LIVE CODING SESSION

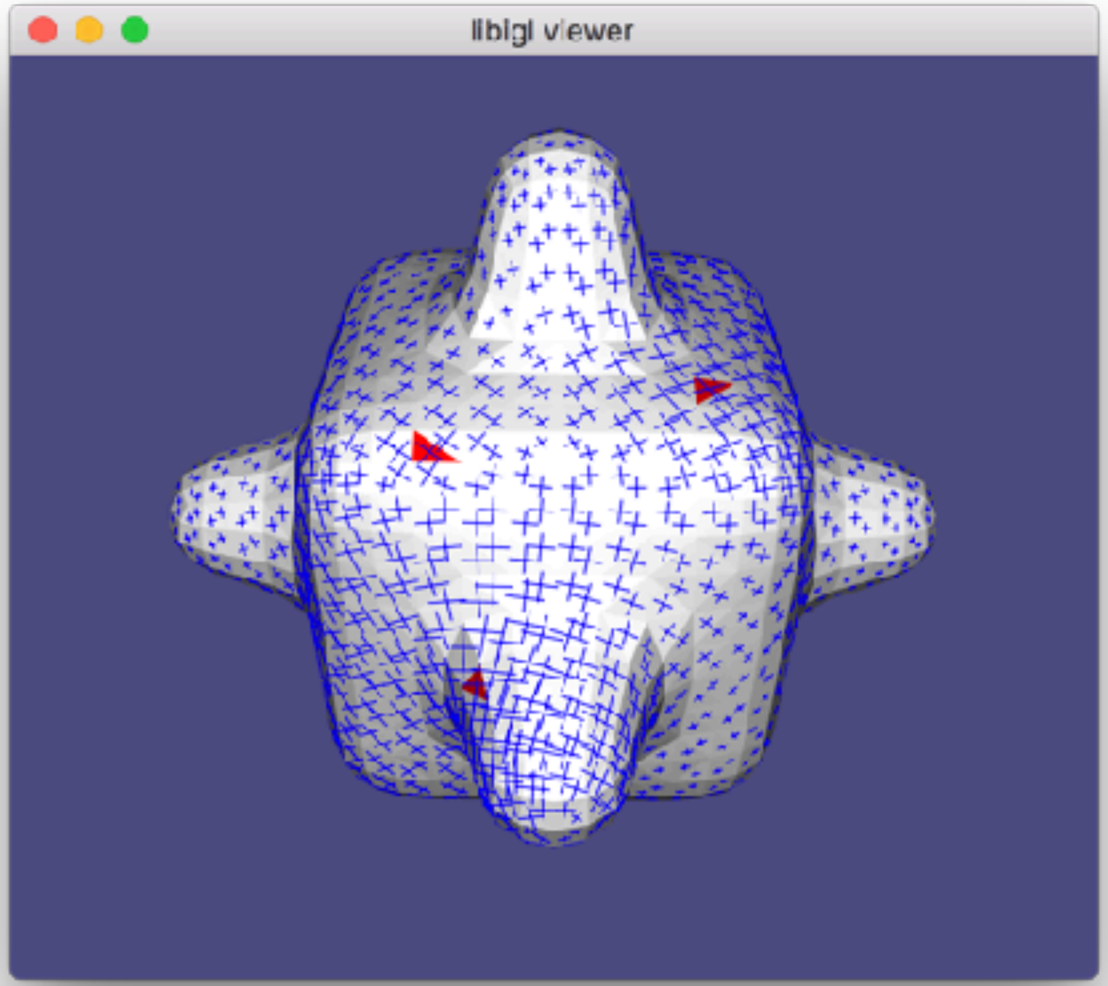
Daniele Panozzo
Courant Institute of Mathematical Sciences
New York University

N-ROSY FIELD DESIGN

- We will code a complete implementation of a simple algorithm to design **n-vector and n-directional fields**
- We will implement a method based on [Knöppel et al. 2013] and [Diamanti et. al. 2014]
  - The field will be discretized on **faces**
  - We will use the **Cartesian representation**
  - The topology will be **free**
  - The objective will be **smoothness**
  - We will support **soft alignment constraints**

FRAMEWORK

- You can download the code for this demo at: https://github.com/avaxman/DirectionalFieldSynthesis
- Self-contained, compiles on Windows/Mac/Linux
- It contains a simple graphical UI based on libigl, that allows us to interactively provide constraints and plots the result of our algorithm
- I encourage you to download the code now and experiment with it for the next 30 minutes



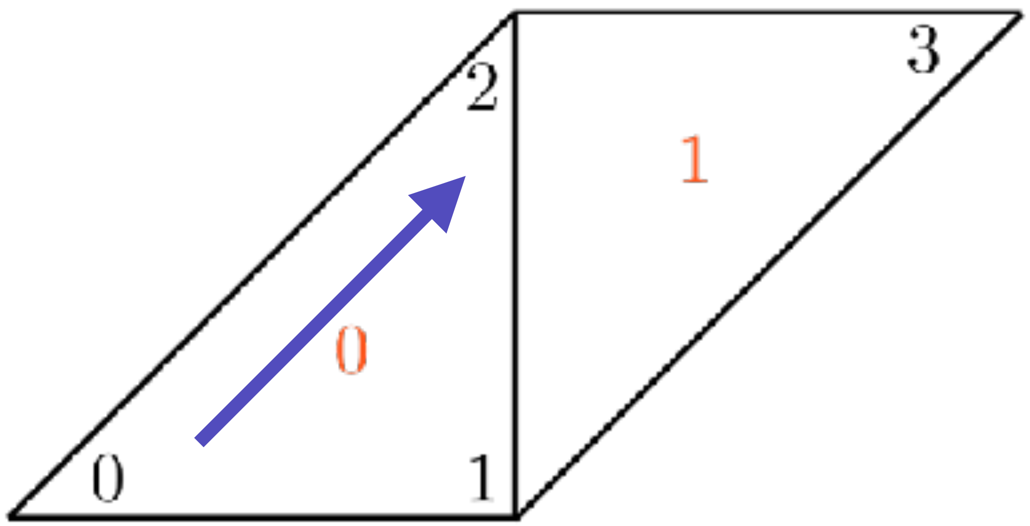
OUTLINE

- We will first implement an algorithm to interpolate 1-vector fields
    - This will require **discrete transport** between faces
- We will then extend it to n-vector fields and finally to n-directional fields
    - This will require to use the **Cartesian representation**
    - The symmetry of the field will be encoded in a complex polynomial (poly-vector)

INPUT/OUTPUT

- Mesh V, F

$$V = \begin{array}{c} \begin{array}{ccc} x & y & z \end{array} \\ \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 1 & 0 \end{pmatrix} \end{array} \qquad F = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 3 & 2 \end{pmatrix}$$



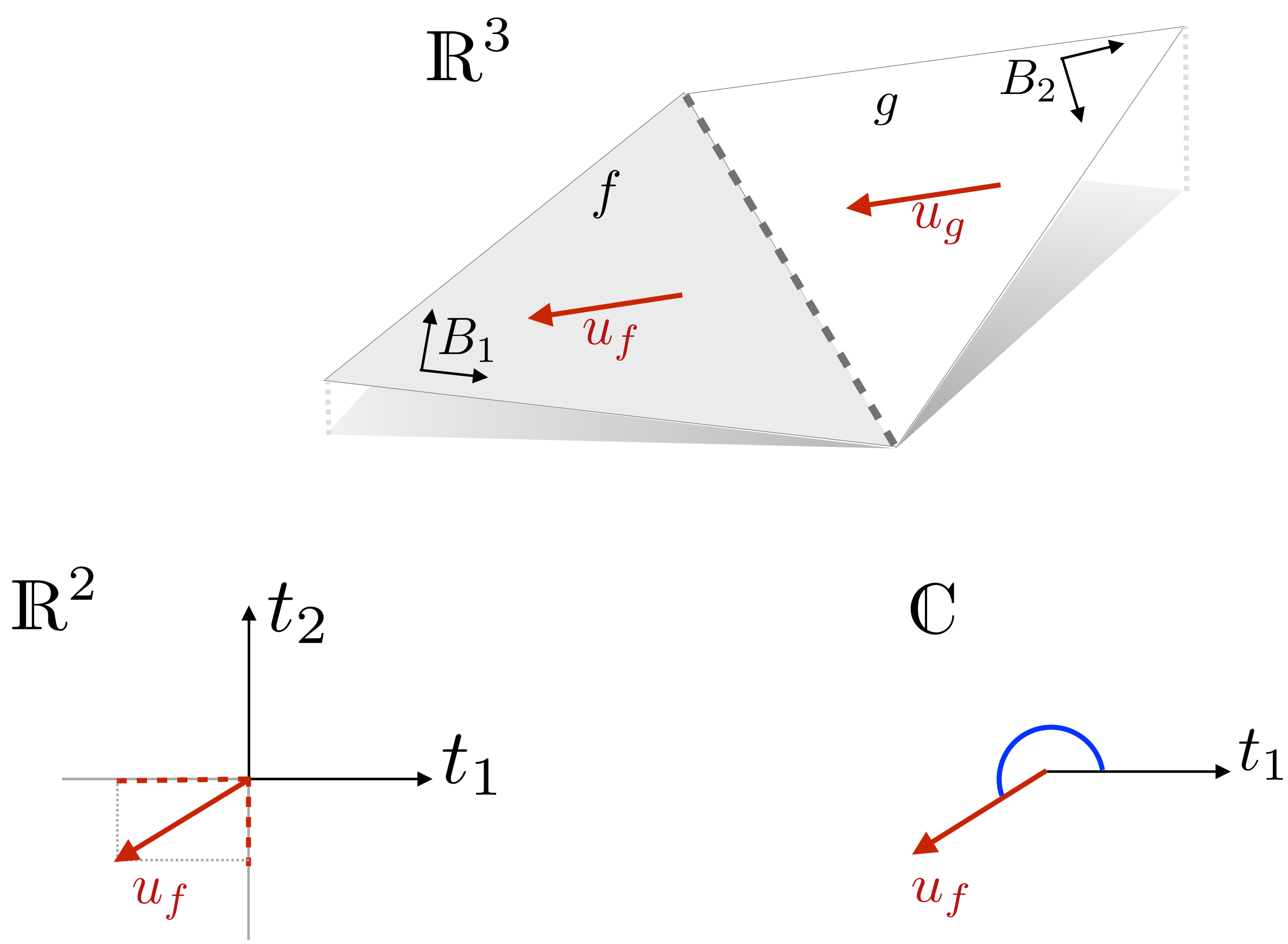
- Adjacency TT

$$TT = \begin{pmatrix} -1 & 1 & -1 \\ -1 & -1 & 0 \end{pmatrix}$$

- Constrained face ids *soft_id* and directions *soft_value*

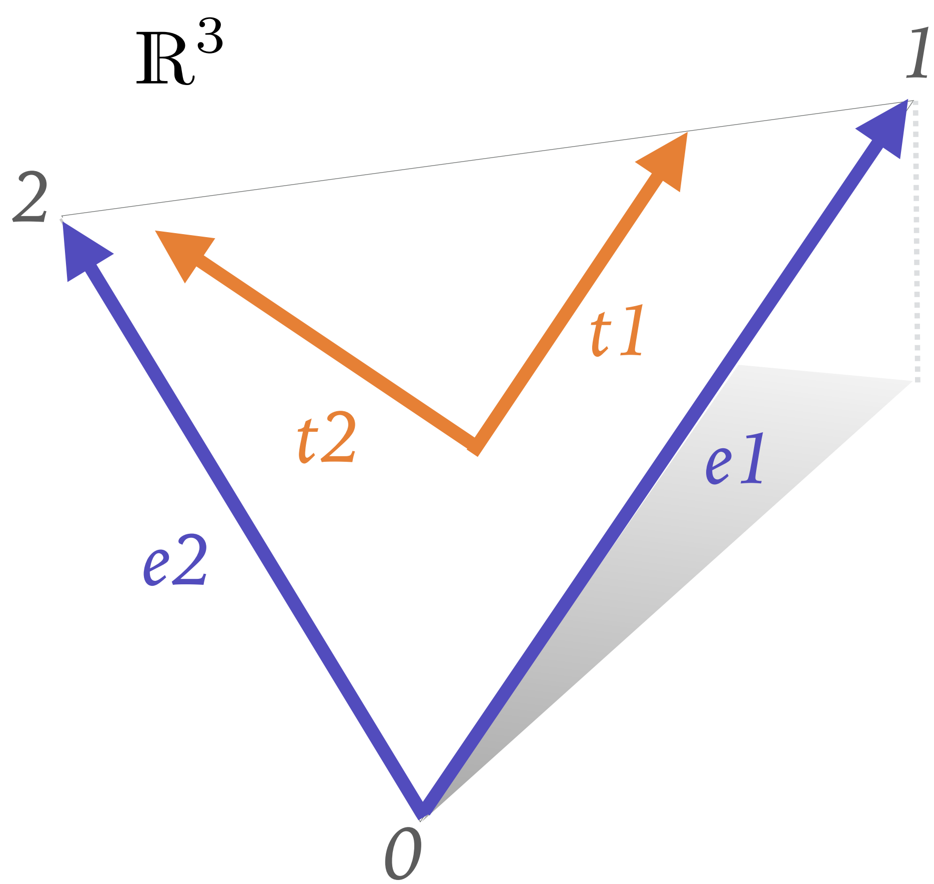
$$soft\_id = 0, soft\_value = (1\ 1\ 0)$$

DISCRETIZATION AND REPRESENTATION



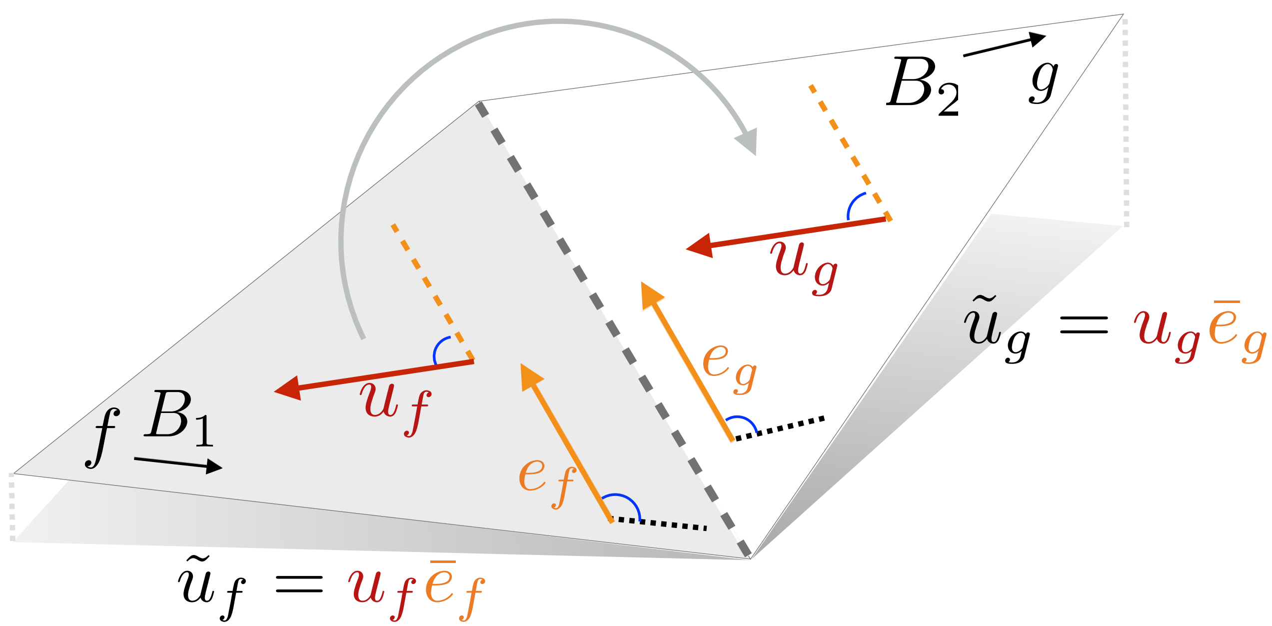
$$u_f = (u_f^x, u_f^y) \in \mathbb{R}^2 \qquad u_f = u_f^x + i u_f^y \in \mathbb{C}$$

CONSTRUCTING LOCAL BASES




```
for (unsigned i=0;i<F.rows();++i)
{
  Vector3d e1 =  V.row(F(i, 1)) - V.row(F(i, 0));
  Vector3d e2 =  V.row(F(i, 2)) - V.row(F(i, 0));
  T1.row(i) = e1.normalized();
  T2.row(i) = T1.row(i).cross(T1.row(i).cross(e2)).normalized();
}
```

DISCRETE TRANSPORT



Constant field (over an edge)

$$\boxed{u_f \bar{e}_f = u_g \bar{e}_g}$$

```
Vector3d e  = (V.row(F(f,(ei+1)%3)) - V.row(F(f,ei)));
Vector2d vef = Vector2d(e.dot(T1.row(f)),e.dot(T2.row(f))).normalized();
std::complex<double> ef(vef(0),vef(1));
Vector2d veg = Vector2d(e.dot(T1.row(g)),e.dot(T2.row(g))).normalized();
std::complex<double> eg(veg(0),veg(1));
```

ENERGY FORMULATION

- An ideally constant field satisfies (for each edge):

$$(u_f \overline{e}_f - u_g \overline{e}_g)$$

- We want to find the field that is as-constant-as-possible:

$$f(\mathbf{u}) = \underbrace{\sum_{f,g \in \mathcal{E}} \|u_f \overline{e}_f - u_g \overline{e}_g\|^2}_{\textit{As-constant-as-possible}} + \lambda \underbrace{\sum_{f \in \mathcal{C}} \|u_f - c_f\|2}_{\textit{Soft-constraints}}$$

- We can rewrite this expression in matrix form:

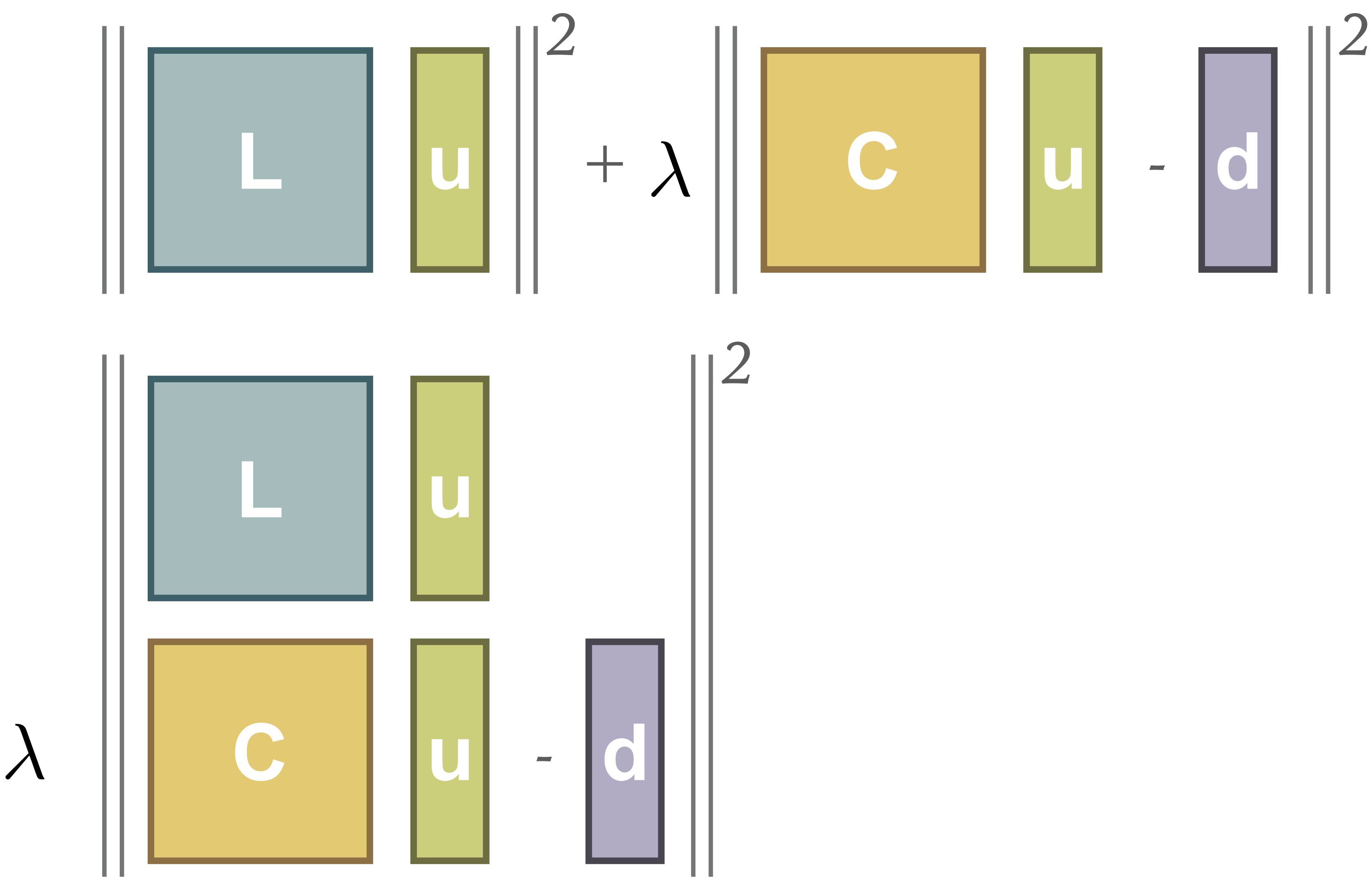
$$f(\mathbf{u}) = \underbrace{\|\mathbf{L}\mathbf{u}\|^2}_{\textit{As-constant-as-possible}} + \lambda \underbrace{\|\mathbf{C}\mathbf{u} - \mathbf{d}\|^2}_{\textit{Soft-constraints}}$$

ENERGY FORMULATION

$$f(\mathbf{u}) = \|\mathbf{L}\mathbf{u}\|^2 + \lambda\|\mathbf{C}\mathbf{u} - \mathbf{d}\|^2$$

ENERGY FORMULATION

$$f(\mathbf{u}) = \|\mathbf{L}\mathbf{u}\|^2 + \lambda\|\mathbf{C}\mathbf{u} - \mathbf{d}\|^2$$



ENERGY FORMULATION

$$f(\mathbf{u}) = \|\mathbf{L}\mathbf{u}\|^2 + \lambda\|\mathbf{C}\mathbf{u} - \mathbf{d}\|^2$$

- We can rearrange the expression in a single norm

$$f(\mathbf{u}) = \left\| \begin{matrix} \mathbf{L}\mathbf{u} \\ \sqrt{\lambda}\mathbf{C}\mathbf{u} - \sqrt{\lambda}\mathbf{d} \end{matrix} \right\|^2 = \|\mathbf{A}\mathbf{u} - \mathbf{b}\|^2 \qquad \mathbf{A} = \begin{pmatrix} \mathbf{L} \\ \sqrt{\lambda}\mathbf{C} \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} \mathbf{0} \\ \sqrt{\lambda}\mathbf{d} \end{pmatrix}$$

- which can be minimized solving a *complex* linear system

$$\nabla f(\mathbf{u}) = \mathbf{A}^*\mathbf{A}\underbrace{\mathbf{u}}_{\textit{Optimized Field}} - \mathbf{A}^*\mathbf{b} = 0$$

SYSTEM ASSEMBLY - SMOOTHNESS

- For each edge we want one row in $\mathbf{A}$ $\|u_f \overline{e}_f - u_g \overline{e}_g\|^2$ $\|\mathbf{Lu}\|^2$

```
unsigned count = 0;
for (unsigned f=0;f<F.rows();++f)
{
  for (unsigned ei=0;ei<F.cols();++ei)
  {
    // Look up the opposite face
    int g = TT(f,ei);
    // If it is a boundary edge, it does not contribute to the energy
    if (g == -1) continue;
    // Avoid to count every edge twice
    if (f > g) continue;
    // Compute the complex representation of the common edge
    Vector3d e  = (V.row(F(f,(ei+1)%3)) - V.row(F(f,ei)));
    Vector2d vef = Vector2d(e.dot(T1.row(f)),e.dot(T2.row(f))).normalized();
    std::complex<double> ef(vef(0),vef(1));
    Vector2d veg = Vector2d(e.dot(T1.row(g)),e.dot(T2.row(g))).normalized();
    std::complex<double> eg(veg(0),veg(1));
    // Add the term conj(f)^n*ui - conj(g)^n*uj to the energy matrix
    t.push_back(Triplet<std::complex<double> >(count,f,    std::conj(ef)));
    t.push_back(Triplet<std::complex<double> >(count,g,-1.*std::conj(eg)));
    ++count;
  }
}
```

SYSTEM ASSEMBLY - SOFT CONSTRAINTS

- Similarly, we add a row to **A** for each constrained face
- Note that the corresponding entry of **b** is not zero

$$\lambda \sum_{f \in \mathcal{C}} \|u_f - c_f\|^2 \qquad \|\sqrt{\lambda}\mathbf{C}\mathbf{u} - \sqrt{\lambda}\mathbf{d}\|^2$$

```
lambda = 10e6;
for (unsigned r=0; r<soft_id.size(); ++r)
{
  int f = soft_id(r);
  Vector3d v = soft_value.row(r);
  std::complex<double> c(v.dot(T1.row(f)),v.dot(T2.row(f)));
  t.push_back(Triplet<std::complex<double> >(count,f, sqrt(lambda)));
  tb.push_back(Triplet<std::complex<double> >(count,0, c * std::complex<double>(sqrt(lambda),0)));
  ++count;
}
```

SOLVING THE LINEAR SYSTEM

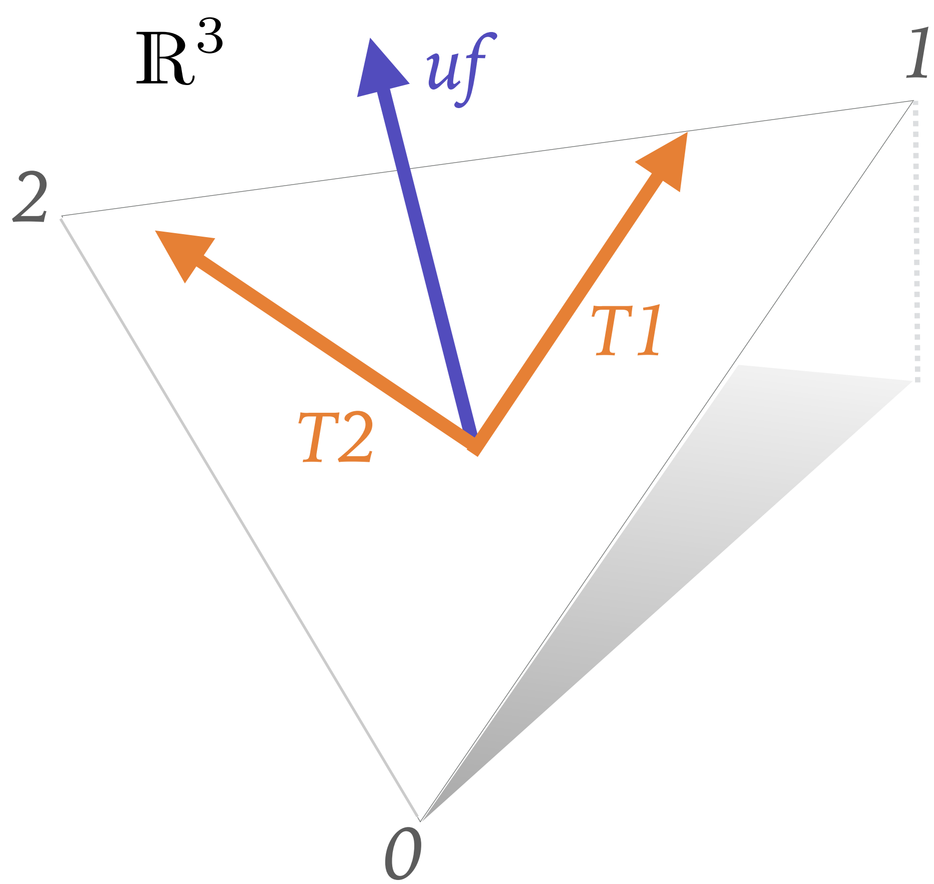
- The linear system is solved using a sparse direct solver.

$$\|\mathbf{A}\mathbf{u} - \mathbf{b}\|^2$$

$$\nabla f(\mathbf{u}) = \mathbf{A}^*\mathbf{A}\mathbf{u} - \mathbf{A}^*\mathbf{b} = 0$$

```
typedef SparseMatrix<std::complex<double>> SparseMatrixXcd;
SparseMatrixXcd A(count,F.rows());
A.setFromTriplets(t.begin(), t.end());
SparseMatrixXcd b(count,1);
b.setFromTriplets(tb.begin(), tb.end());
SimplicialLDLT< SparseMatrixXcd > solver;
solver.compute(A.adjoint()*A);
assert(solver.info()==Success);
MatrixXcd u = solver.solve(A.adjoint()*MatrixXcd(b));
assert(solver.info()==Success);
```

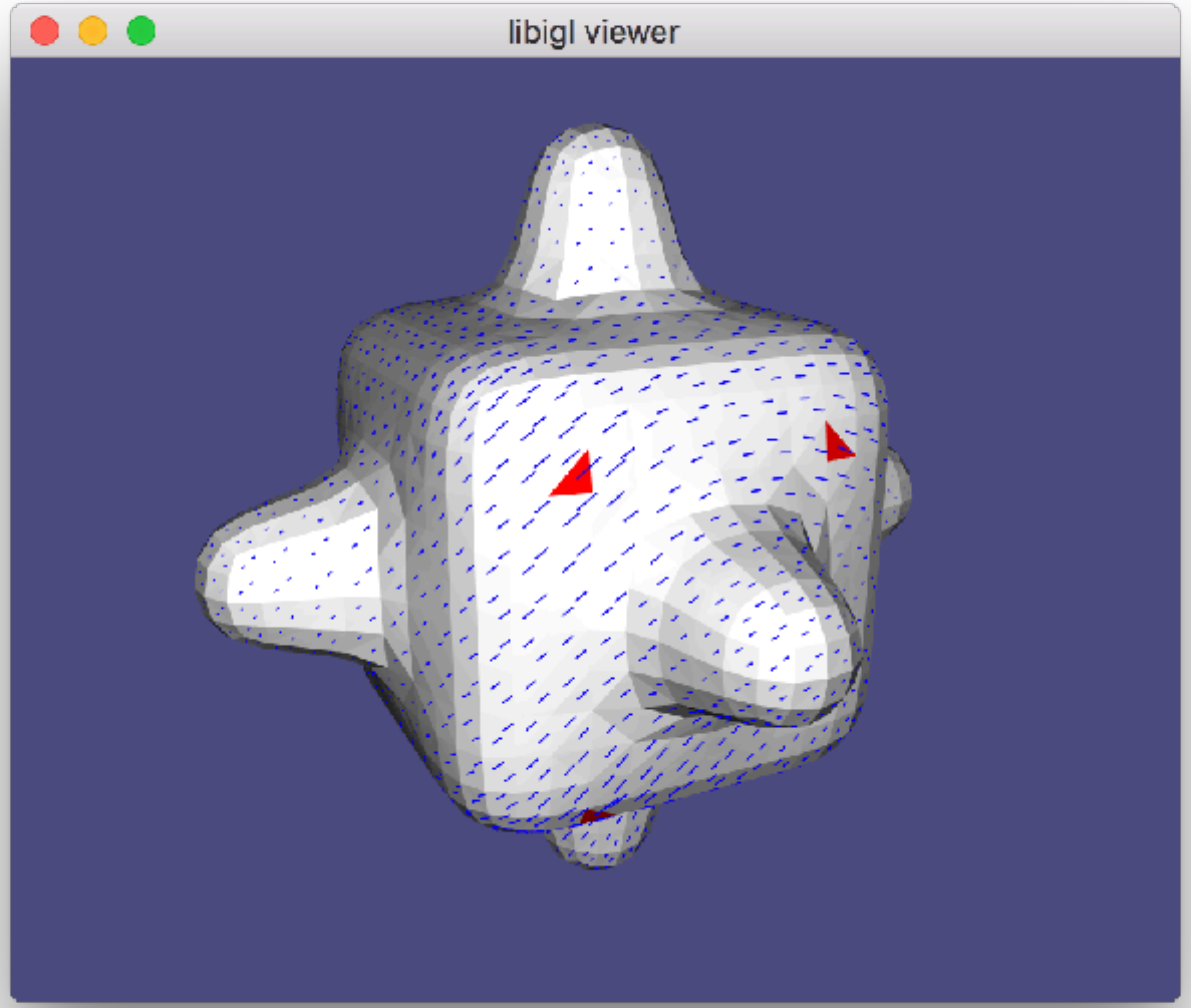
EXTRACTION OF THE INTERPOLATED FIELD




```
MatrixXd R(F.rows(),3);
for (int f=0; f<F.rows(); ++f)
  R.row(f) = T1.row(f) * u(f).real() + T2.row(f) * u(f).imag();

return R;
```

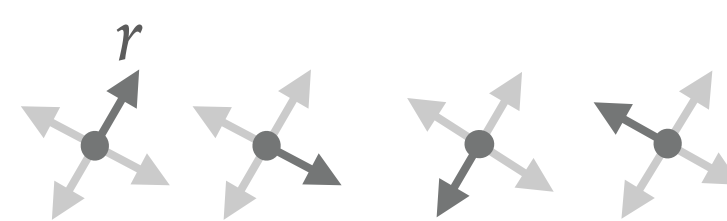
LET'S TAKE A LOOK AT THE RESULTING 1-VECTOR FIELD



EXTENSION TO N-VECTOR FIELDS

- Instead of representing the field with a 2D vector (complex number) we use a complex polynomial to represent a n-vector field.

$$p(z) = z^n - r^n$$



- Our new variable to interpolate is $u = r^n$
- The transport slightly changes in $(u_f(\overline{e}_f)^n - u_g(\overline{e}_g)^n)^2$
- The constraints need to be converted with $u = r^n$
- To extract the field, we need to find the roots of p

[Knöppel et al. 2013] [Diamanti et al. 2014]

ROOTS OF A COMPLEX POLYNOMIAL – COMPANION MATRIX

- The roots of a polynomial in the form

$$p(t) = c_0 + c_1 t + \cdots + c_{n-1} t^{n-1} + t^n$$

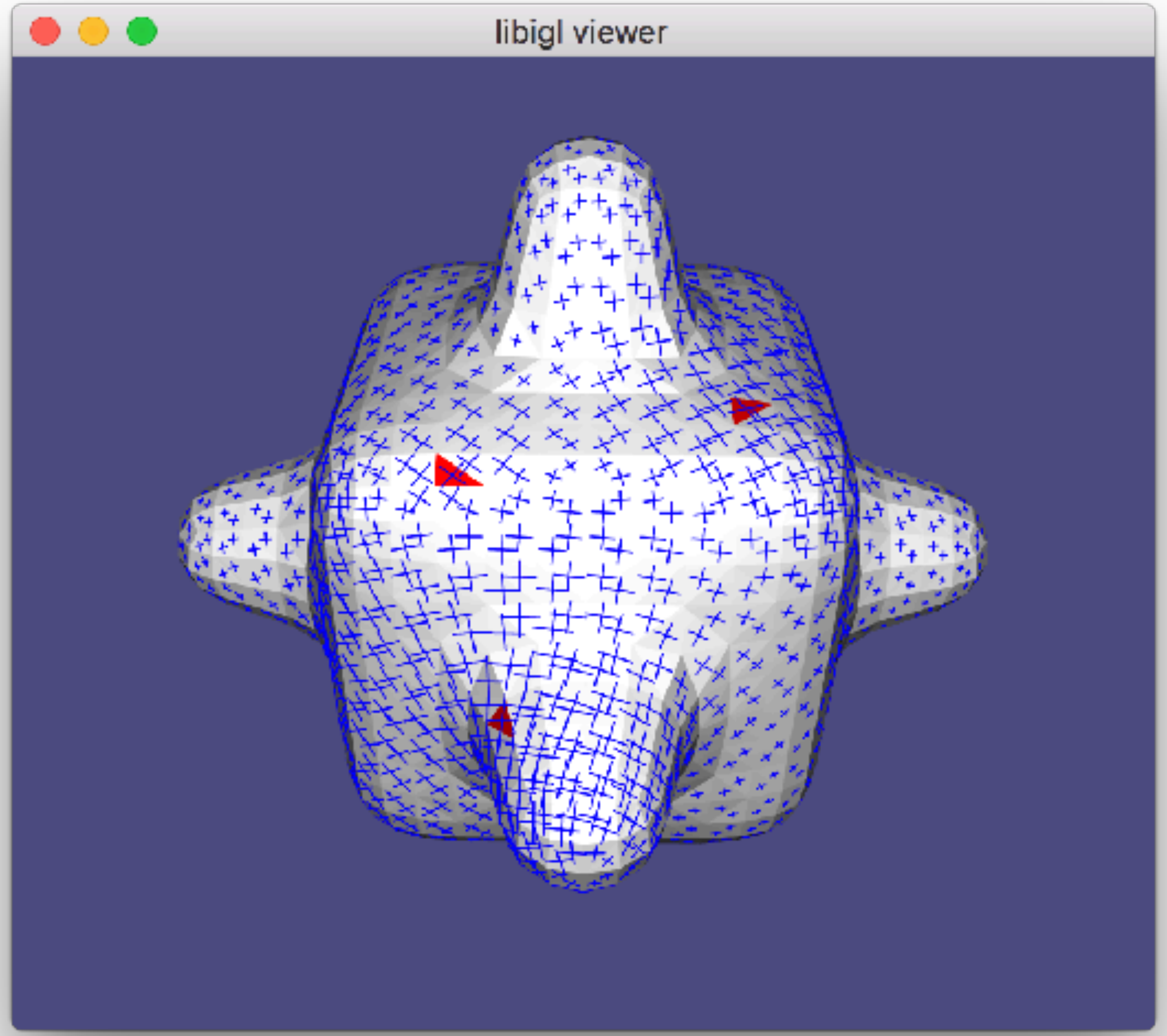
- are the eigenvalues of the corresponding companion matrix

$$C(p) = \begin{bmatrix} 0 & 0 & \dots & 0 & -c_0 \\ 1 & 0 & \dots & 0 & -c_1 \\ 0 & 1 & \dots & 0 & -c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & -c_{n-1} \end{bmatrix}.$$

```
std::complex<double> find_root(std::complex<double> c0, int n)
{
  // Find the roots of p(t) = (t - c0)^n using
  // https://en.wikipedia.org/wiki/Companion_matrix
  Eigen::MatrixXcd M = Eigen::MatrixXcd::Zero(n,n);
  for (int i=1;i<n;++i)
    M(i,i-1) = std::complex<double>(1,0);
  M(0,n-1) = c0;
  return M.eigenvalues()(0);
}
```

- For more information: https://en.wikipedia.org/wiki/Companion_matrix

CODING SESSION



EXTENSION FOR N-DIRECTIONAL FIELDS

- The algorithm we implemented is a special case of [Diamanti et al. 2014]
- If we normalize the extracted roots, we obtain an algorithm to compute N-directional fields that is very similar to [Knöppel et al. 2013] (same objective function, but different discretization)

REMARKS

- We implemented a simple but powerful algorithm to design n-vector and n-directional fields
- Most of the concepts covered in this course have been used in this coding session — we encourage you to experiment with it and to try to code it from scratch
- An extensive collection of public implementations of field design algorithms is included in the course notes.
- The source code for all the demos that we used in this course are available at: *https://github.com/avaxman/DirectionalFieldSynthesis*