Computing vector curvature and total curvature on a triangulated surface

a `deal.ii` implementation

Introduction

In this example we show how to compute the vector curvature (sometimes called *normal curvature*) $\boldsymbol{h}$ on a parametric surface S in $\mathbb{R}^3$. We include a function to extract the (scalar) total curvature h (sometimes called *mean curvature*), and finally we test our computations against the analytic expression for total curvature on ellipsoids.

We follow the notation of **step 38** and introduce the codimension 1 surface S which is known by a parameterization $\boldsymbol{x}_S : \hat{S} \to S$, where $\hat{S} \subset \mathbb{R}^2$, and $\boldsymbol{x}_S$ takes points $\hat{\boldsymbol{x}} = (\hat{x}_1, \hat{x}_2) \in \hat{S}$ to S. In what follows we will continue to use the cumbersome notation $\boldsymbol{x}_S$ to denote points in $\mathbb{R}^3$ that lie on S, and $\boldsymbol{x}$ to denote arbitrary points in $\mathbb{R}^3$. The purpose of this is to emphasize ...

The total curvature h on S is defined at every point $\boldsymbol{x}_S \in S$ and can be described as the sum of the principle curvatures $\kappa_1(\boldsymbol{x}_S) + \kappa_2(\boldsymbol{x}_S)$ of S, hence

$$h(\boldsymbol{x}_S) = \kappa_1(\boldsymbol{x}_S) + \kappa_2(\boldsymbol{x}_S).$$

Another way to think about the total curvature in a geometric sense is as the tangential divergence of the normal vector, that is $h = \mathrm{div}_S \boldsymbol{n} \overset{\mathrm{id}}{=} \nabla_S \cdot \boldsymbol{n}$ (this will appear again below).

The vector curvature $\boldsymbol{h}$ on S is simply defined as the (scalar) total curvature times the normal vector, hence

$$\boldsymbol{h}(\boldsymbol{x}_S) = h(\boldsymbol{x}_S)\boldsymbol{n}(\boldsymbol{x}_S).$$

It is the vector curvature that we will be computing first because, quite unlike the (scalar) total curvature, $\boldsymbol{h}$ has an emminently useful formulation in terms of the identity $\mathbf{id}_S : \boldsymbol{x}_S \mapsto \boldsymbol{x}_S$, and the Laplace Beltrami operator Δ_S on S. In fact,

$$-\Delta_S \, \mathbf{id}_S = \boldsymbol{h}.$$

There is more wrapped up in the notation above than first meets the eye. First, $\mathbf{id}_S$ is vector-valued, so the Laplace-Beltrami operator here is meant to be taken componentwise (and will be made explicit below). Second, recall that we have defined S through the parameterization $\boldsymbol{x}_S : (\hat{x}_1, \hat{x}_2) \to S \subset \mathbb{R}^3$. This makes $\mathbf{id}_S$ appear as the funtion $\mathbf{id}_S := (\mathbf{id} \circ \boldsymbol{x}_S)(\hat{x}_1, \hat{x}_2) = \boldsymbol{x}_S(\hat{x}_1, \hat{x}_2)$ for

$(\hat{x}_1, \hat{x}_2) \in \hat{S} \subset \mathbb{R}^2$. In this setting, the derivatives involved in Δ_S must be taken back in $\hat{S}$, which is cumbersome and best left for `deal.ii` to do for us (cf **step 38**), yet we want to unpack at least some of what has been written above before we proceed with the program. To this end, we hypothesize an extension of $\mathbf{id}_S$ to all of $\mathbb{R}^3$ (which is absolutely trivial in this case!), and take our derivatives in $\mathbb{R}^3$.

We now quickly show that it is reasonable that we may get the vector curvature from the surface Laplacian of the identity: Let $\mathbf{id} : \mathbb{R}^3 \to \mathbb{R}^3$ extend the identity $\mathbf{id}_S$ to at least a neighborhood of S in $\mathbb{R}^3$, and write $\mathbf{id}(\boldsymbol{x}) = (\mathrm{id}_1(\boldsymbol{x}), \mathrm{id}_2(\boldsymbol{x}), \mathrm{id}_3(\boldsymbol{x})) = (x_1, x_2, x_3)$. Then generalizing the expression for the Laplace Beltrami operator in **step 38** to vector-valued functions, we write

$$\Delta_S \, \mathbf{id}_S := \begin{pmatrix} \Delta \mathrm{id}_1 - \boldsymbol{n}^T D^2 \mathrm{id}_1 \, \boldsymbol{n} - (\boldsymbol{n} \cdot \nabla \mathrm{id}_1) \left(\nabla \cdot \boldsymbol{n} - \boldsymbol{n}^T D \boldsymbol{n} \, \boldsymbol{n} \right) \\ \Delta \mathrm{id}_2 - \boldsymbol{n}^T D^2 \mathrm{id}_2 \, \boldsymbol{n} - (\boldsymbol{n} \cdot \nabla \mathrm{id}_2) \left(\nabla \cdot \boldsymbol{n} - \boldsymbol{n}^T D \boldsymbol{n} \, \boldsymbol{n} \right) \\ \Delta \mathrm{id}_3 - \boldsymbol{n}^T D^2 \mathrm{id}_3 \, \boldsymbol{n} - (\boldsymbol{n} \cdot \nabla \mathrm{id}_3) \left(\nabla \cdot \boldsymbol{n} - \boldsymbol{n}^T D \boldsymbol{n} \, \boldsymbol{n} \right) \end{pmatrix}.$$

First notice that $\left(\nabla \cdot \boldsymbol{n} - \boldsymbol{n}^T D \boldsymbol{n} \, \boldsymbol{n} \right) = \mathrm{div}_S \boldsymbol{n}$, which we claimed to be $h(\boldsymbol{x}_S)$ above (see **step 38** and [] for more about tangential derivatives). Since we're dealing with identity functions all around, we have that $\nabla \mathrm{id}_i = \boldsymbol{e}_i$, $\Delta \mathrm{id}_i = 0$, and $D^2 \mathrm{id}_i$ is the zero matrix for $i = 1, 2, 3$, leaving us with

$$\Delta_S \, \mathbf{id}_S = \begin{pmatrix} -(\boldsymbol{n} \cdot (1,0,0)) h(\boldsymbol{x}_S) \\ -(\boldsymbol{n} \cdot (0,1,0)) h(\boldsymbol{x}_S) \\ -(\boldsymbol{n} \cdot (0,0,1)) h(\boldsymbol{x}_S) \end{pmatrix} = \begin{pmatrix} -h(\boldsymbol{x}_S) n_1 \\ -h(\boldsymbol{x}_S) n_2 \\ -h(\boldsymbol{x}_S) n_3 \end{pmatrix} = -\boldsymbol{h}(\boldsymbol{x}_S).$$

Strategy For Computations

Presumably the purpose of computing curvature on a surface is to then use it in further computations. This means that we should make sure the data structure that holds curvature information is be compatible with further finite element comptuations in `deal.ii`. All this is to say that we will use `deali.ii` to 'solve' $-\Delta_S \, \mathbf{id}_S = \boldsymbol{h}$ for $\boldsymbol{h}$ (more precisely, we will weakly enforce the identity $\nabla_S \boldsymbol{\varphi} \nabla_S \mathbf{id} \overset{\mathrm{id}}{=} \boldsymbol{\varphi} \boldsymbol{h}$). For this all we really need is a decent mesh of points lying on S so that we have a discrete representation of $\mathbf{id}_S$. Instead of only specifying the surface S by vertices of a mesh, we will instead specify a `MappingQEulerian<dim,spacedim> mapping(...)` object initialized with a callable `Function<spacedim> map_to_S(spacedim)` which parameterizes S. This allows us to use higher-order finite elements because we can specify S at **support points** as well as at the vertices, enhancing our discrete representation of $\mathbf{id}_S$ and thereby enhancing the accuracy of our approximation of $\boldsymbol{h}$.

As noted in **step-38**, the user will not be expressing derivatives back in $\hat{S}$. Rather, the user will specify a `Mapping` on a `Triangulation` (or specifying just a `Triangulation`, and letting `deal.ii` associate a default piecewise-linear `Mapping` on that `Triangulation`)

Test Case

Ellipsoid parameterized, $\hat{S} = [0, 2\pi] \times [0, \pi]$ by

$$\boldsymbol{x}(\hat{x}_1, \hat{x}_2) = \begin{pmatrix} a \sin \hat{x}_1 \cos \hat{x}_2 \\ b \sin \hat{x}_1 \sin \hat{x}_2 \\ c \cos \hat{x}_1 \end{pmatrix}$$

has total (scalar) curvature

$$h(\hat{x}_1, \hat{x}_2) = \frac{2abc\left[3(a^2+b^2) + 2c^2 + (a^2+b^2-2c^2)\cos(2\hat{x}_1) - 2(a^2-b^2)\cos(2\hat{x}_2)\sin^2\hat{x}_1\right]}{8\left[a^2b^2\cos^2\hat{x}_1 + c^2(b^2\cos^2\hat{x}_2 + a^2\sin^2\hat{x}_2)\sin^2\hat{x}_1\right]^{3/2}}$$

```
template<int spacedim>
double ExactScalarMeanCurvatureOnEllipsoid<spacedim>::value(const Point<spacedim> &p, const
{
  Point<spacedim> unmapped_p(p(0)/a, p(1)/b,  p(2)/c);

  Point<spacedim> chart_point = spherical_manifold.pull_back(unmapped_p);
  // double radius = chart_point(0);
  double x1_hat = chart_point(1);
  double x2_hat = chart_point(2);
  ...
}
```

We don't reimplement the parameterization from $\hat{S}$ to $\mathbb{R}^3$ here – instead we use the built-in `SphericalManfold<dim,spacedim> sphere` and rescale `sphere` in $\mathbb{R}^3$. The `pull_back` and `push_forward` functions allow us to write our functions in terms of $\boldsymbol{x} \in \mathbb{R}^3$ rather than $\hat{\boldsymbol{x}} \in \hat{S} \subset \mathbb{R}^2$.

```
template<int spacedim>
Tensor<1,spacedim> ExactVectorMeanCurvatureOnEllipsoid<spacedim>::value(const Point<spacedim
{
  Tensor<1,spacedim> normal,vector_H;
  normal[0] = p(0)/a;
  normal[1] = p(1)/b;
  normal[2] = p(2)/c;
  normal /= normal.norm();

  vector_H  = normal;
  vector_H *= exact_scalar_H.value(p);

  return vector_H;
}
```

Implementation