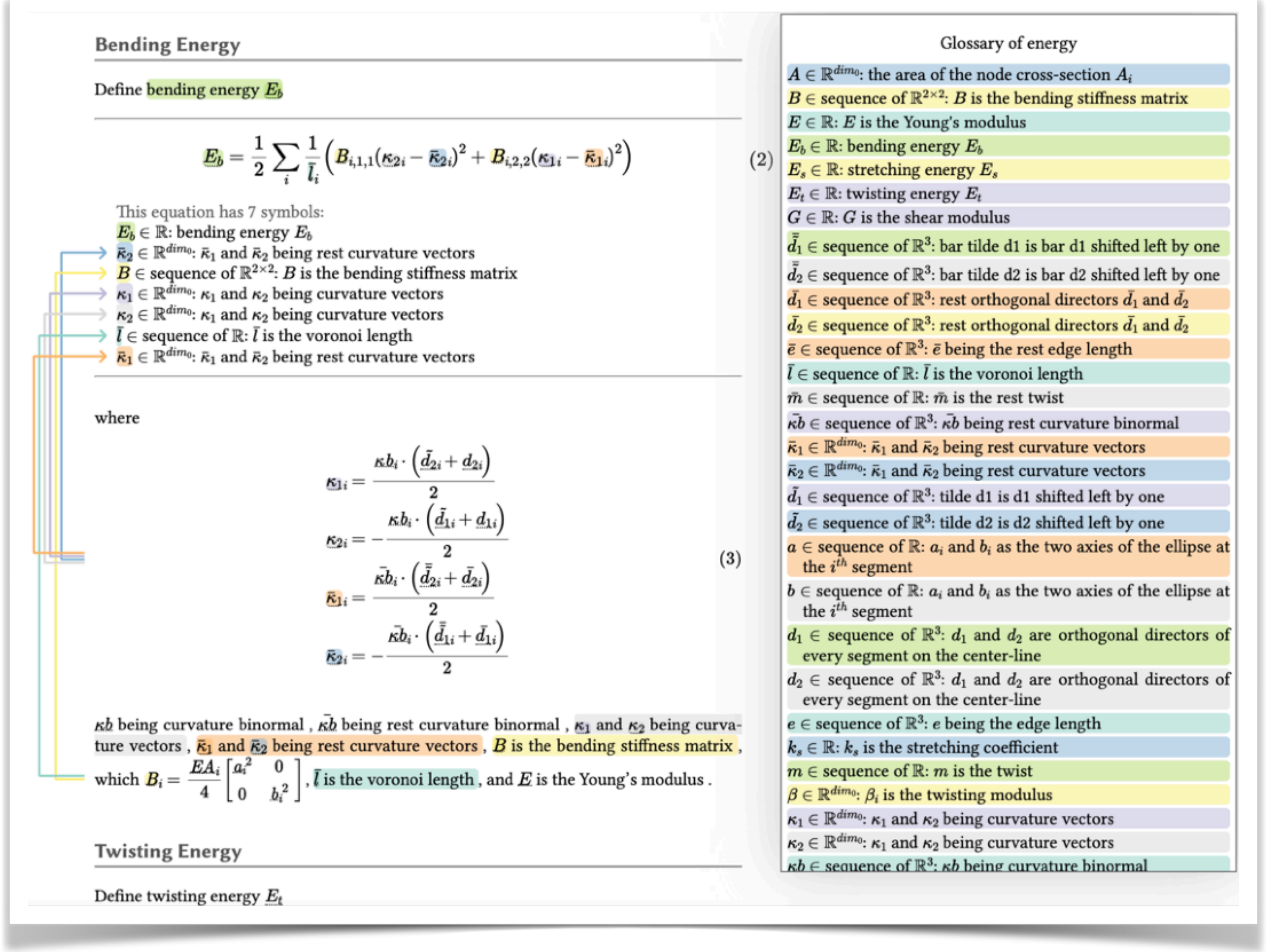
$$E_{par}(V_o, P) = \sum_i E_{parallel}(f(V_o), P_{i,1}, P_{i,2}, P_{i,3}, P_{i,4}) \tag{6}$$

where E_{par} takes V_o, P and sums all the parallel energy value.

$$E_{per}(V_o, Q) = \sum_i E_{perpendicular}(f(V_o), Q_{i,1}, Q_{i,2}, Q_{i,3}, Q_{i,4}) \tag{7}$$

Glossary of ScaffoldSketch

$E_{length} \in \mathbb{R}^{m\times3}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z} \to \mathbb{R}$: E_{length} takes in points V and the index a, b, p, q returns length energy.
$E_{len} \in \mathbb{R}^{n\times3}, \mathbb{Z}^{l\times4} \to \mathbb{R}$: E_{len} takes V_o, L and sums all the length energy value.
$E_{parallel} \in \mathbb{R}^{m\times3}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z} \to \mathbb{R}$: $E_{parallel}$ takes in points V and the index a, b, p, q returns parallel energy.
$E_{par} \in \mathbb{R}^{n\times3}, \mathbb{Z}^{p\times4} \to \mathbb{R}$: E_{par} takes V_o, P and sums all the parallel energy value.
$E_{perpendicular} \in \mathbb{R}^{m\times3}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z}, \mathbb{Z} \to \mathbb{R}$: $E_{perpendicular}$ takes in points V and the index a, b, p, q returns perpendicular energy.
$E_{per} \in \mathbb{R}^{n\times3}, \mathbb{Z}^{q\times4} \to \mathbb{R}$: E_{per} takes V_o, Q and sums all the perpendicular energy value.
$L \in \mathbb{Z}^{l\times4}$: L, P, Q are length, parallel and perpendicular indices.
$P \in \mathbb{Z}^{p\times4}$: L, P, Q are length, parallel and perpendicular indices.
$Q \in \mathbb{Z}^{q\times4}$: L, P, Q are length, parallel and perpendicular indices.
$V \in \mathbb{R}^{m\times3}$: V is the points.
$V_o \in \mathbb{R}^{n\times3}$: V_o is the subset of points to be optimized.
V θ $\in \mathbb{R}^{n\times3}$: V θ is the intial value of V_o.
$a \in \mathbb{Z}$: a, b, p, q are the indices.
$b \in \mathbb{Z}$: a, b, p, q are the indices.
$f \in \mathbb{R}^{n\times3} \to \mathbb{R}^{m\times3}$: f maps V to V_o
$m \in \mathbb{Z}$: m is the number of points
$p \in \mathbb{Z}$: a, b, p, q are the indices.
$q \in \mathbb{Z}$: a, b, p, q are the indices.
$t \in \mathbb{R}$: t is energy equals to the sum of E_{len}, E_{par} and E_{per}.



Bending Energy

Define bending energy E_b

$$E_b = \frac{1}{2}\sum_i \frac{1}{\bar{l}_i}\left(B_{i,1,1}(\kappa_{2i} - \bar{\kappa}_{2i})^2 + B_{i,2,2}(\kappa_{1i} - \bar{\kappa}_{1i})^2\right) \tag{2}$$

This equation has 7 symbols:
$E_b \in \mathbb{R}$: bending energy E_b
$\bar{\kappa}_2 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors
$B \in$ sequence of $\mathbb{R}^{2\times2}$: B is the bending stiffness matrix
$\kappa_1 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors
$\kappa_2 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors
$\bar{l} \in$ sequence of $\mathbb{R}$: $\bar{l}$ is the voronoi length
$\bar{\kappa}_1 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

where

$$\begin{aligned} \kappa_{1i} &= \frac{\kappa b_i \cdot \left(\tilde{d}_{2i} + d_{2i}\right)}{2} \\ \kappa_{2i} &= -\frac{\kappa b_i \cdot \left(\tilde{d}_{1i} + d_{1i}\right)}{2} \\ \bar{\kappa}_{1i} &= \frac{\bar{\kappa b}_i \cdot \left(\bar{\tilde{d}}_{2i} + \bar{d}_{2i}\right)}{2} \\ \bar{\kappa}_{2i} &= -\frac{\bar{\kappa b}_i \cdot \left(\bar{\tilde{d}}_{1i} + \bar{d}_{1i}\right)}{2} \end{aligned} \tag{3}$$

κb being curvature binormal , $\bar{\kappa b}$ being rest curvature binormal , κ_1 and κ_2 being curvature vectors , $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors , B is the bending stiffness matrix , which $B_i = \frac{EA_i}{4}\begin{bmatrix} a_i^2 & 0 \\ 0 & b_i^2 \end{bmatrix}$, $\bar{l}$ is the voronoi length , and E is the Young's modulus .

Twisting Energy

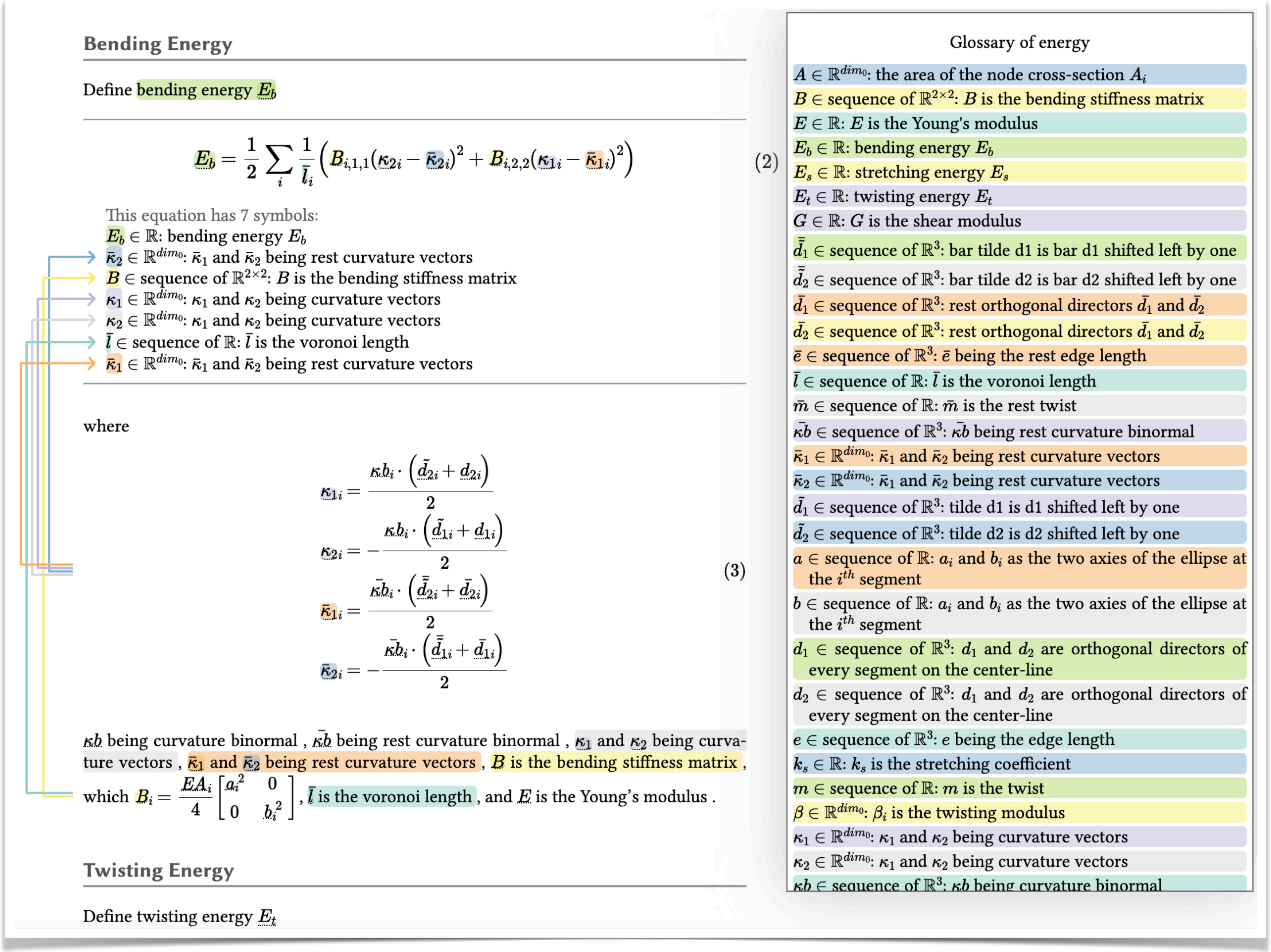
Define twisting energy E_t

Glossary of energy

$A \in \mathbb{R}^{dim_0}$: the area of the node cross-section A_i
$B \in$ sequence of $\mathbb{R}^{2\times2}$: B is the bending stiffness matrix
$E \in \mathbb{R}$: E is the Young's modulus
$E_b \in \mathbb{R}$: bending energy E_b
$E_s \in \mathbb{R}$: stretching energy E_s
$E_t \in \mathbb{R}$: twisting energy E_t
$G \in \mathbb{R}$: G is the shear modulus
$\bar{\tilde{d}}_1 \in$ sequence of $\mathbb{R}^3$: bar tilde d1 is bar d1 shifted left by one
$\bar{\tilde{d}}_2 \in$ sequence of $\mathbb{R}^3$: bar tilde d2 is bar d2 shifted left by one
$\bar{d}_1 \in$ sequence of $\mathbb{R}^3$: rest orthogonal directors $\bar{d}_1$ and $\bar{d}_2$
$\bar{d}_2 \in$ sequence of $\mathbb{R}^3$: rest orthogonal directors $\bar{d}_1$ and $\bar{d}_2$
$\bar{e} \in$ sequence of $\mathbb{R}^3$: $\bar{e}$ being the rest edge length
$\bar{l} \in$ sequence of $\mathbb{R}$: $\bar{l}$ is the voronoi length
$\bar{m} \in$ sequence of $\mathbb{R}$: $\bar{m}$ is the rest twist
$\bar{\kappa b} \in$ sequence of $\mathbb{R}^3$: $\bar{\kappa b}$ being rest curvature binormal
$\bar{\kappa}_1 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors
$\bar{\kappa}_2 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors
$\tilde{d}_1 \in$ sequence of $\mathbb{R}^3$: tilde d1 is d1 shifted left by one
$\tilde{d}_2 \in$ sequence of $\mathbb{R}^3$: tilde d2 is d2 shifted left by one
$a \in$ sequence of $\mathbb{R}$: a_i and b_i as the two axies of the ellipse at the i^{th} segment
$b \in$ sequence of $\mathbb{R}$: a_i and b_i as the two axies of the ellipse at the i^{th} segment
$d_1 \in$ sequence of $\mathbb{R}^3$: d_1 and d_2 are orthogonal directors of every segment on the center-line
$d_2 \in$ sequence of $\mathbb{R}^3$: d_1 and d_2 are orthogonal directors of every segment on the center-line
$e \in$ sequence of $\mathbb{R}^3$: e being the edge length
$k_s \in \mathbb{R}$: k_s is the stretching coefficient
$m \in$ sequence of $\mathbb{R}$: m is the twist
$\beta \in \mathbb{R}^{dim_0}$: β_i is the twisting modulus
$\kappa_1 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors
$\kappa_2 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors
$\kappa b \in$ sequence of $\mathbb{R}^3$: κb being curvature binormal

- Spent 24, 7, and 6 hours, respectively, using H❤️rtDown over a period of two weeks

Expert Study: Observations and Conclusions



Bending Energy

Define bending energy E_b

$$E_b = \frac{1}{2}\sum_i \frac{1}{\bar{l}_i}\left(B_{i,1,1}(\kappa_{2i} - \bar{\kappa}_{2i})^2 + B_{i,2,2}(\kappa_{1i} - \bar{\kappa}_{1i})^2\right) \tag{2}$$

This equation has 7 symbols:

$E_b \in \mathbb{R}$: bending energy E_b

$\bar{\kappa}_2 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

$B \in$ sequence of $\mathbb{R}^{2\times 2}$: B is the bending stiffness matrix

$\kappa_1 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors

$\kappa_2 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors

$\bar{l} \in$ sequence of $\mathbb{R}$: $\bar{l}$ is the voronoi length

$\bar{\kappa}_1 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

where

$$\begin{aligned}
\kappa_{1i} &= \frac{\kappa b_i \cdot \left(\tilde{d}_{2i} + d_{2i}\right)}{2} \\
\kappa_{2i} &= -\frac{\kappa b_i \cdot \left(\tilde{d}_{1i} + d_{1i}\right)}{2} \\
\bar{\kappa}_{1i} &= \frac{\bar{\kappa b}_i \cdot \left(\bar{\tilde{d}}_{2i} + \bar{d}_{2i}\right)}{2} \\
\bar{\kappa}_{2i} &= -\frac{\bar{\kappa b}_i \cdot \left(\bar{\tilde{d}}_{1i} + \bar{d}_{1i}\right)}{2}
\end{aligned} \tag{3}$$

κb being curvature binormal , $\bar{\kappa b}$ being rest curvature binormal , κ_1 and κ_2 being curvature vectors , $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors , B is the bending stiffness matrix , which $B_i = \frac{EA_i}{4}\begin{bmatrix} a_i^2 & 0 \\ 0 & b_i^2 \end{bmatrix}$, $\bar{l}$ is the voronoi length , and E is the Young's modulus .

Twisting Energy

Define twisting energy E_t

Glossary of energy

$A \in \mathbb{R}^{dim_0}$: the area of the node cross-section A_i

$B \in$ sequence of $\mathbb{R}^{2\times 2}$: B is the bending stiffness matrix

$E \in \mathbb{R}$: E is the Young's modulus

$E_b \in \mathbb{R}$: bending energy E_b

$E_s \in \mathbb{R}$: stretching energy E_s

$E_t \in \mathbb{R}$: twisting energy E_t

$G \in \mathbb{R}$: G is the shear modulus

$\bar{\tilde{d}}_1 \in$ sequence of $\mathbb{R}^3$: bar tilde d1 is bar d1 shifted left by one

$\bar{\tilde{d}}_2 \in$ sequence of $\mathbb{R}^3$: bar tilde d2 is bar d2 shifted left by one

$\bar{d}_1 \in$ sequence of $\mathbb{R}^3$: rest orthogonal directors $\bar{d}_1$ and $\bar{d}_2$

$\bar{d}_2 \in$ sequence of $\mathbb{R}^3$: rest orthogonal directors $\bar{d}_1$ and $\bar{d}_2$

$\bar{e} \in$ sequence of $\mathbb{R}^3$: $\bar{e}$ being the rest edge length

$\bar{l} \in$ sequence of $\mathbb{R}$: $\bar{l}$ is the voronoi length

$\bar{m} \in$ sequence of $\mathbb{R}$: $\bar{m}$ is the rest twist

$\bar{\kappa b} \in$ sequence of $\mathbb{R}^3$: $\bar{\kappa b}$ being rest curvature binormal

$\bar{\kappa}_1 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

$\bar{\kappa}_2 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

$\tilde{d}_1 \in$ sequence of $\mathbb{R}^3$: tilde d1 is d1 shifted left by one

$\tilde{d}_2 \in$ sequence of $\mathbb{R}^3$: tilde d2 is d2 shifted left by one

$a \in$ sequence of $\mathbb{R}$: a_i and b_i as the two axies of the ellipse at the i^{th} segment

$b \in$ sequence of $\mathbb{R}$: a_i and b_i as the two axies of the ellipse at the i^{th} segment

$d_1 \in$ sequence of $\mathbb{R}^3$: d_1 and d_2 are orthogonal directors of every segment on the center-line

$d_2 \in$ sequence of $\mathbb{R}^3$: d_1 and d_2 are orthogonal directors of every segment on the center-line

$e \in$ sequence of $\mathbb{R}^3$: e being the edge length

$k_s \in \mathbb{R}$: k_s is the stretching coefficient

$m \in$ sequence of $\mathbb{R}$: m is the twist

$\beta \in \mathbb{R}^{dim_0}$: β_i is the twisting modulus

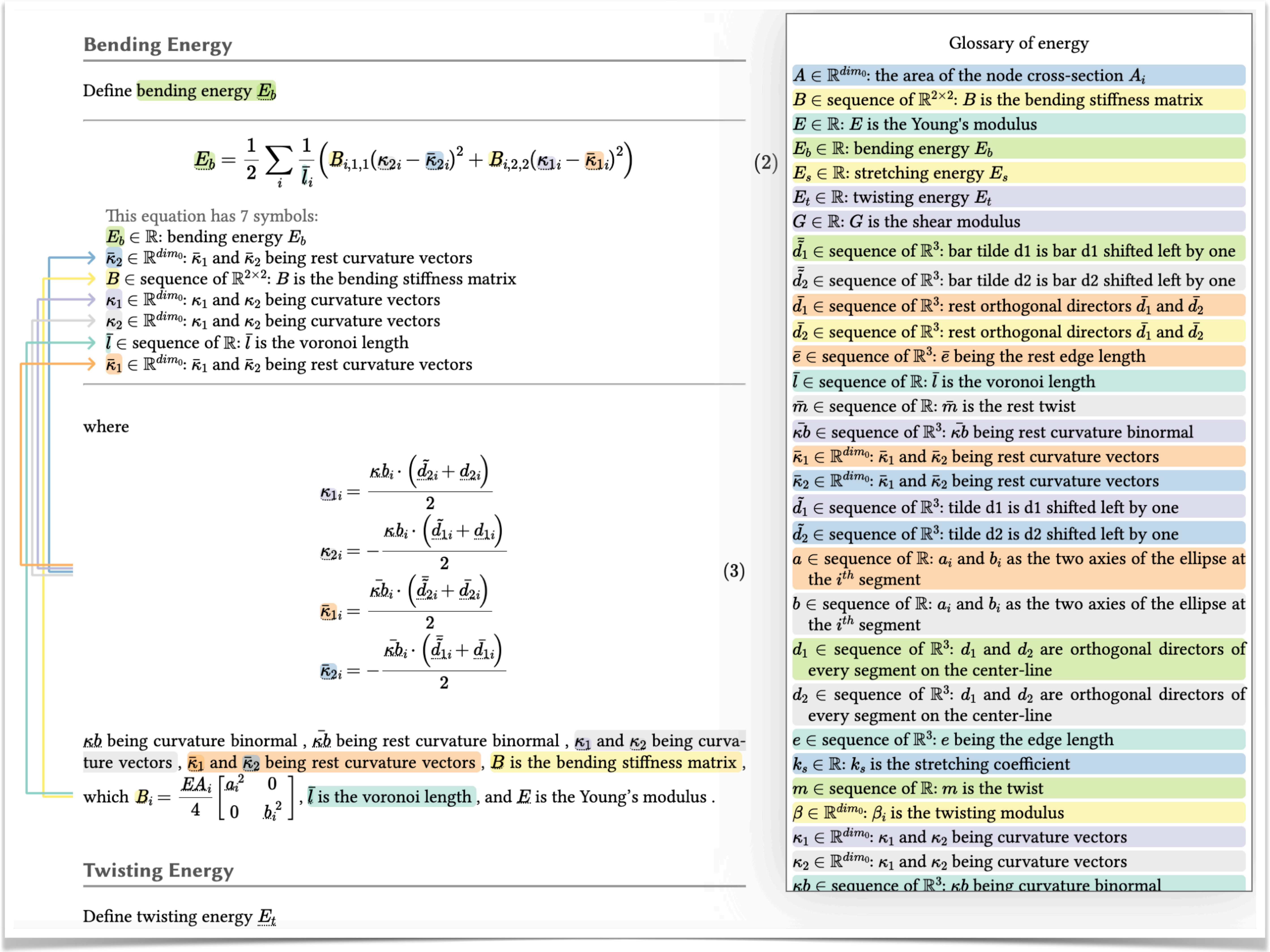
$\kappa_1 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors

$\kappa_2 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors

$\kappa b \in$ sequence of $\mathbb{R}^3$: κb being curvature binormal

Expert Study: Observations and Conclusions

“H❤️rtDown is an excellent tool to share tutorial[s] online—it highlights the vector dimension and variable meaning…following all the vectors/matrices/their dims is ***the hardest part*** *of reproducing a paper.”*



Bending Energy

Define bending energy E_b

$$E_b = \frac{1}{2}\sum_i \frac{1}{\bar{l}_i}\left(B_{i,1,1}(\kappa_{2i} - \bar{\kappa}_{2i})^2 + B_{i,2,2}(\kappa_{1i} - \bar{\kappa}_{1i})^2\right) \qquad (2)$$

This equation has 7 symbols:

$E_b \in \mathbb{R}$: bending energy E_b

$\bar{\kappa}_2 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

$B \in$ sequence of $\mathbb{R}^{2\times2}$: B is the bending stiffness matrix

$\kappa_1 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors

$\kappa_2 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors

$\bar{l} \in$ sequence of $\mathbb{R}$: $\bar{l}$ is the voronoi length

$\bar{\kappa}_1 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

where

$$\begin{aligned}\kappa_{1i} &= \frac{\kappa b_i \cdot \left(\tilde{d}_{2i} + d_{2i}\right)}{2}\\ \kappa_{2i} &= -\frac{\kappa b_i \cdot \left(\tilde{d}_{1i} + d_{1i}\right)}{2}\\ \bar{\kappa}_{1i} &= \frac{\bar{\kappa b}_i \cdot \left(\bar{\tilde{d}}_{2i} + \bar{d}_{2i}\right)}{2}\\ \bar{\kappa}_{2i} &= -\frac{\bar{\kappa b}_i \cdot \left(\bar{\tilde{d}}_{1i} + \bar{d}_{1i}\right)}{2}\end{aligned} \qquad (3)$$

κb being curvature binormal , $\bar{\kappa b}$ being rest curvature binormal , κ_1 and κ_2 being curvature vectors , $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors , B is the bending stiffness matrix , which $B_i = \frac{EA_i}{4}\begin{bmatrix} a_i^2 & 0 \\ 0 & b_i^2 \end{bmatrix}$, $\bar{l}$ is the voronoi length , and E is the Young’s modulus .

Twisting Energy

Define twisting energy E_t

Glossary of energy

$A \in \mathbb{R}^{dim_0}$: the area of the node cross-section A_i

$B \in$ sequence of $\mathbb{R}^{2\times2}$: B is the bending stiffness matrix

$E \in \mathbb{R}$: E is the Young's modulus

$E_b \in \mathbb{R}$: bending energy E_b

$E_s \in \mathbb{R}$: stretching energy E_s

$E_t \in \mathbb{R}$: twisting energy E_t

$G \in \mathbb{R}$: G is the shear modulus

$\bar{\tilde{d}}_1 \in$ sequence of $\mathbb{R}^3$: bar tilde d1 is bar d1 shifted left by one

$\bar{\tilde{d}}_2 \in$ sequence of $\mathbb{R}^3$: bar tilde d2 is bar d2 shifted left by one

$\bar{d}_1 \in$ sequence of $\mathbb{R}^3$: rest orthogonal directors $\bar{d}_1$ and $\bar{d}_2$

$\bar{d}_2 \in$ sequence of $\mathbb{R}^3$: rest orthogonal directors $\bar{d}_1$ and $\bar{d}_2$

$\bar{e} \in$ sequence of $\mathbb{R}^3$: $\bar{e}$ being the rest edge length

$\bar{l} \in$ sequence of $\mathbb{R}$: $\bar{l}$ is the voronoi length

$\bar{m} \in$ sequence of $\mathbb{R}$: $\bar{m}$ is the rest twist

$\bar{\kappa b} \in$ sequence of $\mathbb{R}^3$: $\bar{\kappa b}$ being rest curvature binormal

$\bar{\kappa}_1 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

$\bar{\kappa}_2 \in \mathbb{R}^{dim_0}$: $\bar{\kappa}_1$ and $\bar{\kappa}_2$ being rest curvature vectors

$\tilde{d}_1 \in$ sequence of $\mathbb{R}^3$: tilde d1 is d1 shifted left by one

$\tilde{d}_2 \in$ sequence of $\mathbb{R}^3$: tilde d2 is d2 shifted left by one

$a \in$ sequence of $\mathbb{R}$: a_i and b_i as the two axies of the ellipse at the i^{th} segment

$b \in$ sequence of $\mathbb{R}$: a_i and b_i as the two axies of the ellipse at the i^{th} segment

$d_1 \in$ sequence of $\mathbb{R}^3$: d_1 and d_2 are orthogonal directors of every segment on the center-line

$d_2 \in$ sequence of $\mathbb{R}^3$: d_1 and d_2 are orthogonal directors of every segment on the center-line

$e \in$ sequence of $\mathbb{R}^3$: e being the edge length

$k_s \in \mathbb{R}$: k_s is the stretching coefficient

$m \in$ sequence of $\mathbb{R}$: m is the twist

$\beta \in \mathbb{R}^{dim_0}$: β_i is the twisting modulus

$\kappa_1 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors

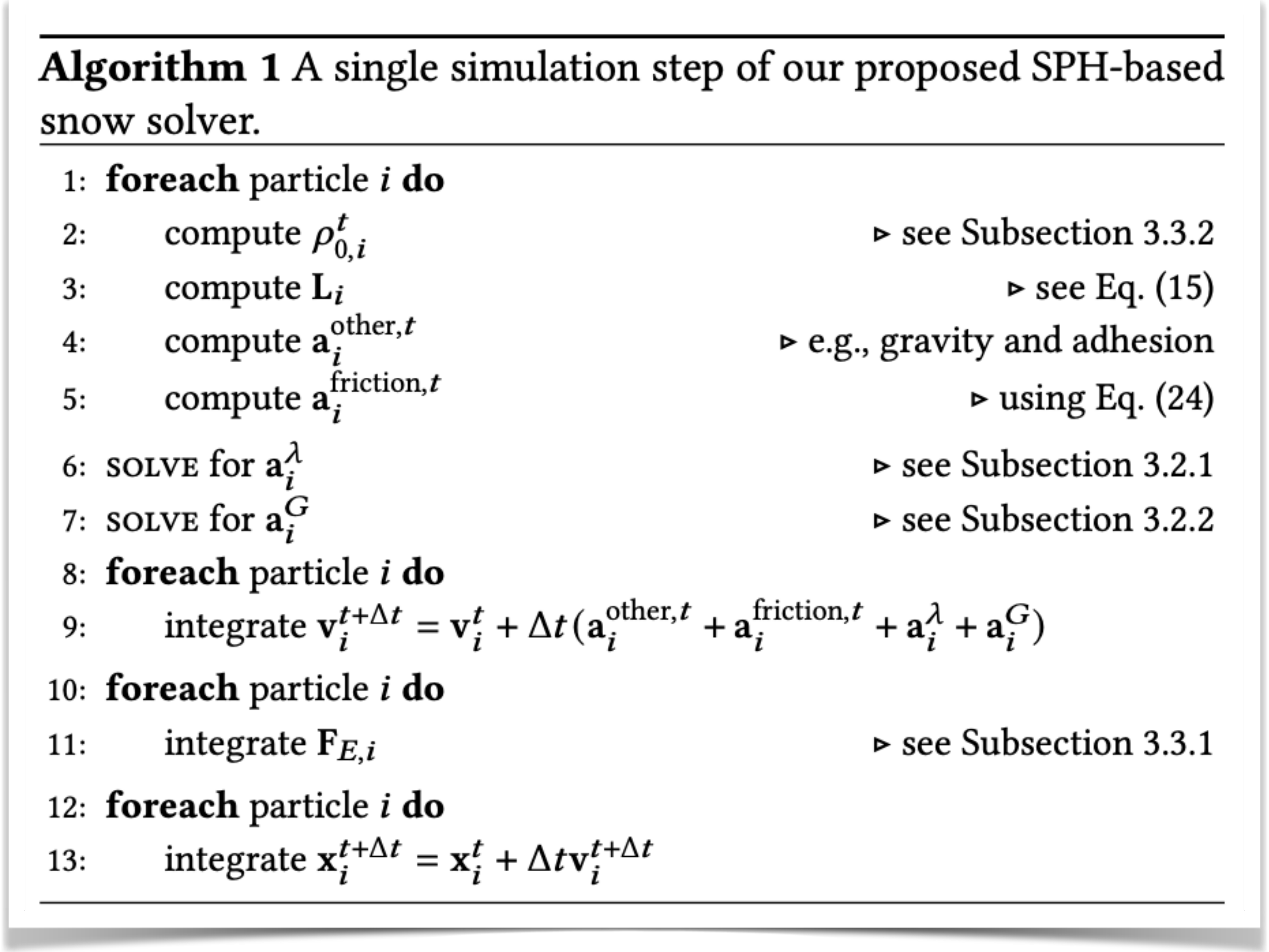
$\kappa_2 \in \mathbb{R}^{dim_0}$: κ_1 and κ_2 being curvature vectors

$\kappa b \in$ sequence of $\mathbb{R}^3$: κb being curvature binormal

Limitations

Limitations

- H❤️rtDown does not consider pseudocode or algorithmic steps described in prose



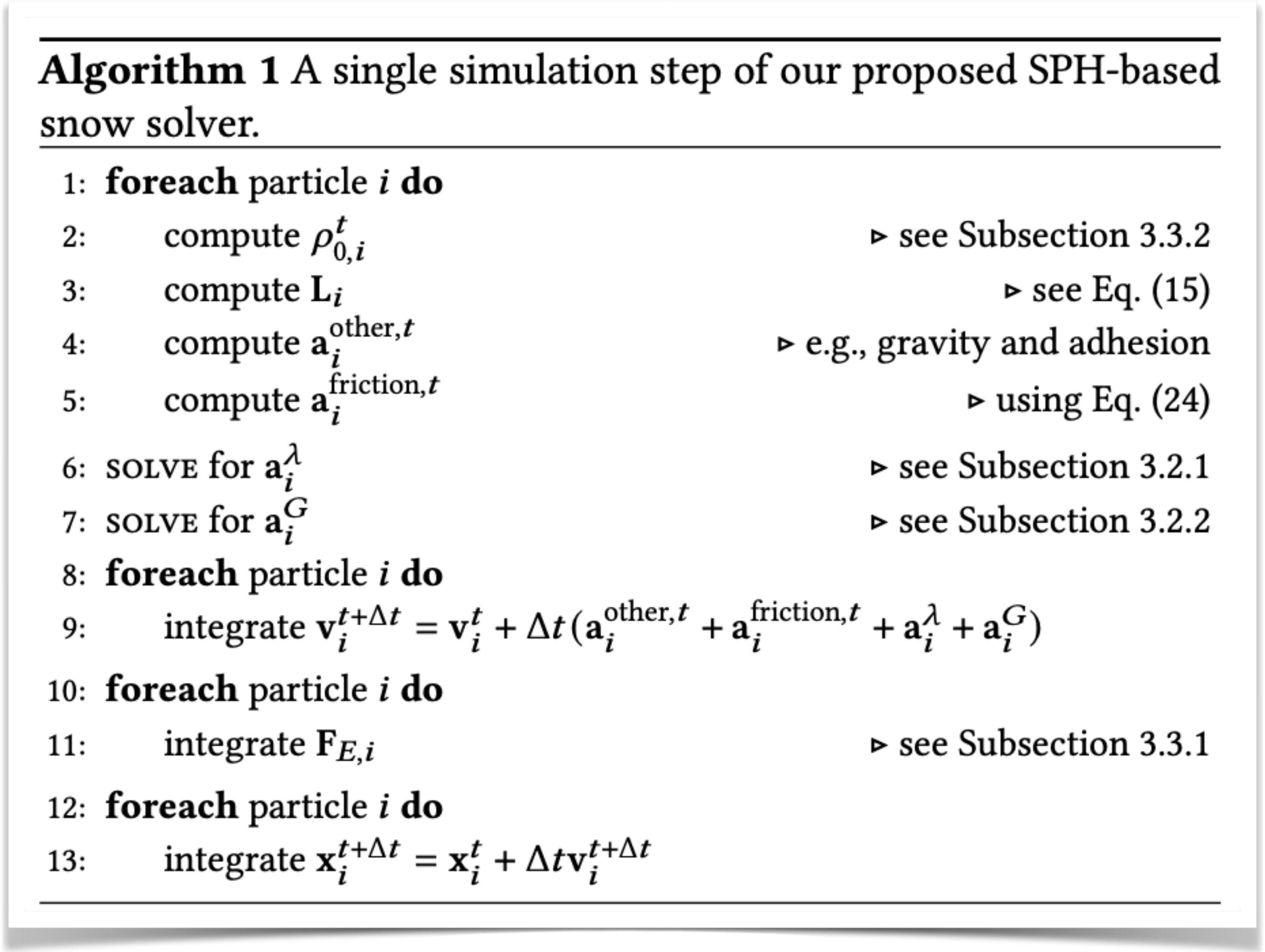
Algorithm 1 A single simulation step of our proposed SPH-based snow solver.

```
foreach particle $i$ do
    compute $\rho_{0,i}^t$                                      ▹ see Subsection 3.3.2
    compute $\mathbf{L}_i$                                       ▹ see Eq. (15)
    compute $\mathbf{a}_i^{\text{other},t}$                      ▹ e.g., gravity and adhesion
    compute $\mathbf{a}_i^{\text{friction},t}$                   ▹ using Eq. (24)
SOLVE for $\mathbf{a}_i^\lambda$                                ▹ see Subsection 3.2.1
SOLVE for $\mathbf{a}_i^G$                                      ▹ see Subsection 3.2.2
foreach particle $i$ do
    integrate $\mathbf{v}_i^{t+\Delta t} = \mathbf{v}_i^t + \Delta t(\mathbf{a}_i^{\text{other},t} + \mathbf{a}_i^{\text{friction},t} + \mathbf{a}_i^\lambda + \mathbf{a}_i^G)$
foreach particle $i$ do
    integrate $\mathbf{F}_{E,i}$                                 ▹ see Subsection 3.3.1
foreach particle $i$ do
    integrate $\mathbf{x}_i^{t+\Delta t} = \mathbf{x}_i^t + \Delta t \mathbf{v}_i^{t+\Delta t}$
```

[Gissler et al. 2020]

Limitations

- H❤️rtDown does not consider pseudocode or algorithmic steps described in prose



Algorithm 1 A single simulation step of our proposed SPH-based snow solver.

```
foreach particle $i$ do
    compute $\rho^t_{0,i}$                                        ▷ see Subsection 3.3.2
    compute $\mathbf{L}_i$                                        ▷ see Eq. (15)
    compute $\mathbf{a}_i^{\text{other},t}$                       ▷ e.g., gravity and adhesion
    compute $\mathbf{a}_i^{\text{friction},t}$                    ▷ using Eq. (24)
SOLVE for $\mathbf{a}_i^{\lambda}$                                ▷ see Subsection 3.2.1
SOLVE for $\mathbf{a}_i^{G}$                                      ▷ see Subsection 3.2.2
foreach particle $i$ do
    integrate $\mathbf{v}_i^{t+\Delta t} = \mathbf{v}_i^t + \Delta t(\mathbf{a}_i^{\text{other},t} + \mathbf{a}_i^{\text{friction},t} + \mathbf{a}_i^{\lambda} + \mathbf{a}_i^{G})$
foreach particle $i$ do
    integrate $\mathbf{F}_{E,i}$                                  ▷ see Subsection 3.3.1
foreach particle $i$ do
    integrate $\mathbf{x}_i^{t+\Delta t} = \mathbf{x}_i^t + \Delta t \mathbf{v}_i^{t+\Delta t}$
```

[Gissler et al. 2020]

- The space of executable math and potential application domains for H❤️rtDown is much broader than linear algebra

Future Work

Future Work

- Automatic conversion from LaTeX to H❤️rtDown

Future Work

- Automatic conversion from LaTeX to H❤️rtDown
- A proof checker to verify derivations

Future Work

- Automatic conversion from LaTeX to H❤️rtDown
- A proof checker to verify derivations
- Callbacks and delegates for expanding the abilities of the generated code

Future Work

- Automatic conversion from LaTeX to H❤️rtDown
- A proof checker to verify derivations
- Callbacks and delegates for expanding the abilities of the generated code
- Support for active reading (e.g. annotating and comparing)

Conclusions

Conclusions

- H❤️rtDown is a low-overhead, ecologically compatible document processor

Conclusions

- H❤️rtDown is a low-overhead, ecologically compatible document processor

- H❤️rtDown supports authors and improves replicability, readability, and experimentation

Conclusions

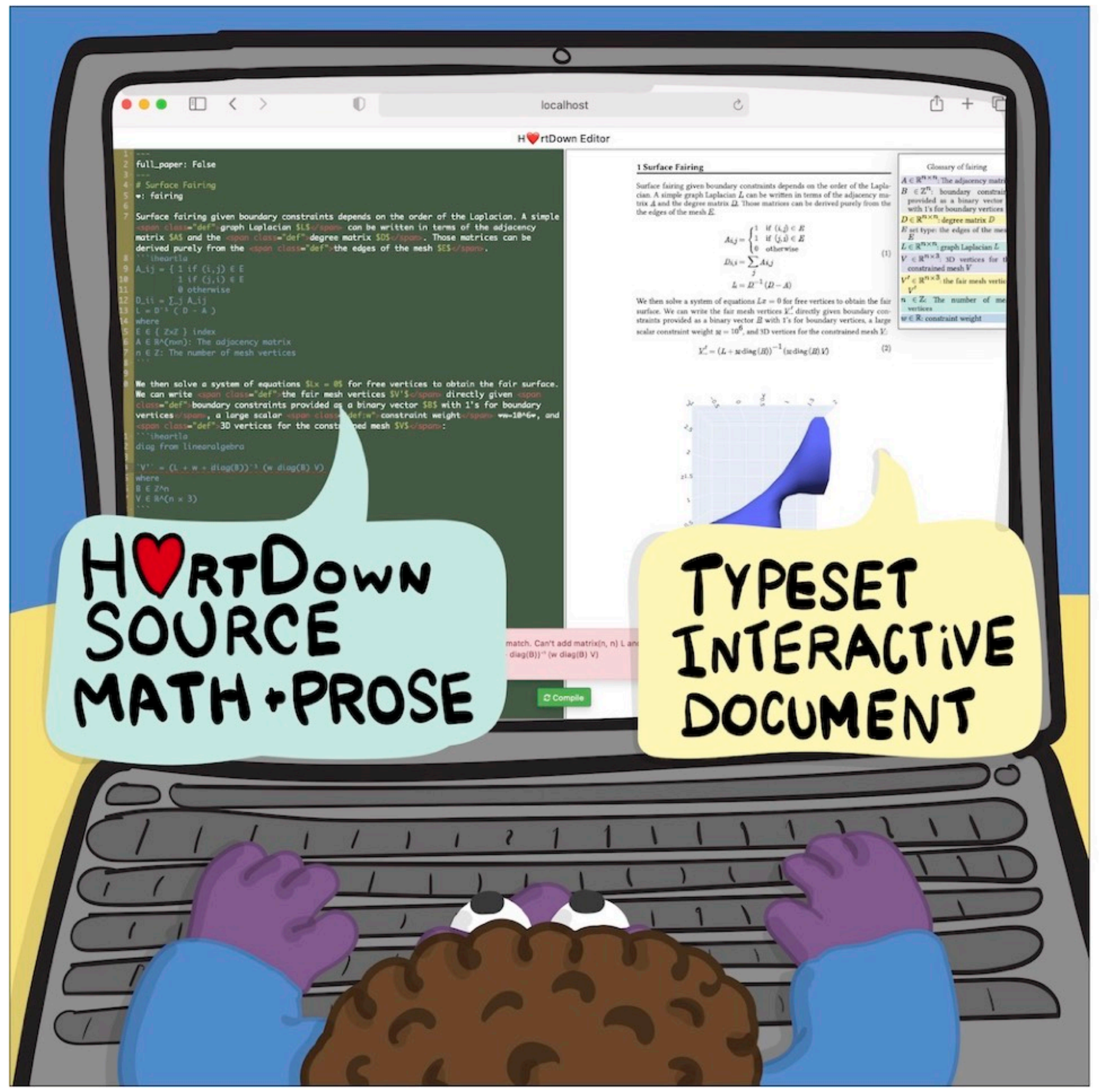
- H❤️rtDown is a low-overhead, ecologically compatible document processor

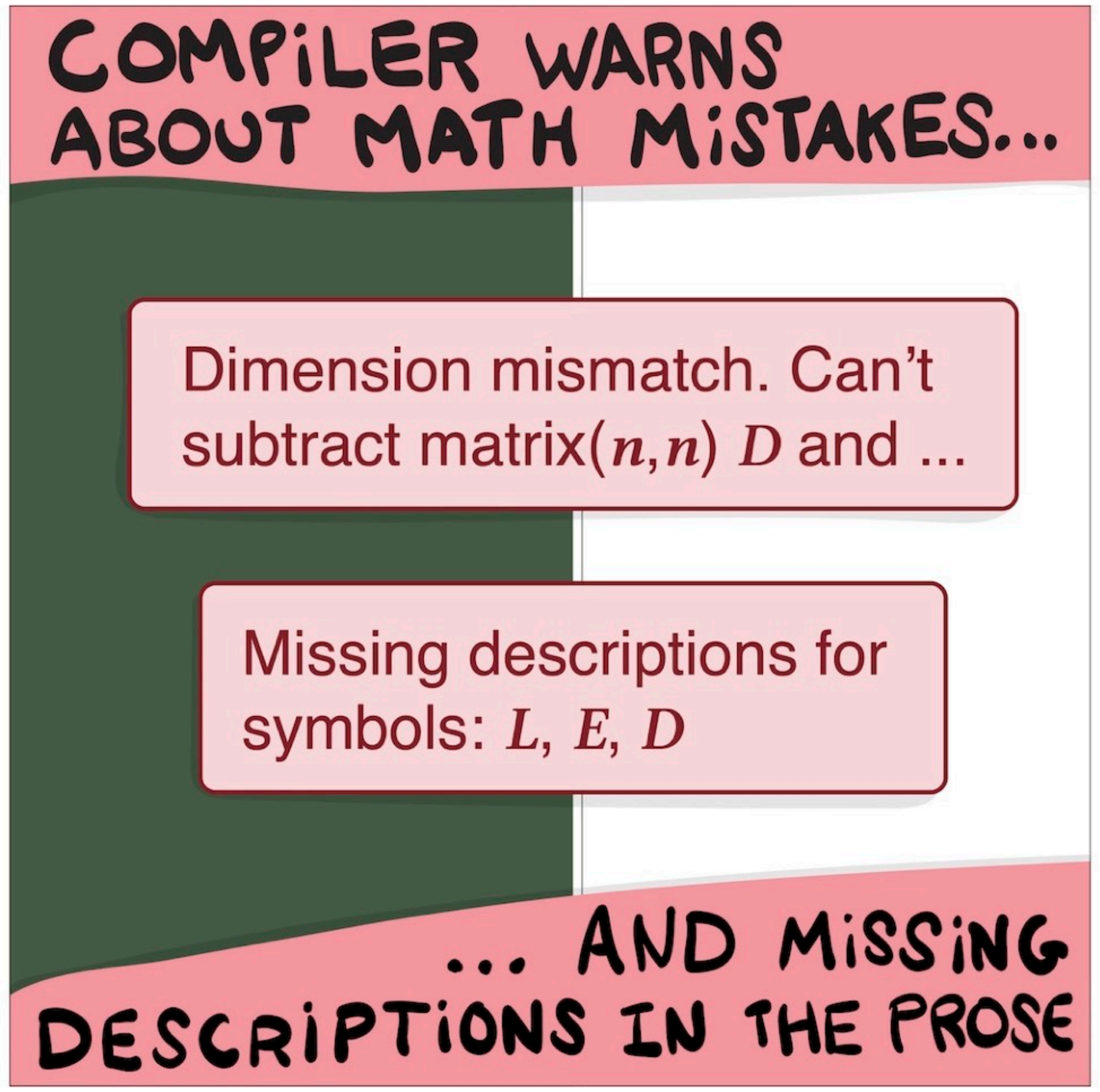
- H❤️rtDown supports authors and improves replicability, readability, and experimentation

- Participants in our expert study found uses for H❤️rtDown in their research practice.

H❤️rtDown

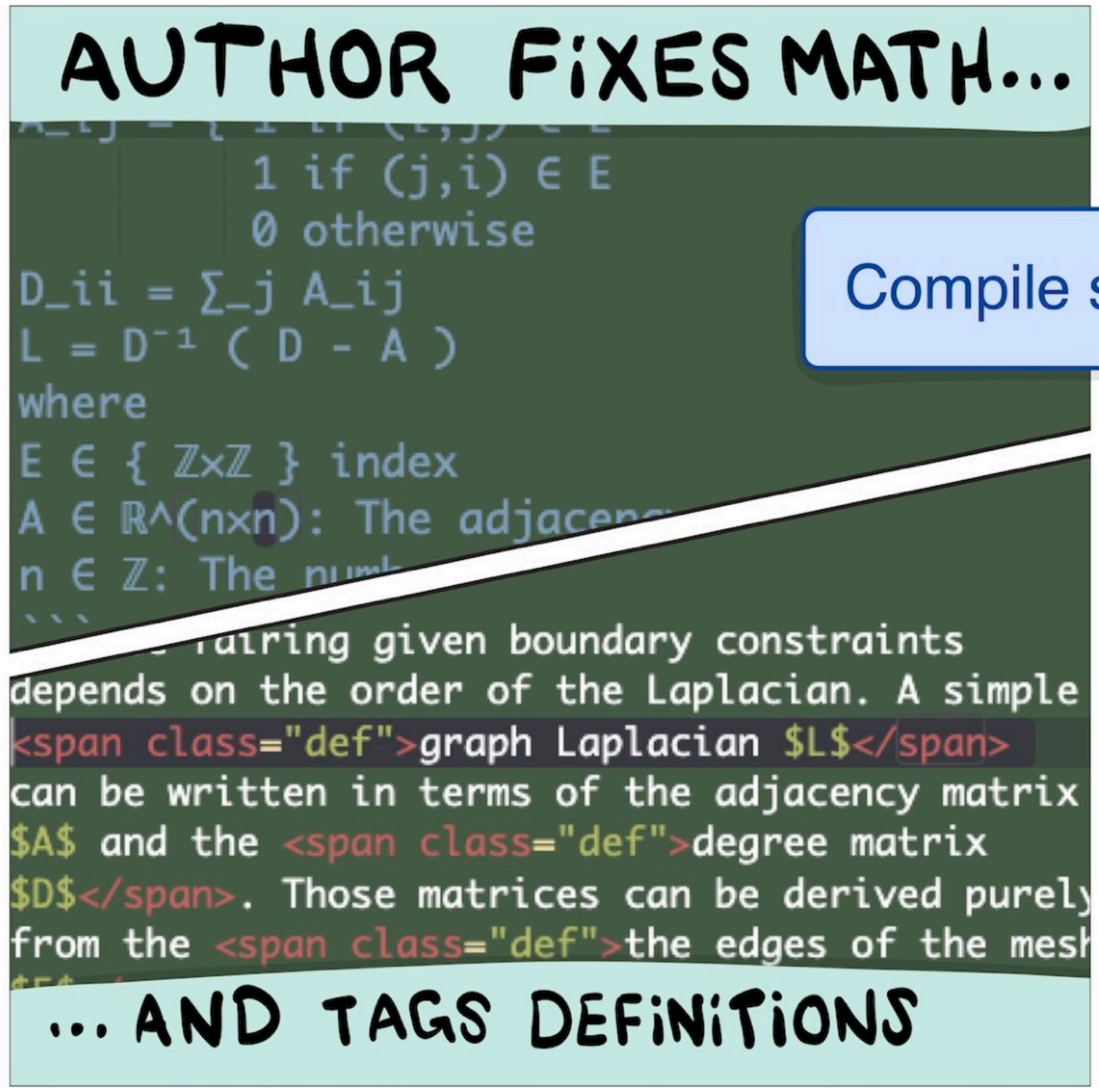
https://iheartla.github.io/heartdown/




COMPILER WARNS ABOUT MATH MISTAKES...
... AND MISSING DESCRIPTIONS IN THE PROSE

Dimension mismatch. Can't subtract matrix(n,n) D and ...

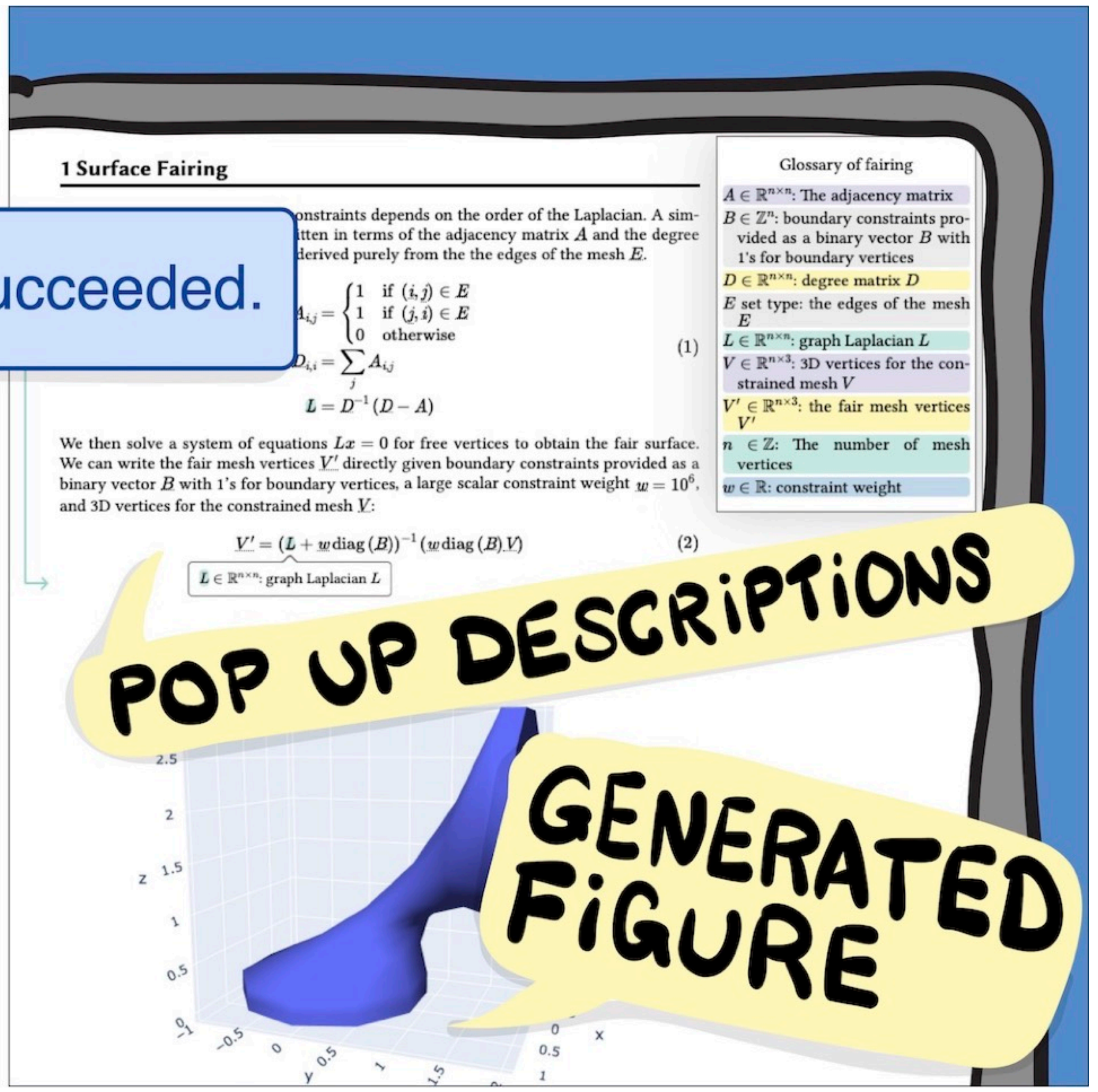
Missing descriptions for symbols: L, E, D



Compile succeeded.

```
        1 if (j,i) ∈ E
        0 otherwise
D_ii = ∑_j A_ij
L = D⁻¹ ( D - A )
where
E ∈ { ℤxℤ } index
A ∈ ℝ^(nxn): The adjacen
n ∈ ℤ: The num
```

```
...airing given boundary constraints
depends on the order of the Laplacian. A simple
<span class="def">graph Laplacian $L$</span>
can be written in terms of the adjacency matrix
$A$ and the <span class="def">degree matrix
$D$</span>. Those matrices can be derived purely
from the <span class="def">the edges of the mesh
```



Acknowledgements: Anonymous reviewers, Seth Walker, Zoya Bylinskii, Zhecheng Wang, Xue Yu, Jialin Huang

Sponsors: Canada Research Chairs Program, Sloan Foundation, Adobe Inc.