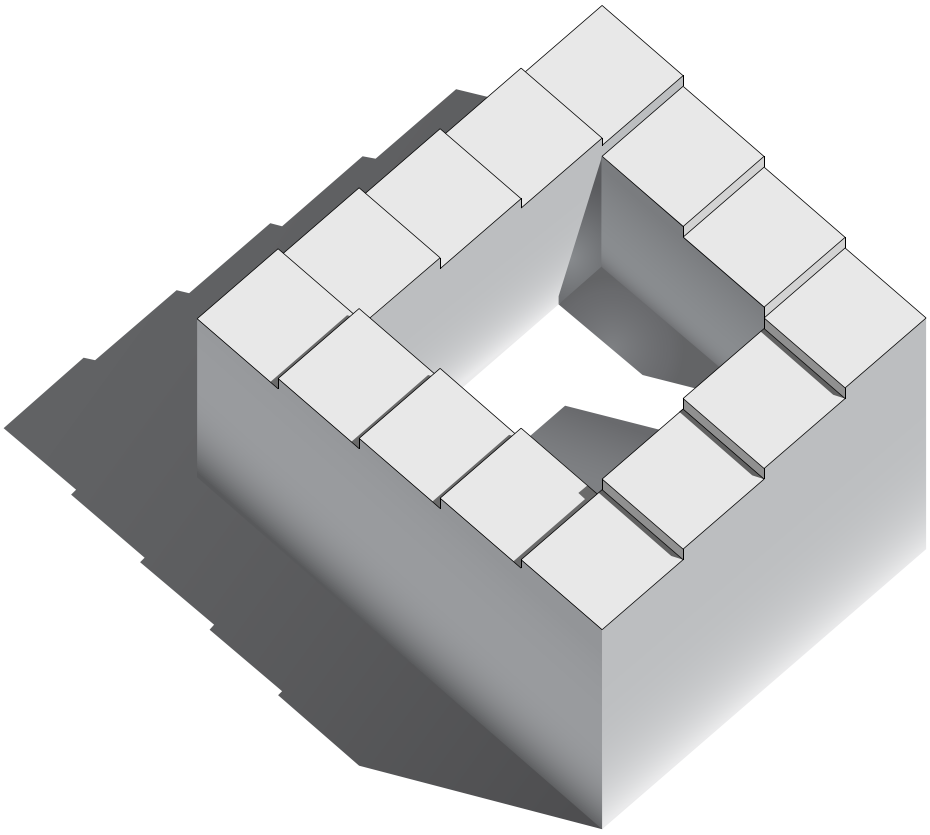
Discrete Differential Geometry: An Applied Introduction



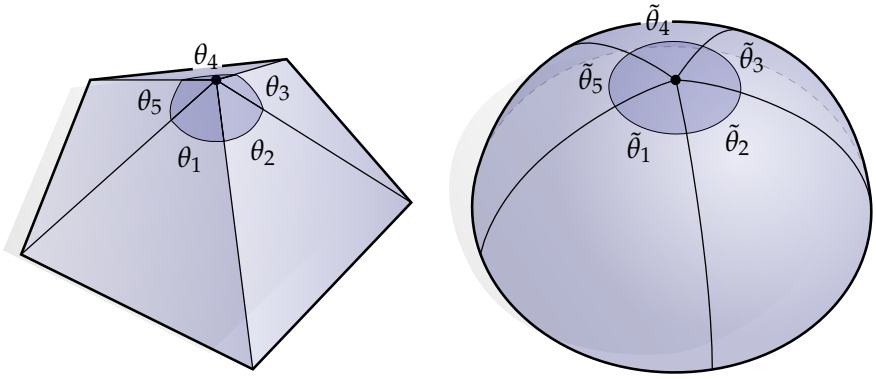
Keenan Crane

Last updated: May 2, 2022

Contents

CHAPTER 1

Introduction



These notes focus on three-dimensional geometry processing, while simultaneously providing a first course in traditional differential geometry. Our main goal is to show how fundamental geometric concepts (like curvature) can be understood from complementary computational and mathematical points of view. This dual perspective enriches understanding on both sides, and leads to the development of practical algorithms for working with real-world geometric data. Along the way we will revisit important ideas from calculus and linear algebra, putting a strong emphasis on *intuitive, visual* understanding that complements the more traditional formal, algebraic treatment. The course provides essential mathematical background as well as a large array of real-world examples and applications. It also provides a short survey of recent developments in digital geometry processing and discrete differential geometry. **Topics include:** curves and surfaces, curvature, connections and parallel transport, exterior algebra, exterior calculus, Stokes' theorem, simplicial homology, de Rham cohomology, Helmholtz-Hodge decomposition, conformal mapping, finite element methods, and numerical linear algebra. **Applications include:** approximation of curvature, curve and surface smoothing, surface parameterization, vector field design, and computation of geodesic distance.

One goal of these notes is to provide an introduction to working with real-world geometric data, expressed in the language of *discrete exterior calculus (DEC).* DEC is a simple, flexible, and efficient framework which provides a unified platform for geometry processing. The notes provide essential mathematical background as well as a large array of real-world examples, with an emphasis on applications and implementation. The material should be accessible to anyone with some exposure to basic linear algebra and vector calculus, though most of the key concepts are reviewed as needed. Coding exercises depend on a basic knowledge of either Javascript or C++, though knowledge of *any* programming language is likely sufficient: we do not make heavy use of paradigms like inheritance, templates, *etc.* The notes also provide guided written exercises that can be used to deepen understanding of the material.

Why use exterior calculus? There are, after all, many other ways to describe algorithms for mesh processing. One reason has to do with *language*: the exterior calculus of differential forms is, to a large degree, the modern language of differential geometry and mathematical physics. By learning to speak this language we can draw on a wealth of existing knowledge to develop new algorithms, and better understand current algorithms in terms of a well-developed theory. It also allows us to easily write down—and implement—many seemingly disparate algorithms in a single, unified framework. In these notes, for instance, we'll see how a large number of basic geometry processing tasks (smoothing, parameterization, vector field design, *etc.*) can be expressed in only a few lines of code, typically by solving a simple *Poisson equation*.

There is another good reason for taking this approach, beyond simply "saying the same thing in a different way." By first formulating algorithms in the smooth geometric setting, we can ensure that essential structures are subsequently preserved at the discrete level. As one elementary example, consider the vertex depicted above. If we take the sum of the tip angles θ_i, we get a number that is (in general) different from 2π. On any smooth surface, however, we expect this number to be exactly 2π—said in a differential-geometric way: the *tangent space* at any point should consist of a "whole circle" of directions. Of course, if we consider finer and finer approximations of a smooth surface by a triangle mesh, the vertex will eventually flatten out and our angle sum will indeed approach 2π as expected. But there is an attractive alternative even at the coarse level: we can redefine the meaning of "angle" so that it always yields the expected result. In particular, let

$$s := \frac{2\pi}{\sum_i \theta_i}$$

be the ratio between the angle sum 2π that we anticipate in the smooth setting, and the Euclidean angle sum $\sum_i \theta_i$ exhibited by our finite mesh, and consider the augmented angles

$$\tilde{\theta}_i := s\theta_i.$$

In other words, we simply normalize the usual Euclidean angles such that they sum to *exactly* 2π, no matter how coarse our mesh is:

$$\sum_i \tilde{\theta}_i = s \sum_i \theta_i = 2\pi.$$

From here we can carry out all the rest of our calculations as usual, using the augmented or "discrete" angles $\tilde{\theta}_i$ rather than the usual Euclidean angles θ_i. Conceptually, we can imagine that each vertex has been smoothed out slightly, effectively pushing the curvature of our surface into otherwise flat triangles. This particular convention may not always (or even often) be useful, but in problems where the tangent space structure of a surface is critical it leads to highly effective algorithms for mesh processing (see for example [**KCPS13, SC18**]).

This message is one theme we'll encounter frequently in these notes: there is no one "right" way to discretize a given geometric quantity, but rather *many different ways*, each suited to a particular purpose. The hope, then, is that one can discretize a whole *theory* such that all the pieces fit together nicely. DEC is one such theory, which has proven to be highly successful at preserving the *homological* structure of a surface, as we'll discuss in Chapter 8.

The remainder of these notes proceeds as follows. We first give an overview of the differential geometry of surfaces (Chapter 3), using a description that leads naturally into a discussion of smooth exterior calculus (Chapter 4) and its discretization via DEC. We then study some basic properties of discrete surfaces (Chapter 2) and their normals (Chapter 5), leading up to an equation that is central to our applications: the discrete Poisson equation (Chapter 6). The remaining chapters investigate various geometry processing applications, introducing essential geometric concepts

along the way (conformal structure, homology, parallel transport, *etc.*). Coding exercises refer to a supplementary C++ framework, available from

`https://github.com/dgpdec/course`

which includes basic mesh data structures, linear algebra libraries, and visualization tools—any similar framework or library would be suitable for completing these exercises. Solutions to written exercises are available upon request.

Our goal throughout these notes was to describe every concept in terms of a concrete geometric picture—we have tried as much as possible to avoid abstract algebraic arguments. Likewise, to get the most out of the written exercises one should try to make an intuitive geometric argument *first*, and only later fill in the formal details.

1.1. Disclaimer

These notes are very much a work in progress and ***there will be errors***. As always, your brain is the best tool for determining whether a statement is actually true! If you encounter errors please don't hesitate to contact the author, noting the page number and the version of the notes.

1.2. Copyright

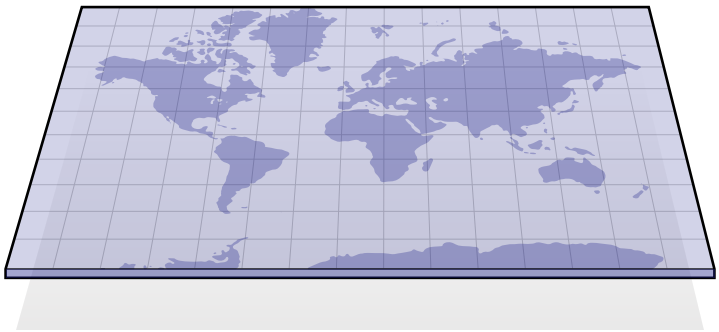
1.3. Acknowledgements

These notes grew out of a course on discrete differential geometry (DDG) taught annually starting in 2011, first at Caltech and now at CMU. Peter Schröder, Max Wardetzky, and Clarisse Weischedel provided invaluable feedback for the first draft of many of these notes; Mathieu Desbrun, Fernando de Goes, Peter Schröder, and Corentin Wallez provided extensive feedback on the SIGGRAPH 2013 revision. Joshua Brakensiek and Mark Gillespie made some nice contributions to written exercises; Nicholas Sharp and Rohan Sawhney helped revolutionize the associated codebase. Thanks to Mark Pauly's group at EPFL for suffering through (very) early versions of these lectures, to Eitan Grinspun for detailed feedback and for helping develop exercises about convergence, and to David Bommes for test-driving the whole thing at Aachen. David Bachman and Katherine Breeden provided a bunch of useful feedback in Spring 2020, during their run of the course at Harvey Mudd College and Pitzer College. Thanks also to those who have pointed out errors over the years: Mirela Ben-Chen, Nina Amenta, Chris Wojtan, Yuliy Schwarzburg, Robert Luo, Andrew Butts, Scott Livingston, Christopher Batty, Howard Cheng, Gilles-Philippe Paillé, Jean-François Gagnon, Nicolas Gallego-Ortiz, Henrique Teles Maia, Joaquín Ruales, Papoj Thamjaroenporn, Niklas Rieken, Yuxuan Mei, John C. Bowers, and all the students in 15-458/858 at CMU and CS177 at Caltech, as well as others who I am forgetting!

Most of the algorithms described in these notes appear in previous literature. The method for mean curvature flow appears in [**DMSB99**]. The conformal parameterization scheme described in Chapter 7 is based on [**MTAD08**]. The approach to discrete Helmholtz-Hodge decomposition described in Chapter 8 is based on the scheme described in [**DKT08**]. The method for computing smooth vector fields with prescribed singularities is based on [**CDS10**]; the improvement using Helmholtz-Hodge decomposition (Section 8.4.1) is previously unpublished and due to Fernando de Goes [**dGC10**]. More material on DEC itself can be found in a variety of sources [**Hir03, DHLM05, DKT08**]. Finally, the cotan-Laplace operator central to many of these algorithms has a long history, dating back at least as far as [**Mac49**].

CHAPTER 2

Combinatorial Surfaces



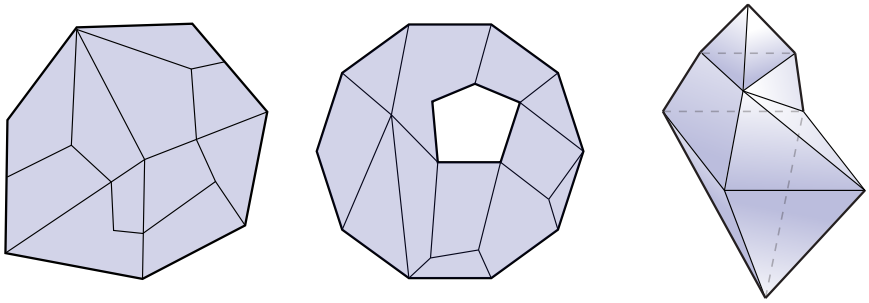
"Everything should be made as simple as possible, but no simpler."
—Al

A surface is, roughly speaking, the "outer shell" of a shape—for instance, you can think of a whole orange as a solid ball; its peel describes a spherical *surface* (especially if we consider an idealized peel with zero thickness). Different objects we encounter in our daily lives have boundaries described by different surfaces. For instance, the glaze covering a donut makes a *torus* rather than a sphere. (Hopefully all this talk of oranges and donuts is making you hungry for some geometry…) As a prelude to really getting into the differential geometry of surfaces, we're going to start by looking at objects that are easy to understand from a purely discrete point of view, namely, *combinatorial surfaces*, or descriptions of shapes that only tell you how surfaces are *connected up* and not *where they are in space*. In discrete differential geometry, combinatorial surfaces effectively play the same role that *topological surfaces* do in the smooth setting. We won't get deep into topology in these notes, but working with discrete surfaces "sans geometry" should give you a pretty good feel for what topology is all about. In particular, we'll talk about several different ways to encode the connectivity of combinatorial surfaces: using an *abstract simplicial complex*, *adjacency matrices*, and a *halfedge mesh*, all of which tie in to the richer geometric objects and algorithms we want to work with later on. (Those craving a more technical treatment may want to check out Hatcher's book on *algebraic topology* [**Hat02**].)

Taking a cue from "Al",[1] we're going to make some simplifying assumptions about what shapes look like, while still retaining enough flexibility to describe the kinds of objects found in the natural world. These simplifications will both make it easier to establish clean descriptions and definitions of geometric phenomena (such as curvature), and will ultimately help us build lean, clean algorithms that don't need to consider lots of special situations and corner cases. The basic simplifying assumption of differential geometry is that the shapes we want to study are *manifold*.

[1] The quote above paraphrases Albert Einstein, who actually said, *"It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience."*

Loosely speaking this means that, at least under a microscope, they look the same as ordinary Euclidean space. For instance, standing on the surface of the (spherical) Earth, it's pretty hard to tell that you're not standing on a flat plane. The manifold assumption is powerful because it lets us translate many of the things we know how to do in flat Euclidean spaces (*e.g.*, work with vectors, differentiate, integrate, *etc.*) to more interesting curved spaces. There are in fact many distinct ways in which a shape can "look like Euclidean space," leading to many distinct sub-areas of differential geometry (differential topology, conformal geometry, Riemannian geometry, ...). For now, we want to focus on an utterly basic property of surfaces, namely that around any point you can find a small neighborhood that is a *topological disk*.



polygonal disk *(neither)* *polyhedron*

A *topological disk* is, roughly speaking, any shape you can get by deforming the unit disk in the plane without tearing it, puncturing it, or gluing its edges together. Some examples of shapes that are disks include a flag, a leaf, and a glove. Some examples of shapes that are *not* disks include a circle (*i.e.*, a disk *without* its interior), a solid ball, a hollow sphere, a donut, a fidget spinner, and a teapot. Pictured above, for instance, we have a topological disk (left) a topological annulus (center) and a topological (sphere) made by gluing together a finite number of polygons along their edges.

In this chapter we'll start out by defining an *abstract simplicial complex*, which breaks a shape up into simple pieces like edges, triangles, and tetrahedra. Any abstract simplicial complex can be encoded by *incidence matrices*, which are basically just big tables recording which elements are next to which other elements. Although this description can capture some pretty complicated shapes, it's often *more general* than what we really need for discrete differential geometry. We therefore take a look at the *halfedge mesh*, which is specifically tailored to two-dimensional *surfaces*, and can easily describe surfaces with general polygonal faces (rather than just triangles). The halfedge mesh will serve as our basic data structure for most of the algorithms we consider in these notes. At the end of the chapter, we'll do some exercises that reveal some fun, interesting, and useful properties of combinatorial surfaces, and will get some hands-on intuition for how all these representations fit together by writing some code that lets us interactively navigate a combinatorial surface.

2.1. Abstract Simplicial Complex

How can we encode surfaces by a finite amount of information that makes it possible to distinguish a sphere from a torus? For now, let's forget about shape or *geometry* (how big, small, thick, thin, *etc.*, the shape is) and focus purely on *connectivity*: which pieces of the surface are connected to each other, and how?

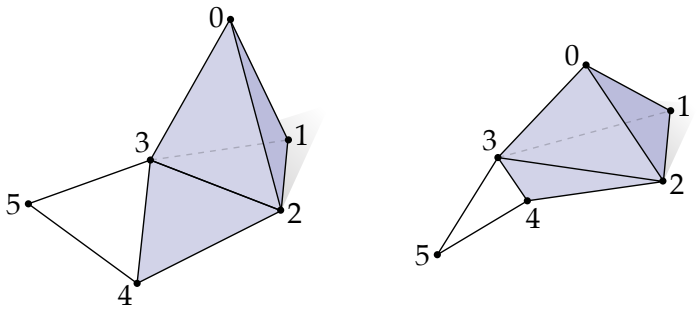


FIGURE 1. An abstract simplicial complex specifies how vertices are connected, but not where they are in space. For instance, both of the figures above represent the same simplicial complex, comprised of six vertices, ten edges, five triangles, and one tetrahedron.

There are many different ways to describe the connectivity of a discrete surface; one way is to use a *simplicial complex*—which in in fact can encode much more complicated objects than just surfaces. The basic idea is to start out with a set V of *vertices*, which we can identify with a collection of integers:

$$V = \{0, 1, 2, \ldots, n\}$$

We also need some information about how these vertices are connected. The idea of a simplicial complex is to specify subsets of these vertices that are "right next to each-other," called *k-simplices*. The number $k \in \mathbb{Z}_{>=0}$ is a nonnegative integer telling us how many elements are in this set: an abstract k-simplex is a set of $(k+1)$ distinct vertices, and we call k the *degree* of the simplex. For instance, here's a triangle or 2-simplex:

$$\{3, 4, 2\}$$

and here's a 1-simplex:

$$\{3, 5\}$$

Geometrically, we can think of a 2-simplex as specifying a triangle, and a 1-simplex as specifying an edge, as depicted in Figure 1, left; a 0-simplex contains just a single vertex. For now we won't associate specific locations with the vertices—for instance, Figure 1, right is another perfectly good depiction of these simplices.[2] For this reason we call these simplices *abstract*—they don't pin down some concrete shape in space, but just tell us (abstractly) how vertices are connected up. For this reason we can go as high as we like without having to think about how this thing looks in space (3-simplex, 4-simplex, 5-simplex, ...). We also don't care (for now) about the order in which the vertices[3] are specified: for instance $\{2, 3, 4\}$ and $\{3, 2, 4\}$ specify the same 2-simplex as $\{3, 4, 2\}$. Note that for convenience, we will often identify any vertex $i \in V$ with the corresponding 0-simplex $\{i\} \in \mathcal{K}$.

Any (nonempty) subset of a simplex is another simplex, which we call a *face*; a *strict* subset is called a *proper* face. For instance, $\{2, 3\}$ is a proper face of $\{3, 4, 2\}$, and $\{2, 3, 4\}$ is a face of $\{3, 4, 2\}$, but not a proper one.

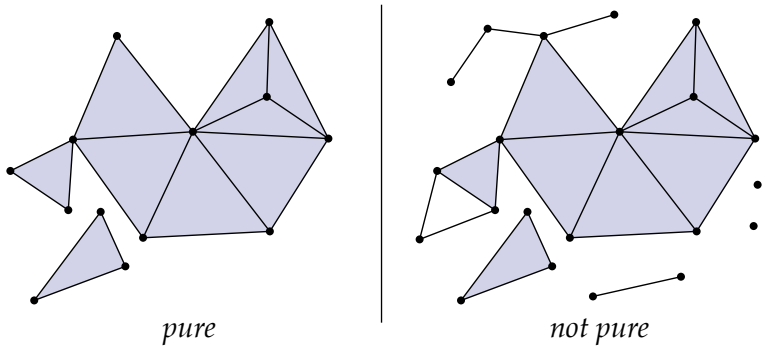
An *abstract simplicial complex* is, roughly speaking, just a collection of abstract simplices. However, we will put a very basic condition on this collection that ensures we can work with it in a natural way—and which ultimately helps us to make connections with smooth surfaces. In particular, we will say that a collection of simplices $\mathcal{K}$ is a simplicial complex if for every simplex

[2]Note: the plural of *simplex* is *simplices*—not "simplexes"!

[3]The plural of *vertex* is *vertices*—not "vertexes"!

$\sigma \in \mathcal{K}$, every face $\sigma' \subseteq \sigma$ is also contained in $\mathcal{K}$. For instance, a bunch of triangles do not constitute a simplicial complex; you have to include their edges and vertices as well. We will often assume that a simplicial complex is *finite* (*i.e.*, contains finitely many simplices), though in principle there's no reason you can't consider an infinite complex—say, a triangulation of the whole Euclidean plane.

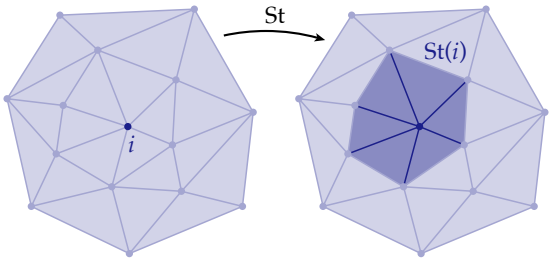
A *subcomplex* $\mathcal{K}'$ of a simplicial complex $\mathcal{K}$ is a subset that is also a simplicial complex. For instance, a single edge is not a subcomplex of any complex, but an edge with its two vertices is a subcomplex. A complex $\mathcal{K}$ is a *pure k-simplicial complex* if every simplex $\sigma' \in \mathcal{K}$ is contained in some simplex of degree k (possibly itself). For instance, a bunch of triangles with edges and vertices hanging off the side or floating around by themselves is not pure:



In the end, we end up with a pretty simple (and abstract) object: an abstract simplicial complex is just a subset of the integers, closed under the operation of taking subsets. This deceivingly simple object makes it possible to exactly encode the topology of any surface, no matter how complicated. To do discrete differential *geometry* we'll eventually need to associate some kind of shape with a simplicial complex. But for now there are already some interesting things we can say about surfaces in the purely combinatorial setting.

2.2. Anatomy of a Simplicial Complex: Star, Closure, and Link

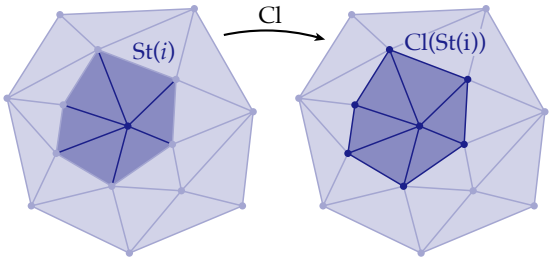
When working with simplicial complexes,[4] it's helpful to be able to quickly and succinctly refer to various elements and regions. Let's start out by considering just a single vertex $i \in V$. The *(simplicial) star* of this vertex, denoted $\mathrm{St}(i)$ is the collection of all simplices $\sigma \in \mathcal{K}$ such that $i \in \sigma$ [5]. Consider for instance the following example:



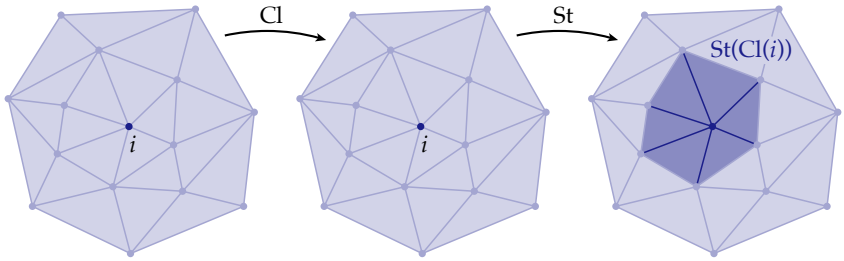
[4]Note: the plural of *complex* is *complexes*—not "complices"! Welcome to the English language.

[5]Be careful to distinguish this star from the *Hodge star*, which is a completely different object that we'll study later.

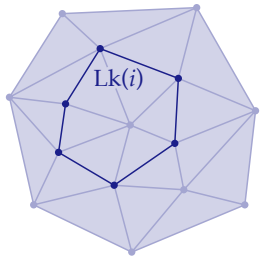
From this picture, one gets the sense that the $\mathrm{St}(i)$ is sort of the "local neighborhood" of i. However, this neighborhood is not itself a simplicial complex, since it doesn't contain the "outer" edges. To get such a complex, we can consider the *closure* Cl of $\mathrm{St}(i)$, which is the smallest subcomplex of $\mathcal{K}$ containing $\mathrm{St}(i)$:



What if we go the other direction, and take the closure before the star? In other words, we first consider the closure $\mathrm{Cl}(i)$ which is the smallest subcomplex of $\mathcal{K}$ containing i. Since $\{i\}$ has no proper faces, the closure is just the vertex itself. If we then take the star, we therefore get the same picture as the first one above, *i.e.*, $\mathrm{St}(\mathrm{Cl}(i)) = \mathrm{St}(i)$:



The only difference between these two sets is the ring of outer edges that was initially missing from our subcomplex. We give this set a special name: the *link* $\mathrm{Lk}(i) = \mathrm{Cl}(\mathrm{St}(i)) \setminus \mathrm{St}(\mathrm{Cl}(i))$ (where $A \setminus B$ denotes the *set difference*, *i.e.*, all the elements of A that are not also in B):

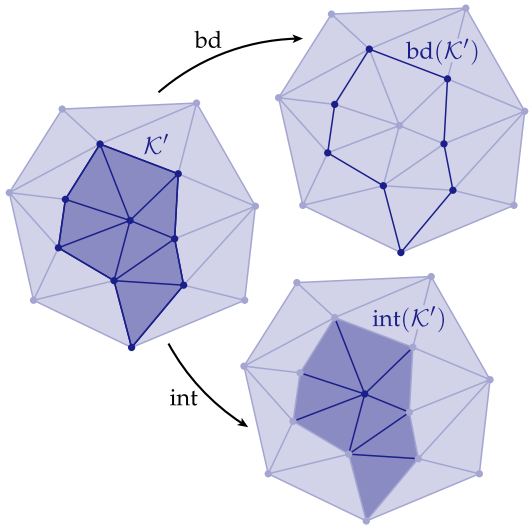


$\mathrm{Lk}(i) = \mathrm{Cl}(\mathrm{St}(i)) \setminus \mathrm{St}(\mathrm{Cl}(i))$

More generally for any subset S of a simplicial complex $\mathcal{K}$ (not necessarily a subcomplex) we have the following definitions:

- The *star* $\mathrm{St}(S)$ is the collection of all simplices in $\mathcal{K}$ that contain any simplex in S.
- The *closure* $\mathrm{Cl}(S)$ is the smallest (*i.e.*, fewest elements) subcomplex of $\mathcal{K}$ that contains S.
- The *link* $\mathrm{Lk}(S)$ is equal to $\mathrm{Cl}(\mathrm{St}(S)) \setminus \mathrm{St}(\mathrm{Cl}(S))$.

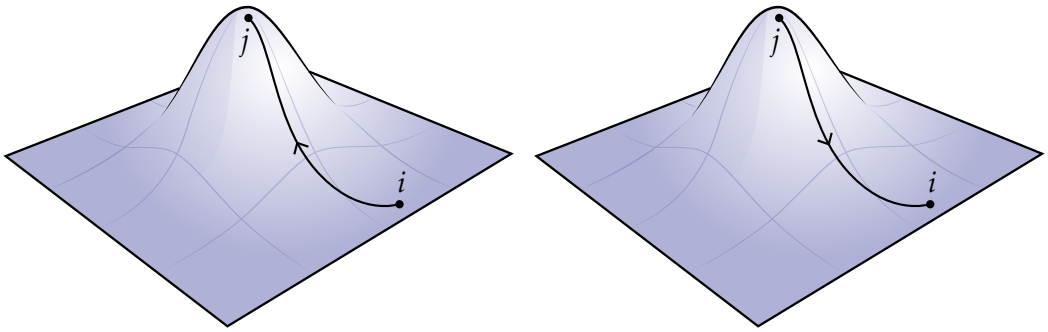
Another closely related object is the *boundary* $\mathrm{bd}(\mathcal{K}')$ of a pure k-subcomplex $\mathcal{K}' \subseteq \mathcal{K}$. The boundary is the closure of the set of all simplices σ that are proper faces of exactly one simplex of $\mathcal{K}'$. This definition naturally captures what you might think of as the "boundary" of a set. For instance:



The *interior* $\mathrm{int}(\mathcal{K}') = \mathcal{K}' \setminus \mathrm{bd}(\mathcal{K}')$ is then everything *but* the boundary (as pictured above).

In general, these operations (star, closure, link, boundary, and interior) provide a natural way to talk about and navigate any kind of simplicial complex in any dimension. In fact, they are a fair bit *more* general than what we need to just talk about simple combinatorial surfaces. In a little bit, we'll introduce a different way to navigate around combinatorial surfaces called a *half edge mesh*, which is in some ways "slicker" and easier to work with if we don't care about the general case. But to do so, we first need to define what we really mean by a *combinatorial surface*—and to do so, we'll need the star, closure, and link!

2.2.1. Oriented Simplicial Complex.



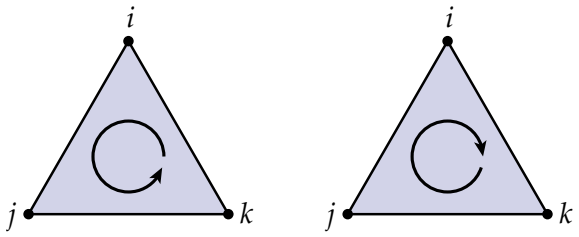
So far we've assumed that the order of vertices in a simplex doesn't matter, and specified simplices using *sets*. For instance, $\{i, j, k\}$ is a triangle with vertices i, j, and k, and $\{j, i, k\}$ or $\{k, j, i\}$ describe the same triangle. But in many cases, we'll want to make a distinction between simplices with different *orientation*, because the orientation encodes some information about a quantity we're

measuring or computing. For instance: the change in altitude from the *bottom* of a hill to the *top* of the hill is opposite the change in altitude from the *top* of the hill to the *bottom* of the hill.

To capture the notion of orientation, we'll start by replacing our unordered sets with *ordered tuples*. For instance, if $i, j \in V$ are two vertices sharing an edge, then we have two distinct ordered tuples (i, j) and (j, i). The first tuple describes an *oriented edge* pointing from i to j, whereas the second tuple points from j to i. For higher-degree simplices (triangles, tetrahedra, *etc.*), the story gets just a bit more complicated. For instance, consider three vertices $i, j, k \in V$ sharing a common triangle. Instead of the single ordered set $\{i, j, k\}$, we now have six possible ordered tuples:

$$\begin{array}{ll} (i,j,k) & (i,k,j) \\ (j,k,i) & (j,i,k) \\ (k,i,j) & (k,j,i) \end{array}$$

Each of these tuples specifies some way of walking around the triangle. For instance, we could first visit i, then j, then k. Or we could first visit j, then i, then k. If we consider these six different possibilities, what we notice is that they fall into two obvious categories: we either walk "clockwise" or "counter-clockwise" around the triangle. These two possibilities describe the two possible orientations for our triangle:



Since we don't care about the starting point, an orientation of a simplex is really an *equivalence class* of ordered tuples—in this case, the first column of tuples above are all equivalent (clockwise), and the second column are all equivalent (counter-clockwise). Therefore, to specify an oriented triangle we'll just give a representative triple of indices, rather than singling out one particular tuple. For instance, we'll say that

$$ijk := \{(i,j,k), (j,k,i), (k,i,j)\}$$

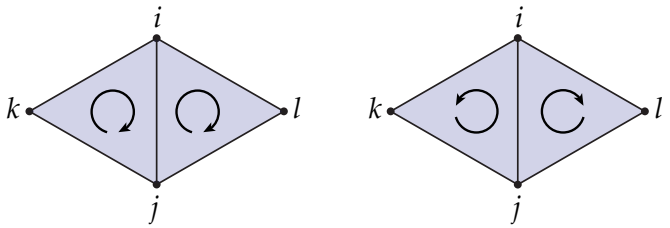
and

$$ikj := \{(i,k,j), (j,i,k), (k,j,i)\}.$$

Hopefully this makes sense: a tuple written with parentheses describes one particular way of walking around a triangle; a raw triple of indices refers to all tuples with equivalent orientation. We'll basically always use the latter, since it gives us *just* enough information to know which oriented simplex we're talking about: its vertices, and its orientation.

More generally, for any k-simplex we have two possible orientations: the set of all *even* permutations of its vertices, and the set of all *odd* permutations of its vertices. For instance, with an edge we have just $ij = \{(i,j)\}$ and $ji = \{(j,i)\}$. For a triangle we have the two orientations ijk and jik given above. For a tetrahedron we have $ijkl$ and $jikl$. And so on. The only exception is 0-simplex, where there is only one way to write the list of vertices (*i.e.*, $i = \{i\}$). A 0-simplex therefore has only one possible orientation.

If two oriented simplices share vertices, then we can talk about their *relative orientation*. For instance, the oriented triangles ijk and jil have the same orientation (they are both "clockwise") whereas ijk and ijl are oppositely oriented:



consistent orientation *inconsistent orientation*

Likewise, the edge ij has the same clockwise orientation as ijk, whereas ji has the opposite orientation. In general, if two k-simplices σ_1, σ_2 share exactly $k-1$ vertices, then they have the same orientation if their restrictions to these shared vertices are oppositely oriented. For any oriented simplex σ a proper face σ' has the same orientation if σ' appears in some even permutation of σ. An important special case is 0- and 1-simplices: an oriented edge ij has the same orientation as j, but the opposite orientation of i; this convention captures the fact that ij goes *from i to j*.

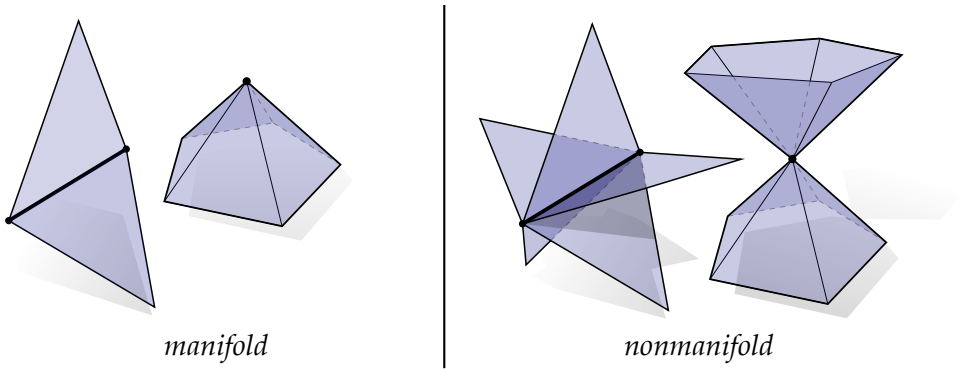
An *(abstract) oriented simplicial complex* is an abstract simplicial complex where each simplex is assigned an orientation. I.e., we start with an ordinary simplicial complex, and simpy pick one of two orientations for each simplex. There are no conditions on these orientations: they can be assigned arbitrarily. Though (when possible) it's often convenient to assume that k-simplices sharing common $(k-1)$-faces have the same orientation (for instance, that all triangles in a planar triangulation have clockwise orientation).

2.3. Simplicial Surfaces

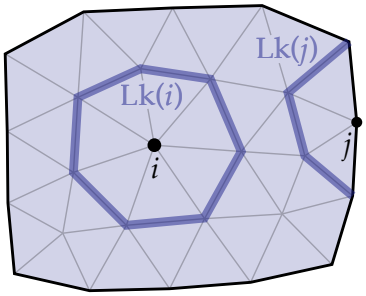
As mentioned at the beginning of this section, a general simplicial complex is a bit *more* general than what we need to study ordinary shapes (a hat, a face, a heart, a banana), which are all pretty well-captured by surfaces. Instead, it is often enough to work with an *abstract simplicial surface*. An abstract simplicial surface is a pure simplicial 2-complex where the link of every vertex is a single loop of edges, or equivalently, where the star of every vertex is a combinatorial disk made of triangles. The fact that every vertex has a "disk-like" neighborhood captures the basic idea of a topological surface; we therefore say that such a complex is *manifold*.[6]

Unlike a general simplicial complex, a simplicial surface can't have stuff like three triangles meeting at an edge, or multiple "cones" of vertices meeting at a vertex. We will henceforth call such configurations *nonmanifold*:

[6]Note that at this point we could very easily put a topology on our complex that makes it into a topological surface in the usual sense. But the interesting point is that we don't *have* to define a topology in order to understand a lot of the behavior of topological surfaces: the purely combinatorial description will take us surprisingly far.

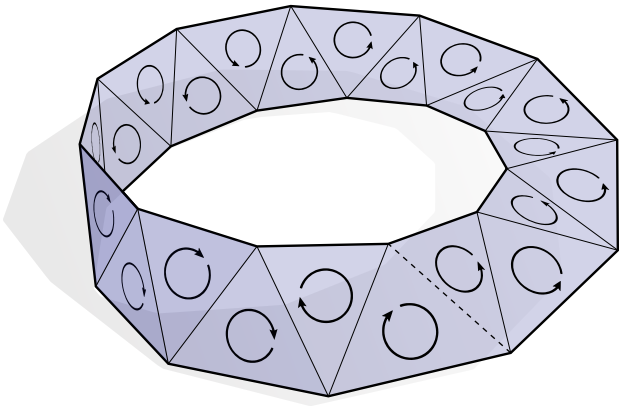


We can extend our definition a bit to a simplicial surface *with boundary* by also allowing the link to be a simple path of edges, rather than a loop:



For any simplicial surface $\mathcal{K}$, its boundary $\mathrm{bd}(\mathcal{K})$ will always be a collection of (zero or more) closed loops.

An *oriented simpicial surface* is an abstract simplicial surface where we can assign a consistent orientation to every triangle, *i.e.*, where any two triangles that share a common edge are given the same orientation. We will henceforth assume that any simplicial surface whose faces *can* be consistently oriented *will* be consistently oriented. Is a consistent orientation always available? At first glance, it seems easy: start with an arbitrary triangle, assign it an arbitrary orientation, and now "grow outwards," assigning a consistent orientation to every triangle you encounter. The problem is that, at some point, you may loop back around and discover that there is no way to assign an orientation to a new triangle that is compatible with all previous orientations. Consider for instance this combinatorial *Möbius band*:



Such *unorientable* surfaces don't come up all that often in nature—though it's certainly worth being aware that they can!

Our definition of a simplicial surface easily extends to higher dimensions: a *(combinatorial or abstract) simplicial n-manifold* is a pure simplicial n-complex where the link of every vertex is a simplicial $(n-1)$-sphere. A simplicial n-sphere is just a (simplicial) triangulation of the n-dimensional sphere

$$\mathbb{S}^n := \{x \in \mathbb{R}^{n+1} : |x| = 1\},$$

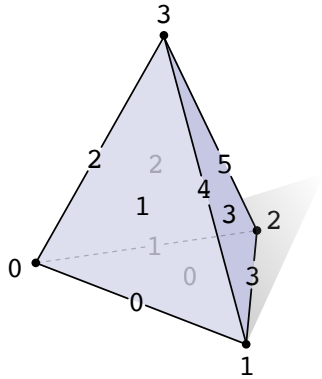
i.e., the set of all points unit distance from the origin in n-dimensional space. For instance, $\mathbb{S}^2$ is just the ordinary unit sphere; $\mathbb{S}^1$ is the unit circle; and $\mathbb{S}^0$ is nothing more than a pair of points. A simplicial surface is then a simplicial 2-manifold: every link is a simplicial 1-sphere, *i.e.*, a closed loop of edges. A simplicial 3-manifold is a tetrahedral mesh where every vertex is enclosed by a triangulation of the sphere. And so on. How about a simplicial 1-manifold? This would just mean the link of every vertex is a pair of points; hence, a simplicial 1-manifold must be a collection of closed loops (do you see why?).

From here there's a *lot* more we could say about simplicial surfaces, but this will get the ball rolling for now. In particular, it's enough to let us define a *halfedge mesh*, which will be our basic way of navigating around simplicial (and more generally, polyhedral) surfaces.

2.4. Adjacency Matrices

One nice way to encode an abstract simplicial complex is using *adjacency matrices*, which will make it easy to do computation on simplicial complexes (by way of numerical linear algebra). These matrices are also closely linked to the *discrete differential forms* which provides the foundations for many of our geometric algorithms.

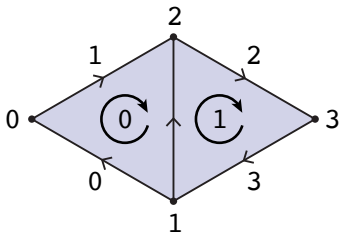
The first thing we have to do is assign distinct *indices* to simplices of each degree. For instance, if we have a complex $\mathcal{K}$ comprised of vertices V, edges E, and triangles F, then we might assign indices $0, \ldots, |V|-1$ to the vertices, $0, \ldots, |E|-1$ to the edges, $0, \ldots, |F|-1$ and to the triangles. It doesn't matter which indices get assigned to which triangles, as long as each index is used only once for each degree of simplex. For instance, here's a simplicial 2-complex where we've indexed all the vertices, edges, and faces:



To record how simplices are connected up, we're going to store one matrix A_0 that says which edges contain which vertices, another matrix A_1 that says which triangles contain which edges, and so on. We'll put a "1" in row r, column c of A_0 if the rth edge contains the cth vertex; all other entries get a "0". Likewise, we'll put a "1" in row r, column c of A_1 if the rth triangle contains the cth edge. And so on. Hence, the number of columns in adjacency matrix A_k is the same as the

number of k-simplices; the number of rows is the number of $(k+1)$-simplices. In this example, our matrices look like this: One important thing to notice here is that—especially for a very *large* complex with a relatively *small* number of connections—most of the entries are going to be *zero*. In practice, it's therefore essential to use a *sparse matrix*, *i.e.*, a data structure that efficiently stores only the location and value of nonzero entries. The design of sparse matrix data structures is an interesting question all on its own, but conceptually you can imagine that a sparse matrix is simply a list of triples (r, c, x) where $r, c \in \mathbb{N}$ specify the row and column index of a nonzero entry and $x \in \mathbb{R}$ gives its value.

If we have an *oriented* simplicial complex, we can also build *signed* adjacency matrices, which keep track of relative orientation in addition to connectivity. The only change is that the sign of each nonzero entry will depend on the relative orientation of the two corresponding simplices: $+1$ if they have the same orientation; -1 if they have opposite orientation. For instance, here are the signed adjacency matrices for a pair of consistently-oriented triangles sharing a common edge:



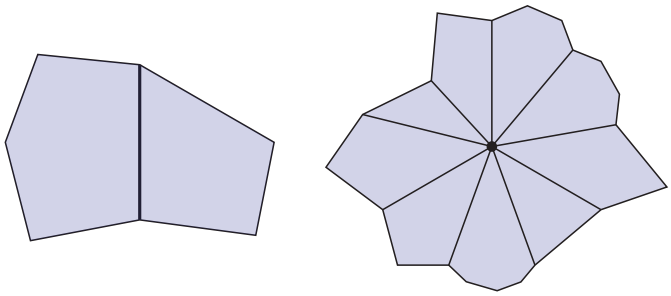
2.5. Halfedge Mesh

As discussed above, many of the shapes we encounter in the natural world are well-captured by manifold, orientable surfaces. Our third and final encoding of surface combinatorics, the *halfedge mesh*, takes advantage of the special structure of such surfaces. In some ways, this encoding is less general than, say, the adjacency matrix representation: we cannot capture edges that dangle off the side of a triangulation, or surfaces like the Möbius band, or higher-dimensional shapes (*e.g.*, volumes rather than surfaces). On the other hand, it *will* allow us to describe combinatorial surfaces made of general polygons rather than just triangles, and in certain important cases even allows us to use *fewer* triangles (or polygons) than is possible with any simplicial complex. (Formally, a halfedge mesh allows us to encode a surface as a 2-dimensional *CW complex*; see [**Hat02**, Chapter 0] for a definition.)

From our discussion in Section 2.3 we can notice a few things about any oriented simplicial surface $\mathcal{K}$ (which for now we'll assume has no boundary):

- every edge is contained in two polygons, and
- the edges around a vertex can be given a cyclic order.

In fact, these same statements hold if our surface is made out of n-sided polygons rather than just 3-sided triangles:



The halfedge mesh takes advantage of this special structure to provide a particularly nice description of surfaces. The basic idea is to consider that for every unoriented edge $\{i, j\}$ between vertices i and j, we have two oriented edges $ij \neq ji$ which in this context we refer to as *halfedges*. We'll use H to denote the set of all halfedges; note that for a surface without boundary $|H| = 2|E|$, *i.e.*, we have twice as many halfedges as edges.

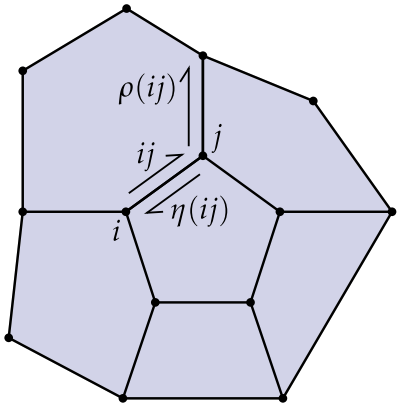
We can use information about how halfedges are connected up to describe an oriented simplicial surface. In particular, we have two key functions: *twin* and *next*. The twin function is the map $\eta : H \to H$ such that

$$\eta(ij) = ji,$$

i.e., that just takes any halfedge to the halfedge with the same vertices but in the opposite direction. The next function is the map $\rho : H \to H$ such that

$$\rho(ij) = jk \qquad \forall ijk \in \mathcal{K},$$

i.e., that takes each halfedge of an oriented triangle ijk to the next halfedge around this triangle. These maps are reasonably straightforward to figure out if we're handed some other description of the surface (such as an oriented simplicial complex or a pair of adjacency matrices).



Going the other direction, we can easily figure out the vertices, edges, and faces of a polygonal mesh from nothing more than the two maps ρ and η. For instance, to get a face we can start with some halfedge i_1i_2 and use the map ρ to get the next halfedge $i_2i_3 = \rho(i_1i_2)$, then the next halfedge $i_3i_4 = \rho(i_2i_3)$, and then the next, until we eventually get back to i_1i_2. In other words, the faces of the mesh are described by the *orbits* of the "next" map ρ. What do the orbits of the "twin" map η give us? Well, starting with ij we get $ji = \eta(ij)$ and then $ij = \eta(ji)$. Hence, the orbits of η describe the edges. To get the vertices, we can instead consider the orbits of the map $\rho \circ \eta$, *i.e.*, first we get the twin halfedge, then the next halfedge, then the twin, then the next, ..., until we get back to the beginning. To summarize:

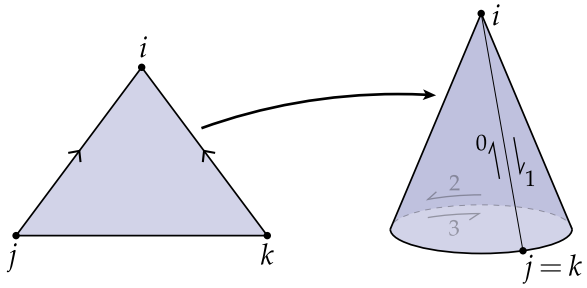
- the **faces** are orbits of ρ,
- the **edges** are orbits of η, and

- the **vertices** are orbits of $\rho \circ \eta$.

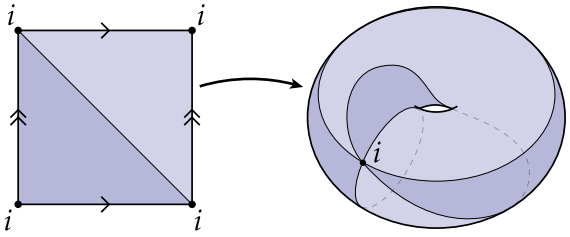
In fact, *any* pair of maps ρ, η which satisfy some very basic properties will describe a valid combinatorial surface. All we need is that (i) the set H has an even number of elements, (ii) ρ and η are both permutations of this set, and (iii) η is an *involution* with no *fixed points*, *i.e.*, $\eta(\eta(ij)) = ij$ for all $ij \in H$, and $\eta(ij) \neq ij$ for any $ij \in H$. This last condition is just common sense: it says that the twin of your twin is yourself, and you are not your own twin. (If one of these statements were not true about your real, biological twin, you'd be in serious trouble!) As long as ρ and η satisfy these basic properties, we can trace out their orbits and recover a combinatorial surface.

In fact, if we index our halfedges in a special way we don't even really have to worry about η. In particular, suppose we assign the indices 0 and 1 to the first pair of halfedges, 2 and 3 to the next pair of halfedges, and so on. Then η has a very simple description: the twin of any *even* number h is $h + 1$; the twin of any *odd* number h is $h - 1$. The combinatorics of the surface are then described entirely by the permutation ρ. In summary, then, ***every*** *permutation of an even number of things describes a combinatorial surface!* Pretty weird. But true! Think about that next time you encounter a permutation.

Note that even for surfaces made out of triangles, a halfedge mesh can describe triangulations that a simplicial complex cannot. Consider for instance the following example:

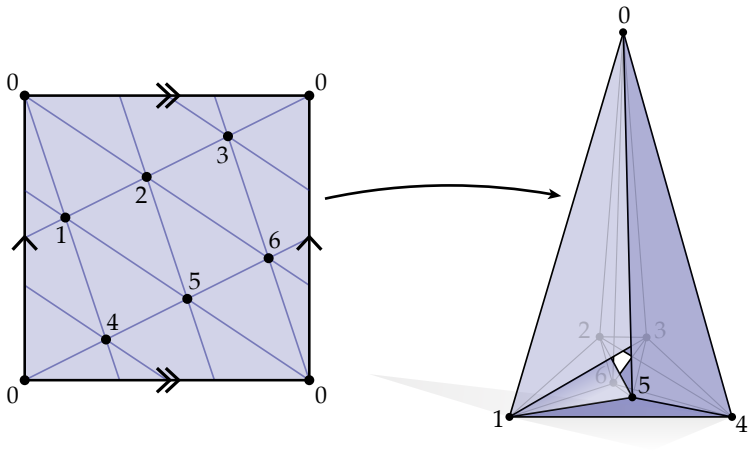


Here we obtain a cone by gluing two edges of the triangle ijk together. If we fill in the bottom of the cone with a circular disk, then overall we have four halfedges: three halfedges h_0, h_1, h_2 going around the triangle, and one halfedge h_3 going around the disk. The *next* map is given by $\rho(h_0) = h_1, \rho(h_1) = h_2, \rho(h_2) = h_0$ and $\rho(h_3) = h_3$, *i.e.*, we have a loop around the triangle, and a loop around the disk. The *twin* map is determined by the relationships $\eta(h_0) = h_1$ and $\eta(h_2) = h_3$, *i.e.*, two of the triangle halfedges are glued together, and the remaining triangle halfedge is glued to the halfedge around the disk. There's no way to describe this triangulation using a simplicial complex (oriented or otherwise): a simplicial complex only allows us to specify the triangle ijk; we have no further opportunity to specify how the edges get glued together. Likewise, including the disk is totally out of the question: it's not even an ordinary polygon; more like a "unigon!" Here's another interesting example:



This time we have a torus made of two distinct triangles, but only one vertex. (Can you write down corresponding maps η and ρ?) There's clearly no way to describe this triangulation using a simplicial complex: for one thing, the "set" $\{i, i, i\}$ is not a set: it doesn't have three *distinct* vertices.

Even if we allow repeats (*i.e.*, *multi-sets*), we have no way in a simplicial complex to distinguish between the two distinct triangles "*iii*" and "*iii*", and no way to explain how these triangles get glued together. A halfedge mesh handles these cases with ease, allowing us to describe interesting spaces with fewer elements. In contrast, to describe a torus using a simplicial complex we need at least 7 vertices, 21 edges, and 14 triangles:



Here we've drawn this triangulation in two ways: on the left, we draw it on a square and imagine that left/right and bottom/top sides get glued together. Amazingly enough, this same triangulation can be drawn using straight lines and flat, non-intersecting triangles in $\mathbb{R}^3$, as depicted on the right—something known as the *Császár polyhedron*. In either case, it's a lot more complicated than the two-triangle decomposition of the torus we obtained via a halfedge mesh.

At a practical level, the halfedge description will provide the basic data structure for the algorithms we'll implement in these notes. By chasing "next" and "twin" halfedges around, you can easily access any mesh element your heart desires. One final question though: how do we deal with surfaces that have boundary (such as a disk or annulus)? These would seem to violate one of our basic axioms, that every edge is contained in exactly two polygons. The easy answer is: *just treat each boundary component as a single polygon with many sides*. In other words, turn your surface with boundary into a surface *without* boundary by simply "filling in the holes" (and marking these extra polygons in some way). From there you have a surface without boundary, and it's just business as usual.

2.6. Written Exercises

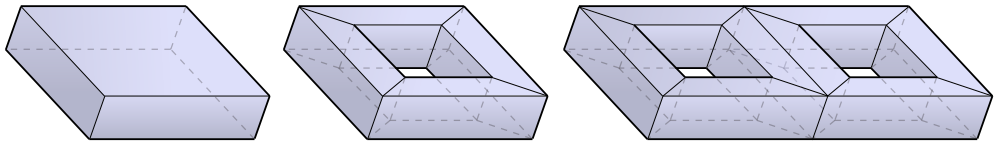
EXERCISE 2.1 **Euler's Polyhedral Formula**

The *Euler characteristic* $\chi = V - E + F$ is a *topological invariant*: it remains the same even if we make small local changes to the connectivity (like inserting a new vertex in the middle of a polygon). It changes only if there is a *global* change to the topology, like adding an extra component, or an additional handle. Show in particular that for any polygonal disk with V vertices, E edges, and F faces, the following relationship holds:

$$V - E + F = 1$$

and explain, then, why $V - E + F = 2$ for any polygonal sphere.

Hint: use induction. Note that induction is generally easier if you start with a given object and decompose it into ***smaller*** *pieces rather than trying to make it* ***larger****, because there are fewer cases to think about.*

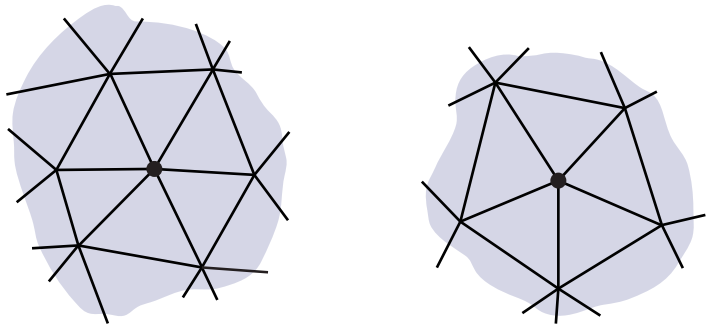


sphere ($g = 0$) *torus* ($g = 1$) *double torus* ($g = 2$)

Clearly not all surfaces look like disks or spheres. Some surfaces have additional *handles* that distinguish them topologically; the number of handles g is known as the *genus* of the surface (see illustration above for examples). In fact, among all surfaces that have no boundary and are connected (meaning a single piece), compact (meaning closed and contained in a ball of finite size), and orientable (having two distinct sides), the genus is the *only* thing that distinguishes two surfaces. A more general formula applies to such surfaces, namely

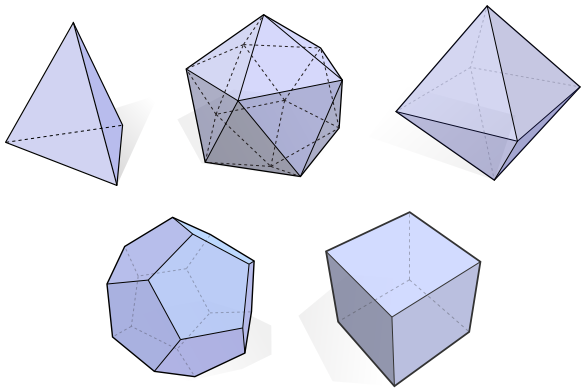
$$V - E + F = 2 - 2g,$$

which is known as the *Euler-Poincaré formula.*



regular *irregular*

The *valence* of a vertex in a combinatorial surface is the number of edges that contain that vertex. A vertex of a *simplicial* surface is said to be *regular* when its valence equals six. Many numerical algorithms (such as subdivision) exhibit ideal behavior only in the regular case, and generally behave better when the number of irregular valence vertices is small. The next few exercises explore some useful facts about valence in combinatorial surfaces.



EXERCISE 2.2 **Platonic Solids**

Even the ancient Greeks were interested in regular meshes. In particular, they knew that there are only five genus-zero polyhedra where all faces have the same number of sides, and the same number of faces meet at every vertex. These polyhedra are the *Platonic solids*: the tetrahedron, icosahedron, octahedron, dodecahedron, and cube. Show that this list is indeed exhaustive. *Hint: you do not need to use any facts about lengths or angles; just connectivity.*

EXERCISE 2.3 **Regular Valence**

Show that the only (connected, orientable) simplicial surface for which every vertex has regular valence is a torus ($g = 1$). You may assume that the surface has finitely many faces. *Hint: apply the Euler-Poincaré formula.*

EXERCISE 2.4 **Minimum Irregular Valence**

Show that the minimum possible number of irregular valence vertices in a (connected, orientable) simplicial surface K of genus g is given by

$$m(K) = \begin{cases} 4, & g = 0 \\ 0, & g = 1 \\ 1, & g \geq 2, \end{cases}$$

assuming that all vertices have valence at least three and that there are finitely many faces. *Note:* you do not actually have to *construct* the minimal triangulation; just make an argument based on the Euler-Poincaré formula.

EXERCISE 2.5 **Mean Valence (Triangle Mesh)**

Show that the mean valence approaches six as the number of vertices in a (connected, orientable) simplicial surface goes to infinity, and that the ratio of vertices to edges to triangles hence approaches

$$V : E : F = 1 : 3 : 2.$$

You may assume that the genus g remains fixed as the number of vertices increases. *Hint: Euler-Poincaré formula!*

EXERCISE 2.6 **Mean Valence (Quad Mesh)**

Similar to the previous exercise, consider a *quad mesh*, *i.e.*, a combinatorial surface made entirely out of four-sided quadrilaterials rather than three-sided triangles. Letting Q denote the number of quadrilaterals, give an expression for the ratio

$$V : E : Q$$

in the limit as the number of vertices approaches infinity. You may again assume that the genus remains fixed.

Knowing the approximate ratios of mesh elements can be useful when making decisions about algorithm design (e.g., it costs about three times as much to store a quantity on edges as on vertices), and simplifies discussions about asymptotic growth (since the number of different element types are essentially related by a constant). Similar ratios can be computed for a *tetrahedral* mesh, though here one has to be a bit more approximate:

EXERCISE 2.7 **Mean Valence (Tetrahedral).**

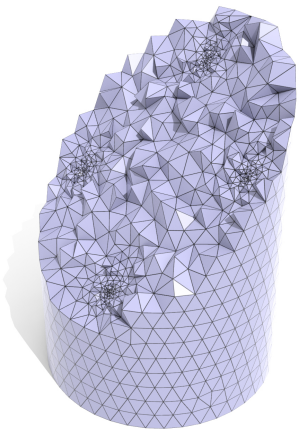
Letting V, E, F, and T be the number of vertices, edges, triangles, and tetrahedra in a manifold simplicial 3-complex, come up with a rough estimate for the ratios

$$V : E : F : T$$

as the number of elements goes to infinity. For tet meshes, there is a formula analogous to Euler's polyhedral formula: $V - E + F - T = c$, for some constant c that depends only on the global topology (number of handles, *etc.*). To get a rough estimate, you should pretend that the link $\mathrm{Lk}(i)$ of every vertex $i \in V$ is a combinatorial icosahedron. Since you care about the asymptotic behavior, you can safely ignore boundary vertices. *Hint: which ratios can you figure out easily?*

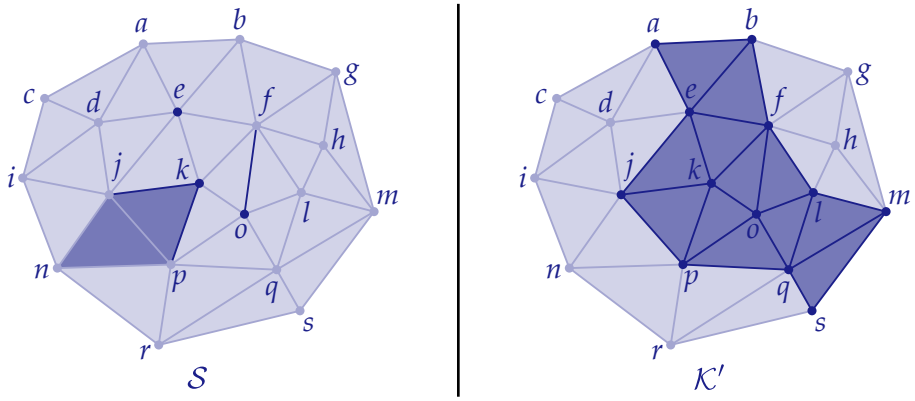
Here are some statistics on large-ish tetrahedral meshes coming from real data. Do they roughly match your estimated ratios? (You should *NOT* worry if they don't match exactly!)

	V	E	F	T
mesh #1	9344	64814	109660	54189
mesh #2	10784	69807	114345	55323
mesh #3	13630	97271	166689	83047
mesh #4	20514	144661	245764	121616
mesh #5	21222	146117	245959	121063
mesh #6	21464	144263	240663	117865
mesh #7	22933	163360	279634	139206
mesh #8	24522	175177	300272	149616
mesh #9	37483	262803	447463	222143



EXERCISE 2.8 **Star, Closure, and Link**

For the subset $\mathcal{S}$ indicated below in dark blue (consisting of three vertices, three edges, and two triangles), give the star $\mathrm{St}(\mathcal{S})$, the closure $\mathrm{Cl}(\mathcal{S})$, and the link $\mathrm{Lk}(\mathcal{S})$, either by drawing pictures or providing a list of simplices in each set.



EXERCISE 2.9 **Boundary and Interior**

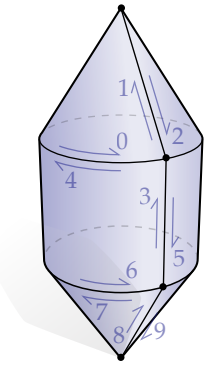
For the subset $\mathcal{K}'$ indicated above in dark blue (consisting of 12 vertices, 23 edges, and 12 triangles), give the boundary $\mathrm{bd}(\mathcal{K}')$ and the interior $\mathrm{int}(\mathcal{K}')$, either by drawing pictures or providing a list of simplices in each set.

EXERCISE 2.10 **Surface as Permutation**

For the combinatorial surface pictured below, give the *twin* and *next* permutations η and ρ (*resp.*) by filling out the following tables:

h	0	1	2	3	4	5	6	7	8	9
$\eta(h)$										

h	0	1	2	3	4	5	6	7	8	9
$\rho(h)$										



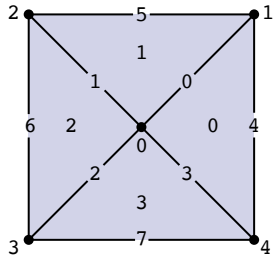
EXERCISE 2.11 **Permutation as Surface**

For the permutation ρ given below, describe the combinatorial surface it describes—either in words, or by drawing a picture. You should assume that η is determined as described in Section 2.5, *i.e.*, the twin of an even halfedge h is $h+1$; the twin of an odd halfedge h is $h-1$.

h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\rho(h)$	8	2	14	4	12	6	10	0	7	15	5	9	3	11	1	13

EXERCISE 2.12 **Surface as Matrices**

Give the adjacency matrices A_0 and A_1 for the simplicial disk depicted in the figure below.



For the next three exercises you may be as rigorous or as informal as you like, so long as you correctly convey the core reason why each of the statements is true.

EXERCISE 2.13 **Classification of Simplicial 1-Manifolds**

Explain why every simplicial 1-manifold (possibly with boundary) cannot contain anything other than paths of edges and closed loops of edges.

EXERCISE 2.14 **Boundary Loops**

Explain why the boundary of any simplicial surface (*i.e.*, any simplicial 2-manifold) always has to be a collection of closed loops.

EXERCISE 2.15 **Boundary Has No Boundary**

Explain why the boundary $\mathrm{bd}(\mathcal{K})$ of a simplicial manifold has no boundary. In other words, why does $\mathrm{bd}(\mathrm{bd}(\mathcal{K})) = \emptyset$? Here you may assume that a *simplicial manifold with boundary* means a pure simplicial k-complex where the link of every vertex is either a simplicial $(k-1)$-sphere (in which case it's an interior vertex), or a simpicial $(k-1)$-ball (in which case it's a boundary vertex). For instance, when $k=2$ the link of a boundary vertex will just be a path of edges.

2.7. Coding Exercises

To get a feel for how we're going to work with the combinatorics of a surface in practice, we'll now write some code that nicely ties together a bunch of the ideas discussed above. In particular, given a half edge mesh describing the combinatorics of a manifold triangle mesh, you will build the vertex-edge and edge-face adjacency matrices. These matrices can be used to implement two concepts we've studied:

- the boundary and coboundary operators, and
- the star, closure, and link operators.

Later on, we'll also see that these matrices have an important connection to *discrete differential forms*. Note that the methods below *must* be implemented using these matrices; from here on out you should *not* be doing everything purely in terms of halfedge operations. The input subset of simplices $\mathcal{S}$ will be provided as a data structure containing sets of vertex, edge, and face indices.

CODING 1. Implement the method `assignElementIndices`, which assigns a unique index to each vertex, edge, and face of the triangle mesh. For each type of element, indices should start at zero—for instance, the vertices should be assigned indices $0, \ldots, |V| - 1$, and the edges should be assigned indices $0, \ldots, |E| - 1$, etc. The order doesn't matter at all, so long as the mapping between elements and indices is one-to-one. These indices provide a correspondence between elements of the mesh, and rows/columns of matrices you will build in the next few coding exercises.

CODING 2. Implement the method `buildVertexEdgeAdjacencyMatrix`, which constructs the *unsigned* vertex-edge adjacency matrix $A_0 \in \mathbb{R}^{|E| \times |V|}$ (not the signed one), described in Section 2.4. Since this matrix contains mostly zeros, it must be implemented using a *sparse* matrix data structure (otherwise, computation will become extraordinarily slow on large meshes!).

CODING 3. Implement the method `buildEdgeFaceAdjacencyMatrix`, which constructs the unsigned edge-face adjacency matrix $A_1 \in \mathbb{R}^{|F| \times |E|}$ (just as in the previous exercise).

CODING 4. Implement the methods `buildVertexVector`, `buildEdgeVector`, `buildFaceVector`, which each take a subset $\mathcal{S}$ of simplices as input, and construct a column vector encoding the vertices, edges, or faces (respectively) in that subset. For instance, in `buildVertexVector` you should build a column vector with $|V|$ entries that has a "1" for each vertex in the subset, and "0" for all other vertices.

For the remaining methods, recall that you *must* use the adjacency matrices (as discussed above); you should *not* be implementing these methods directly using the halfedge data structure.

CODING 5. Implement the method `star`, which takes as input a subset $\mathcal{S}$ of simplices, and computes the simplicial star $\mathrm{St}(\mathcal{S})$ of this subset. *Hint: What happens if you apply the two unsigned adjacency matrices in sequence? How do you get* all *the simplices in the star?*

CODING 6. Implement the method `closure`, which finds the closure $\mathrm{Cl}(\mathcal{S})$ of a given subset $\mathcal{S}$.

CODING 7. Implement the method `link`, which finds the link $\mathrm{Lk}(\mathcal{S})$ of a given subset $\mathcal{S}$. *Hint: use the star and the closure!*

CODING 8. Implement the methods `isComplex` and `isPureComplex`, which check whether a given subset $\mathcal{S}$ is a simplicial complex, and a pure simpicial complex, resp. The latter method should return the degree of the complex if it's pure, and -1 otherwise. *Hint: use the* `closure` *method for the first part, plus the adjacency matrices for the second part.*

CODING 9. Implement the method `boundary`, which takes as input a subset $\mathcal{S}$ of simplices, and finds the set of simplices contained in the boundary $\text{bd}(\mathcal{S})$ of this subset. You should first use the method `isPure` to make sure that the given subset is a pure simplicial complex (otherwise, we do not have a straightforward definition for the boundary). *Hint: think carefully about what the result of applying an unsigned adjacency matrix can look like. What do you notice about the simplices that should be included in the output set? See in particular Exercise 9.*

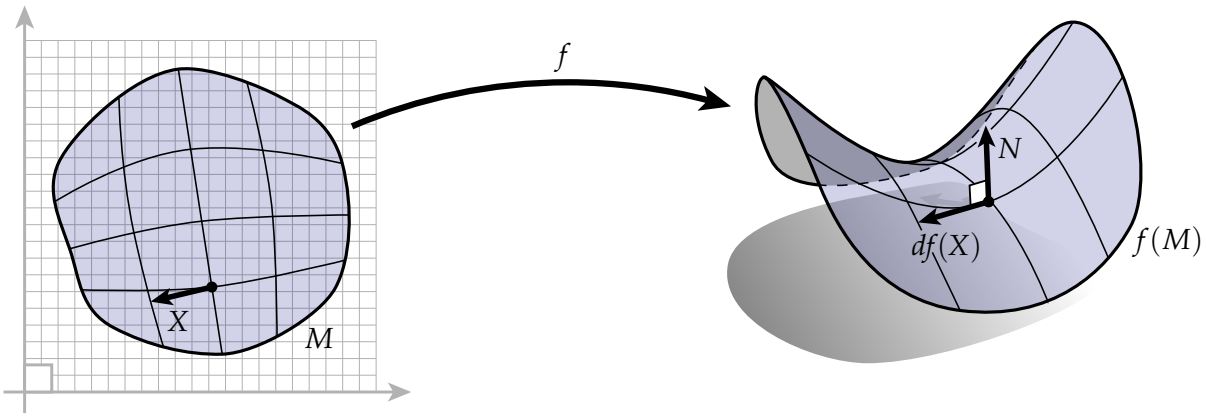
Once everything is implemented you should be able to select a subset of simplices, and click on buttons to apply various operators. You should verify that your code agrees with the examples you computed by hand, and also passes all of the basic tests provided in the test suite.

CHAPTER 3

A Quick and Dirty Introduction to Differential Geometry

3.1. The Geometry of Surfaces

There are many ways to think about the geometry of a smooth surface (using *charts*, for instance) but here's a picture that is well-suited to the way we work with surfaces in the discrete setting. Consider a little patch of material floating in space, as depicted below. Its geometry can be described via a map $f : M \to \mathbb{R}^3$ from a region M in the Euclidean plane $\mathbb{R}^2$ to a subset $f(M)$ of $\mathbb{R}^3$:



The *differential* of such a map, denoted by df, tells us how to map a vector X in the plane to the corresponding vector $df(X)$ on the surface. Loosely speaking, imagine that M is a rubber sheet and X is a little black line segment drawn on M. As we stretch and deform M into $f(M)$, the segment X also gets stretched and deformed into a different segment, which we call $df(X)$. Later on we can talk about how to explicitly express $df(X)$ in coordinates and so on, but it's important to realize that fundamentally there's *nothing deeper to know about the differential than the picture you see here*—the differential simply tells you how to stretch out or *"push forward"* vectors as you go from one space to another. For example, the length of a tangent vector X pushed forward by f can be expressed as

$$\sqrt{df(X) \cdot df(X)},$$

where $\cdot$ is the standard inner product (a.k.a. *dot product* or *scalar product*) on $\mathbb{R}^3$. Note that this length is typically *different* than the length of the vector we started with! To keep things clear, we'll use angle brackets to denote the inner product in the plane, e.g., the length of the original vector would be $\sqrt{\langle X, X \rangle}$. More generally, we can measure the inner product between any *two* tangent vectors $df(X)$ and $df(Y)$:

$$g(X, Y) = df(X) \cdot df(Y).$$

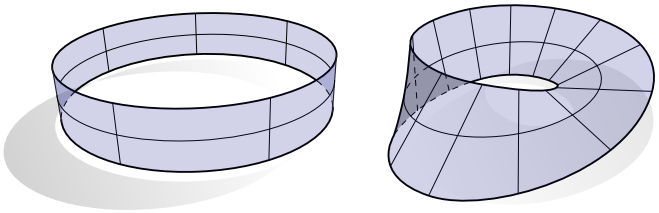
The map g is called the *metric* of the surface, or to be more pedantic, the *metric induced by* f. Note that throughout we will use $df(X)$ interchangeably to denote both the pushforward of a single

vector or an entire *vector field*, *i.e.*, a vector at every point of M. In most of the expressions we'll consider this distinction won't make a big difference, but it's worth being aware of. Throughout we'll use TM to denote the *tangent bundle* of M, *i.e.*, the set of all tangent vectors.

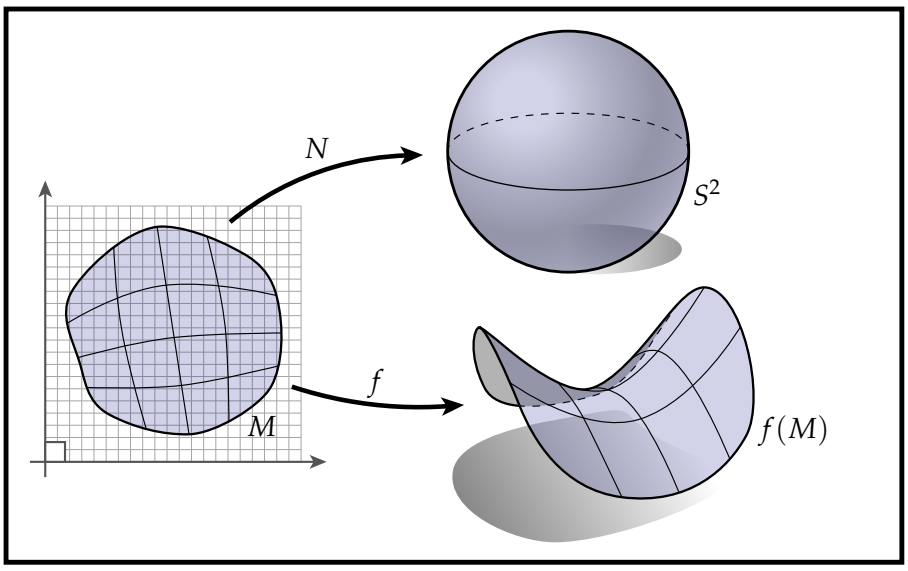
So far we've been talking about *tangent* vectors, i.e., vectors that lay flat along the surface. We're also interested in vectors that are orthogonal to the surface. In particular, we say that a vector $u \in \mathbb{R}^3$ is *normal* to the surface at a point p if

$$df(X) \cdot u = 0$$

for all tangent vectors X at p. For convenience, we often single out a particular normal vector N called the *unit normal*, which has length one. Of course, at any given point there are two distinct unit normal vectors: $+N$ and $-N$. Which one should we use? If we can pick a consistent direction for N then we say that M is *orientable*. For instance, the circular band on the left is orientable, but the *Möbius band* on the right is not:



For orientable surfaces, we can actually think of N as a continuous map $N : M \to S^2$ (called the *Gauss map*) which associates each point with its unit normal, viewed as a point on the unit sphere S^2. In fact, if we think of S^2 as a subset of $\mathbb{R}^3$ (consisting of all the points unit distance from the origin), then we can do all the same things with N that we did with our map f. In particular, the differential dN (called the *Weingarten map*) tells us about the change in the normal direction as we move from one point to the other. For instance, we can look at the change in normal along a particular tangent direction X by evaluating $dN(X)$—this interpretation will become useful when we talk about the curvature of surfaces. Overall we end up with the following picture, which captures the most fundamental ideas about the geometry of surfaces:

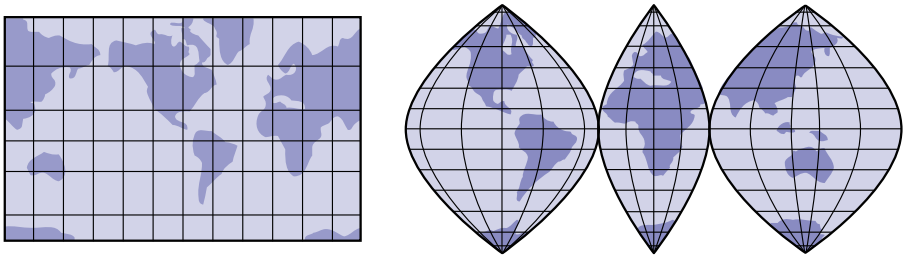


3.1.1. Conformal Coordinates. When working with curves, one often introduces the idea of an *isometric* (a.k.a. *arc-length* or *unit speed*) parameterization. The idea there is to make certain expressions simpler by assuming that no "stretching" occurs as we go from the domain into $\mathbb{R}^3$. One way to state this requirement is

$$|df(X)| = |X|,$$

i.e., we ask that the norm of any vector X is preserved.

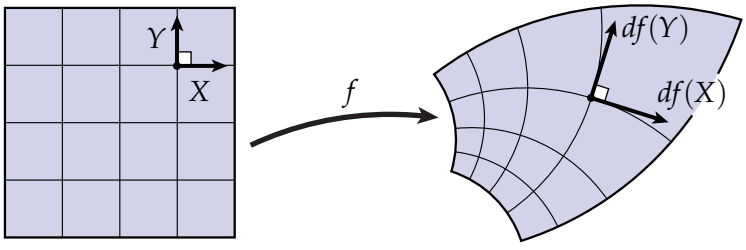
For surfaces, an isometric parameterization does not always exist (not even locally!). Most of the time you simply have to stretch things out. For instance, you may know that it's impossible to flatten the surface of the Earth onto the plane without distortion—that's why we end up with all sorts of different funky projections of the globe.



However, there is a setup that (like arc-length parameterization for curves) makes life a lot easier when dealing with certain expressions, namely *conformal coordinates*. Put quite simply, a map f is conformal if it preserves the *angle* between any two vectors. More specifically, a conformal map $f : \mathbb{R}^2 \supset M \to \mathbb{R}^3$ satisfies

$$df(X) \cdot df(Y) = a\langle X, Y\rangle$$

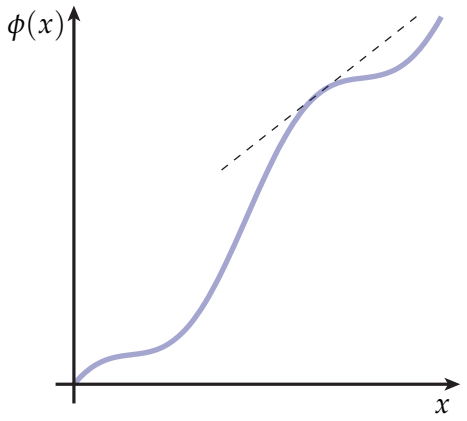
for all tangent vectors X, Y, where a is a **positive** function and $\langle \cdot, \cdot \rangle$ is the usual inner product on $\mathbb{R}^2$. In practice, the function a is often replaced with e^u for some real-valued function u—this way, one never has to worry about whether the scaling is positive. Notice that vectors can still get *stretched out*, but the surface never gets *sheared*—for instance, orthogonal vectors always stay orthogonal:



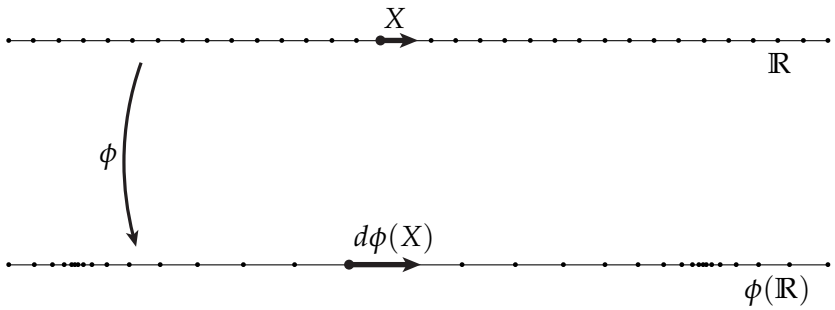
A key fact about conformal maps is that they always exist, as guaranteed by the uniformization theorem. In a nutshell, the uniformization theorem says that any disk can be conformally mapped to the plane. So if we consider any point p on our surface $f(M)$, we know that we can always find a conformal parameterization in some small, disk-like neighborhood around p. As with unit-speed curves, it is often enough to simply know that a conformal parameterization *exists*—we do not have to construct the map explicitly. And, as with arc-length parameterization, we have to keep track of the least possible amount of information about how the domain gets stretched out: just a single number at each point (as opposed to, say, an entire Jacobian matrix).

3.2. Derivatives and Tangent Vectors

3.2.1. Derivatives on the Real Line. So far we've been thinking about the differential in a very geometric way: it tells us how to stretch out or *push forward* tangent vectors as we go from one place to another. In fact, we can apply this geometric viewpoint to pretty much any situation involving derivatives. For instance, think about a good old fashioned real-valued function $\phi(x)$ on the real line. We typically visualize ϕ by plotting its value as a height over the x-axis:



In this case, the derivative ϕ' can be interpreted as the slope of the height function, as suggested by the dashed line in the picture above. Alternatively, we can imagine that ϕ stretches out the real line itself, indicated by the change in node spacing in this picture:



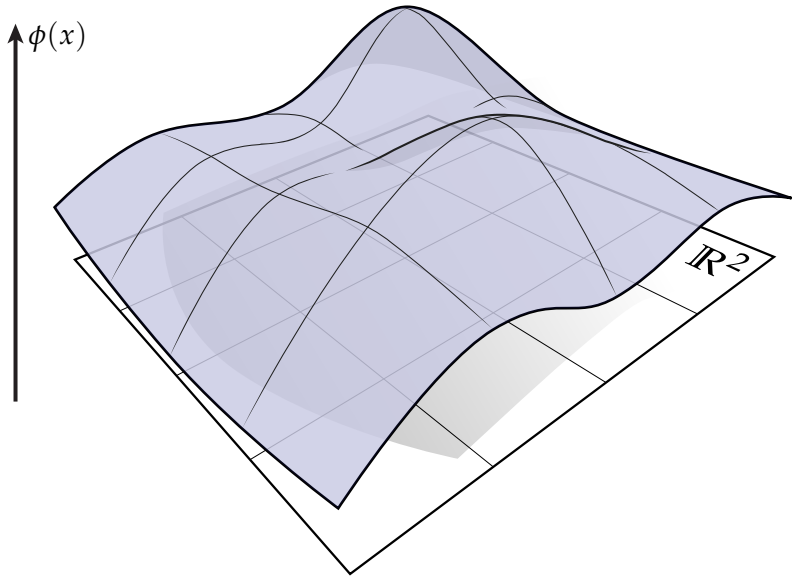
Where the derivative is large, nodes are spaced far apart; where the derivative is small, nodes are spaced close together. This picture inspires us to write the derivative of ϕ in terms of the push-forward $d\phi(X)$ of a unit tangent vector X pointing along the positive x-axis:

$$\phi' = d\phi(X).$$

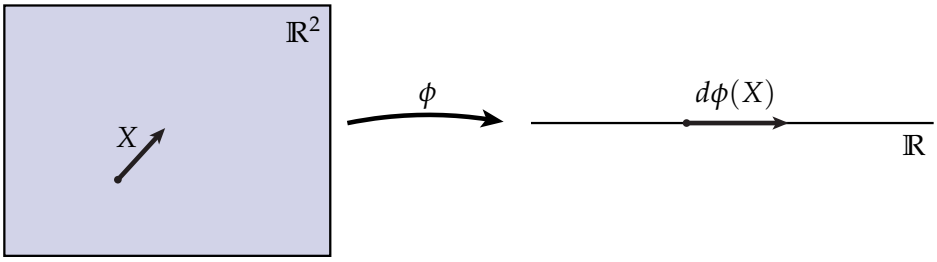
In other words, the derivative of ϕ is just the "stretch factor" as we go from one copy of $\mathbb{R}$ to the other. But wait a minute—does this equality even make sense? The thing on the left is a scalar, but the thing on the right is a vector! Of course, any tangent vector on the real line can be represented as just a single value, quantifying its extent in the positive or negative direction. So this expression does make sense—as long as we understand that we're identifying tangent vectors on $\mathbb{R}$ with real numbers. Often this kind of "type checking" can help verify that formulas and expressions are correct, similar to the way you might check for matching units in a physical equation.

Here's another question: how is this interpretation of the derivative any different from our usual interpretation in terms of height functions? Aren't we also stretching out the real line in that case? Well, yes and no—certainly the real line still gets stretched out into some other curve. But this curve is now a subset of the plane $\mathbb{R}^2$—in particular, it's the curve $\gamma = (x, \phi(x))$. So for one thing, "type checking" fails in this case: ϕ' is a scalar, but $d\gamma(X)$ is a 2-vector. But most importantly, the *amount* of stretching experienced by the curve doesn't correspond to our usual notion of the derivative of ϕ—for instance, if we look at the magnitude of $|d\gamma(X)|$ we get $\sqrt{1+(\phi')^2}$. (Why is this statement true geometrically? How *could* you write ϕ' in terms of $d\gamma(X)$? Can you come up with an expression that recovers the proper sign?)

3.2.2. Directional Derivatives. So far so good: we can think of the derivative of a real-valued function on $\mathbb{R}$ as the pushforward of a (positively-oriented) unit tangent vector X. But what does $d\phi(X)$ mean if ϕ is defined over some other domain, like the plane $\mathbb{R}^2$? This question may "stretch" your mind a little, but if you can understand this example then you're well on your way to understanding derivatives in terms of tangent vectors. Let's take a look at the geometry of the problem—again, there are two ways we could plot ϕ. The usual approach is to draw a height function over the plane:



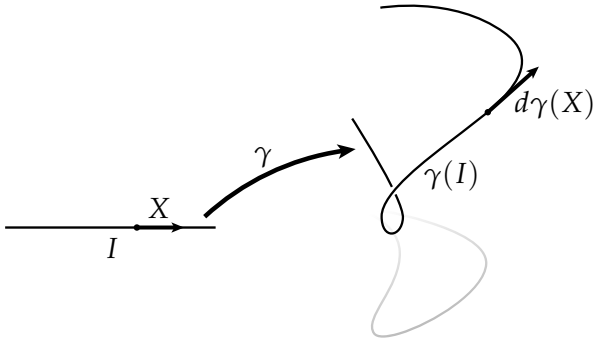
The derivative has something to do with the slope of this hill, but in which direction? To answer this question, we can introduce the idea of a *directional derivative*—i.e., we pick a vector X and see how quickly we travel uphill (or downhill) in that direction. And again we can consider an alternative picture:



Since ϕ is a map from $\mathbb{R}^2$ to $\mathbb{R}$, we can imagine that it takes a flat sheet of rubber and stretches it out into a long, skinny, one-dimensional object along the real line. Therefore if we draw an arrow X on the original sheet, then the "stretched-out" arrow $d\phi(X)$ gives us the rate of change in ϕ along the direction X, i.e., the directional derivative. What about type checking? As before, everything matches up: $d\phi(X)$ is a tangent vector on $\mathbb{R}$, so it can be represented by a single real number. (What if we had continued to work with the height function above? How could we recover the directional derivative in this case?)

By the way, don't worry if this discussion seems horribly informal! We'll see a more explicit, algebraic treatment of these ideas when we start talking about exterior calculus. The important thing for now is to build some geometric intuition about derivatives. In particular: a map from any space to any other space can be viewed as some kind of bending and twisting and stretching (or possibly tearing!); derivatives can be understood in terms of what happens to little arrows along the way.

3.3. The Geometry of Curves



The picture we looked at for surfaces is actually a nice way of thinking about shapes of any dimension. For instance, we can think of a one-dimensional curve as a map $\gamma : I \to \mathbb{R}^3$ from an interval $I = [0, T] \subset \mathbb{R}$ of the real line to $\mathbb{R}^3$. Again the differential $d\gamma$ tells us how tangent vectors get stretched out by γ, and again the induced length of a tangent vector X is given by

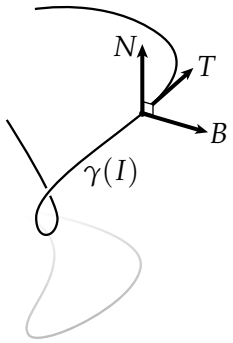
$$|d\gamma(X)| = \sqrt{d\gamma(X) \cdot d\gamma(X)}.$$

Working with curves is often easier if γ preserves *length*, i.e., if for every tangent vector X we have

$$|d\gamma(X)| = |X|.$$

There are various names for such a parameterization (*"unit speed"*, *"arc-length"*, *"isometric"*) but the idea is simply that the curve doesn't get stretched out when we go from $\mathbb{R}$ to $\mathbb{R}^3$—think of γ as a completely relaxed rubber band. This unit-speed view is also often the right one for the discrete setting where we have no notion of a base domain I—from the very beginning, the curve is given to us as a subset of $\mathbb{R}^3$ and all we can do is assume that it sits there in a relaxed state.

3.3.1. The Curvature of a Curve.



Suppose we have a unit-speed curve γ and a positively-oriented unit vector X on the interval I. Then

$$T = d\gamma(X)$$

is a unit vector in $\mathbb{R}^3$ tangent to the curve. Carrying this idea one step further, we can look at the change in tangent direction as we move along γ. Since T may change at any rate (or not at all!) we

split up the change into two pieces: a unit vector N called the *principal normal* that expresses the direction of change, and a scalar $\kappa \in \mathbb{R}$ called the *curvature* that expresses the magnitude of change:

$$dT(X) = -\kappa N.$$

One thing to realize is that T and N are always orthogonal. Why? Because if the change in T were *parallel* to T, then it would cease to have unit length! (This argument is a good one to keep in mind any time you work with unit vector fields.) By convention, we choose N to be the normal pointing to the "left" of the curve, *i.e.*, if at any point we consider a plane spanned by the tangent and the normal, N is a quarter turn in the counter-clockwise direction from T. Together with a third vector $B = T \times N$ called the *binormal*, we end up with a very natural orthonormal coordinate frame called the *Frenet frame*.

How does this frame change as we move along the curve? The answer is given by the *Frenet-Serret formula*:

$$\underbrace{\begin{bmatrix} T' \\ N' \\ B' \end{bmatrix}}_{Q' \in \mathbb{R}^{3\times 3}} = \underbrace{\begin{bmatrix} 0 & -\kappa & 0 \\ \kappa & 0 & -\tau \\ 0 & \tau & 0 \end{bmatrix}}_{A \in \mathbb{R}^{3\times 3}} \underbrace{\begin{bmatrix} T \\ N \\ B \end{bmatrix}}_{Q \in \mathbb{R}^{3\times 3}}.$$

Here T, N, and B are interpreted as row vectors, and a *prime* indicates the change in a quantity as we move along the curve at unit speed. For instance, $T' = dT(X)$, where X is a positively-oriented unit vector on I. The quantity τ is called the *torsion*, and describes the way the normal and binormal twist around the curve.

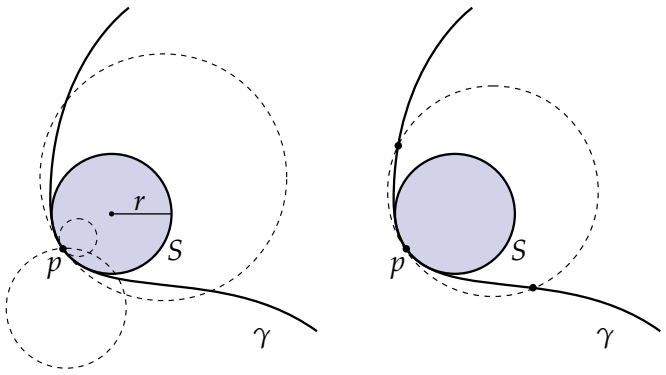
A concise proof of this formula was given by Cartan. First, since the vectors T, N, and B are mutually orthogonal, one can easily verify that $QQ^T = I$, i.e., Q is an orthogonal matrix. Differentiating this relationship in time, the identity vanishes and we're left with $Q'Q^T = -(Q'Q^T)^T$, i.e., the matrix $Q'Q^T$ is *skew-symmetric*. But since $A = Q'Q^T$, A must also be skew-symmetric. Skew symmetry implies that the diagonal of A is zero (why?) and moreover, we already know what the top row (and hence the left column) looks like from our definition of κ and N. The remaining value $A_{23} = -A_{32}$ is not constrained in any way, so we simply give it a name: $\tau \in \mathbb{R}$.

What do you think about this proof? On the one hand it's easy to verify; on the other hand, it provides little geometric understanding. For instance, why does N change in the direction of both T and B, but B changes only in the direction of N? Can you come up with more geometric arguments?

3.3.2. Visualizing Curvature. What's the curvature of a circle S? Well, if S has radius r then it takes time $2\pi r$ to go all the way around the circle at unit speed. During this time the tangent turns around by an angle 2π. Of course, since T has unit length the instantaneous change in T is described exclusively by the instantaneous change in angle. So we end up with

$$\kappa = |\kappa N| = |dT(X)| = 2\pi/2\pi r = 1/r.$$

In other words, the curvature of a circle is simply the reciprocal of its radius. This fact should make some intuitive sense: if we watch a circle grow bigger and bigger, it eventually looks just like a straight line with zero curvature: $\lim_{r\to\infty} 1/r = 0$. Similarly, if we watch a circle get smaller and smaller it eventually looks like a single point with infinite curvature: $\lim_{r\to 0} 1/r = \infty$.



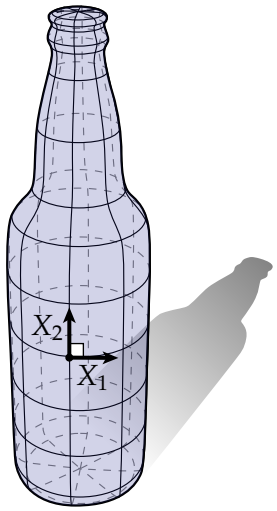
Now consider a smooth curve γ in the plane. At any point $p \in \gamma$ there is a circle S called the *osculating circle* that best approximates γ, meaning that it has the same tangent direction T and curvature vector κN. In other words, the circle and the curve agree "up to second order." (The phrase "agree up to nth order" is just shorthand for saying that the first n derivatives are equal.) How do we know such a circle exists? Easy: we can always construct a circle with the appropriate curvature by setting $r = 1/\kappa$; moreover *every* circle has some tangent pointing in the direction T. Alternatively, we can consider a circle passing through p and two other points: one approaching from the left, another approaching from the right. Since these three points are shared by both γ and S, the first and second derivatives will agree in the limit (consider that these points can be used to obtain consistent finite difference approximations of T and κN).

The radius and center of the osculating circle are often referred to as the *radius of curvature* and *center of curvature*, respectively. We can tell this same story for any curve in $\mathbb{R}^3$ by considering the *osculating plane* $T \times N$, since this plane contains both the tangent and the curvature vector.

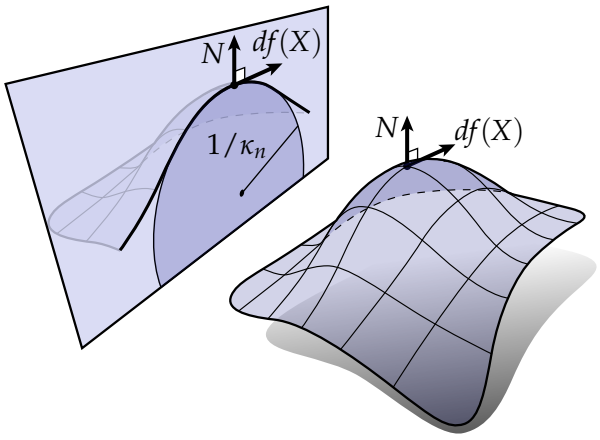
For curves it makes little difference whether we express curvature in terms of a change in the tangent vector or a change in the (principal) normal, since the two vectors are the same up to a quarter-rotation in the osculating plane. For surfaces, however, it will often make more sense to think of curvature as the change in the normal vector, since we typically don't have a distinguished tangent vector to work with.

3.4. Curvature of Surfaces

Let's take a more in-depth look at the curvature of surfaces. The word "curvature" really corresponds to our everyday understanding of what it means for something to be curved: eggshells, donuts, and cavatappi pasta have a lot of curvature; floors, ceilings, and cardboard boxes do not. But what about something like a beer bottle? Along one direction the bottle quickly curves around in a circle; along another direction it's completely flat and travels along a straight line:



This way of looking at curvature—in terms of curves contained in the surface—is often how we treat curvature in general. In particular, let $df(X)$ be a unit tangent direction at some distinguished point on the surface, and consider a plane containing both $df(X)$ and the corresponding normal N. This plane intersects the surface in a curve, and the curvature κ_n of this curve is called the *normal curvature* in the direction X:

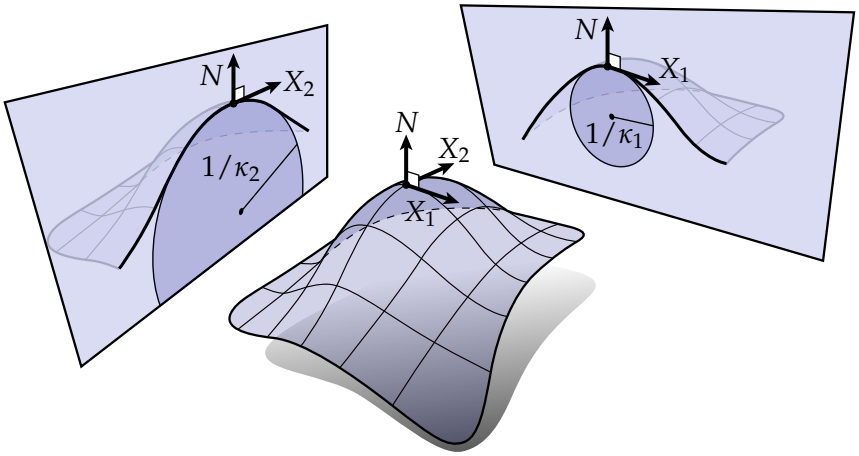


Remember the Frenet-Serret formulas? They tell us that the change in the normal along a *curve* is given by $dN = \kappa T - \tau B$. We can therefore get the normal curvature along X by extracting the tangential part of dN:

$$\kappa_n(X) = \frac{df(X) \cdot dN(X)}{|df(X)|^2}.$$

The factor $|df(X)|^2$ in the denominator simply normalizes any "stretching out" that occurs as we go from the domain M into $\mathbb{R}^3$. Note that normal curvature is *signed*, meaning the surface can bend toward the normal or away from it.

3.4.1. Principal, Mean, and Gaussian Curvature.



At any given point we can ask: along which directions does the surface bend the most? The unit vectors X_1 and X_2 along which we find the maximum and minimum normal curvatures κ_1 and κ_2 are called the *principal directions*; the curvatures κ_i are called the *principal curvatures*. For instance, the beer bottle above might have principal curvatures $\kappa_1 = 1$, $\kappa_2 = 0$ at the marked point.

We can also talk about principal curvature in terms of the *shape operator*, which is the unique map $S : TM \to TM$ satisfying

$$df(SX) = dN(X)$$

for all tangent vectors X. The shape operator S and the Weingarten map dN essentially represent the same idea: they both tell us how the normal changes as we travel along a direction X. The only difference is that S specifies this change in terms of a tangent vector on M, whereas dN gives us the change as a tangent vector in $\mathbb{R}^3$. It's worth noting that many authors do not make this distinction, and simply assume an isometric identification of tangent vectors on M and the corresponding tangent vectors in $\mathbb{R}^3$. However, we choose to be more careful so that we can explicitly account for the dependence of various quantities on the immersion f—this dependence becomes particularly important if you actually want to compute something! (By the way, why can we always express the change in N in terms of a *tangent* vector? It's because N is the *unit* normal, hence it cannot grow or shrink in the normal direction.)

One important fact about the principal directions and principal curvatures is that they correspond to eigenvectors and eigenvalues (respectively) of the shape operator:

$$SX_i = \kappa_i X_i.$$

Moreover, the principal directions are orthogonal with respect to the induced metric: $g(X_1, X_2) = df(X_1) \cdot df(X_2) = 0$. The principal curvatures therefore tell us everything there is to know about normal curvature at a point, since we can express any tangent vector Y as a linear combination of the principal directions X_1 and X_2. In particular, if Y is a unit vector offset from X_1 by an angle θ,

then the associated normal curvature is

$$\kappa_n(Y) = \kappa_1 \cos^2\theta + \kappa_2 \sin^2\theta,$$

as you should be able to easily verify using the relationships above. Often, however, working directly with principal curvatures is fairly inconvenient—especially in the discrete setting.

On the other hand, two closely related quantities—called the *mean curvature* and the *Gaussian curvature* will show up over and over again (and have some particularly nice interpretations in the discrete world). The mean curvature H is the arithmetic mean of principal curvatures:

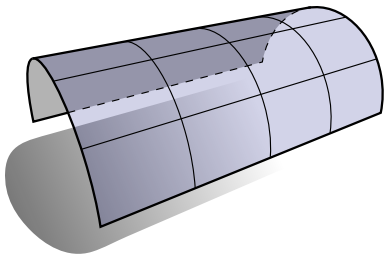
$$H = \frac{\kappa_1 + \kappa_2}{2},$$

and the Gaussian curvature is the (square of the) geometric mean:

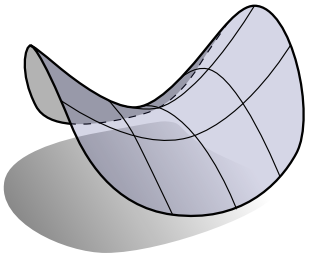
$$K = \kappa_1 \kappa_2.$$

What do the values of H and K imply about the shape of the surface? Perhaps the most elementary interpretation is that Gaussian curvature is like a logical "and" (is there curvature along *both* directions?) whereas mean curvature is more like a logical "or" (is there curvature along *at least one* direction?) Of course, you have to be a little careful here since you can also get zero mean curvature when $\kappa_1 = -\kappa_2$.

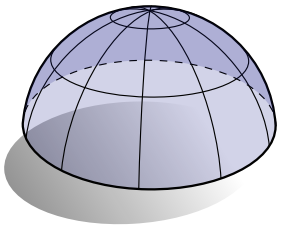
It also helps to see pictures of surfaces with zero mean and Gaussian curvature. Zero-curvature surfaces are so well-studied in mathematics that they have special names. Surfaces with zero Gaussian curvature are called *developable surfaces* because they can be "developed" or flattened out into the plane without any stretching or tearing. For instance, any piece of a cylinder is developable since one of the principal curvatures is zero:



Surfaces with zero mean curvature are called *minimal surfaces* because (as we'll see later) they minimize surface area (with respect to certain constraints). Minimal surfaces tend to be saddle-like since principal curvatures have equal magnitude but opposite sign:



The saddle is also a good example of a surface with negative *Gaussian* curvature. What does a surface with positive Gaussian curvature look like? The hemisphere is one example:



Note that in this case $\kappa_1 = \kappa_2$ and so principal directions are not uniquely defined—maximum (and minimum) curvature is achieved along *any* direction X. Any such point on a surface is called an *umbilic point*.

There are plenty of cute theorems and relationships involving curvature, but those are the basic facts: the curvature of a surface is completely characterized by the *principal curvatures*, which are the maximum and minimum *normal curvatures*. The Gaussian and mean curvature are simply averages of the two principal curvatures, but (as we'll see) are often easier to get your hands on in practice.

3.4.2. The Fundamental Forms. For historical reasons, there are two objects we should probably mention: *first fundamental form* I and the *second fundamental form* II. I'm actually not sure what's so fundamental about these forms, since they're nothing more than a mashup of the metric g and the shape operator S, which themselves are simple functions of two *truly* fundamental objects: the immersion f and the Gauss map N. In fact, the first fundamental form is literally just the induced metric, *i.e.*,

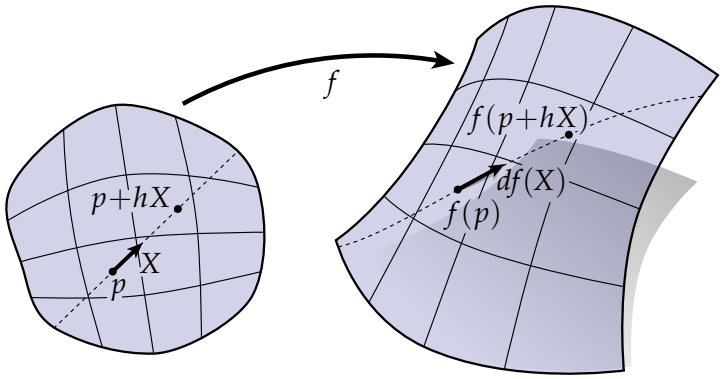
$$I(X, Y) := g(X, Y).$$

The second fundamental form looks quite similar to our existing expression for normal curvature:

$$II(X, Y) := -g(SX, Y) = -dN(X) \cdot df(Y).$$

The most important thing to realize is that I and II do not introduce any new *geometric* ideas—just another way of writing down things we've already seen.

3.5. Geometry in Coordinates



So far we've given fairly abstract descriptions of the geometric objects we've been working with. For instance, we said that the differential df of an immersion $f : M \to \mathbb{R}^3$ tells us how to stretch out tangent vectors as we go from the domain $M \subset \mathbb{R}^2$ into the image $f(M) \subset \mathbb{R}^3$. Alluding to the picture above, we can be a bit more precise and define $df(X)$ in terms of limits:

$$df_p(X) = \lim_{h \to 0} \frac{f(p + hX) - f(p)}{h}.$$

Still, this formula remains a bit abstract—we may want something more concrete to work with in practice. When we start working with discrete surfaces we'll see that $df(X)$ often has an incredibly concrete meaning—for instance, it might correspond to an edge in our mesh. But in the smooth setting a more typical representation of df is the *Jacobian matrix*

$$\mathsf{J} = \begin{bmatrix} \partial f^1/\partial x^1 & \partial f^1/\partial x^2 \\ \partial f^2/\partial x^1 & \partial f^2/\partial x^2 \\ \partial f^3/\partial x^1 & \partial f^3/\partial x^2 \end{bmatrix}.$$

Here we pick coordinates on $\mathbb{R}^2$ and $\mathbb{R}^3$, and imagine that

$$f(x^1, x^2) = (f_1(x^1, x^2), f_2(x^1, x^2), f_3(x^1, x^2))$$

for some triple of scalar functions $f_1, f_2, f_3 : M \to \mathbb{R}$. So if you wanted to evaluate $df(X)$, you could simply apply J to some vector $X = [X^1 \;\; X^2]^T$.

3.5.1. Coordinate Representations Considered Harmful. You can already see one drawback of the approach taken above: expressions get a lot longer and more complicated to write out. But there are other good reasons to avoid explicit matrix representations. The most profound reason is that matrices can be used to represent many different types of objects, and these objects can behave in very different ways. For instance, can you guess what the following matrix represents?

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Give up? It's quite clear, actually: it's the adjacency matrix for the complete graph on two vertices. No, wait a minute—it must be the Pauli matrix σ_x, representing spin angular momentum along the x-axis. Or is it the matrix representation for an element of the dihedral group D_4? You get the idea: when working with matrices, it's easy to forget where they come from—which makes it very easy

to forget which rules they should obey! (Don't you already have enough things to keep track of?) The real philosophical point here is that *matrices are not objects: they are merely* ***representations*** *of objects!* Or to paraphrase Plato: matrices are merely shadows on the wall of the cave, which give us nothing more than a murky impression of the real objects we wish to illuminate.

A more concrete example that often shows up in geometry is the distinction between linear operators and bilinear forms. As a reminder, a *linear operator* is a map from one vector space to another, e.g.,

$$f : \mathbb{R}^2 \to \mathbb{R}^2; u \mapsto f(u),$$

whereas a *bilinear form* is a map from a pair of vectors to a scalar, e.g.,

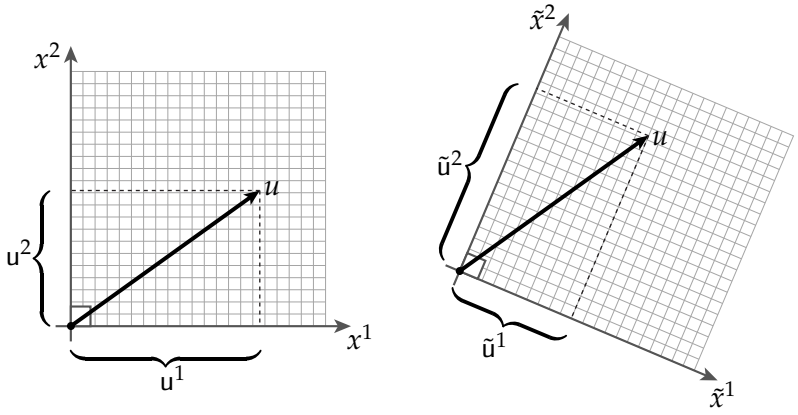
$$g : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}; (u, v) \mapsto g(u, v).$$

Sticking with these two examples let's imagine that we're working in a coordinate system (x^1, x^2), where f and g are represented by matrices $\mathsf{A}, \mathsf{B} \in \mathbb{R}^{2\times 2}$ and their arguments are represented by vectors $\mathsf{u}, \mathsf{v} \in \mathbb{R}^2$. In other words, we have

$$f(u) = \mathsf{Au}$$

and

$$g(u, v) = \mathsf{u}^\mathsf{T}\mathsf{Bv}.$$



Now suppose we need to work in a different coordinate system $(\tilde{x}^1, \tilde{x}^2)$, related to the first one by a change of basis $\mathsf{P} \in \mathbb{R}^{2\times 2}$. For instance, the vectors u and v get transformed via

$$\tilde{\mathsf{u}} = \mathsf{Pu},$$

$$\tilde{\mathsf{v}} = \mathsf{Pv}.$$

How do we represent the maps f and g in this new coordinate system? We can't simply evaluate $\mathsf{A}\tilde{\mathsf{u}}$, for instance, since A and $\tilde{\mathsf{u}}$ are expressed in different bases. What we need to do is evaluate

$$f(u) = \mathsf{PAu} = \mathsf{PAP}^{-1}\tilde{\mathsf{u}}$$

and similarly

$$g(u, v) = \mathsf{u}^T\mathsf{Bv} = (\mathsf{P}^{-1}\tilde{\mathsf{u}})^T\mathsf{B}(\mathsf{P}^{-1}\tilde{\mathsf{v}}) = \tilde{\mathsf{u}}^T(\mathsf{P}^{-\mathsf{T}}\mathsf{BP}^{-1})\tilde{\mathsf{v}}.$$

In other words, linear operators transform like

$$\mathsf{A} \mapsto \mathsf{P}\mathsf{A}\mathsf{P}^{-1},$$

whereas bilinear forms transform like

$$\mathsf{B} \mapsto \mathsf{P}^{-\mathsf{T}}\mathsf{B}\mathsf{P}^{-1}.$$

So what we discover is that *not all matrices transform the same way!* But if we're constantly scrawling out little grids of numbers, it's very easy to lose track of which transformations should be applied to which objects.

3.5.2. Standard Matrices in the Geometry of Surfaces. Admonitions about coordinates aside, it's useful to be aware of standard matrix representations for geometric objects because they provide an essential link to classical results. We've already seen a matrix representation for one object: the differential df can be encoded as the Jacobian matrix J containing first-order derivatives of the immersion f. What about the other objects we've encountered in our study of surfaces? Well, the induced metric g should be pretty easy to figure out since it's just a function of the differential—remember that

$$g(u, v) = df(u) \cdot df(v).$$

Equivalently, if we use a matrix $\mathsf{I} \in \mathbb{R}^{2\times 2}$ to represent g, then we have

$$\mathsf{u}^{\mathsf{T}}\mathsf{I}\mathsf{v} = (\mathsf{J}\mathsf{u})^{\mathsf{T}}(\mathsf{J}\mathsf{v})$$

which means that

$$\mathsf{I} = \mathsf{J}^{\mathsf{T}}\mathsf{J}.$$

We use the letter "I" to denote the matrix of the induced metric, which was historically referred to as the *first fundamental form* —fewer authors use this terminology today. In older books on differential geometry you may also see people talking about "E", "F", and "G", which refer to particular entries of I:

$$\mathsf{I} = \begin{bmatrix} \mathsf{E} & \mathsf{F} \\ \mathsf{F} & \mathsf{G} \end{bmatrix}.$$

(Is it clear why "F" appears twice?) One might conjecture that these fifth, sixth, and seventh letters of the alphabet have fallen out of fashion precisely because they are so coordinate-dependent and hence carry little geometric meaning on their own. Nonetheless, it is useful to be able to recognize these critters, because they do show up out there in the wild.

Earlier on, we also looked at the *shape operator,* defined as the unique map $S : TM \to TM$ satisfying

$$dN(X) = df(SX),$$

and the *second fundamental form,* defined as

$$I\!I(u, v) = g(Su, v).$$

(Remember that S turned out to be self-adjoint with respect to g, and likewise $I\!I$ turned out to be symmetric with respect to its arguments u and v.) If we let $\mathsf{S}, \mathbb{I} \in \mathbb{R}^{2\times 2}$ be the matrix representations of S and $I\!I$, respectively, then we have

$$\mathsf{u}^{\mathsf{T}}\mathbb{I}\mathsf{v} = \mathsf{u}^{\mathsf{T}}\mathsf{I}\mathsf{S}\mathsf{v}$$

for all vectors $\mathsf{u}, \mathsf{v} \in \mathbb{R}^2$, or equivalently,

$$\mathbb{I} = \mathsf{I}\mathsf{S}.$$

Components of $\mathbb{II}$ are classically associated with *lowercase* letters from the Roman alphabet, namely

$$\mathbb{II} = \begin{bmatrix} \mathsf{e} & \mathsf{f} \\ \mathsf{f} & g \end{bmatrix},$$

which in coordinates (x, y) are given explicitly by

$$\begin{aligned} \mathsf{e} &= N \cdot f_{xx}, \\ \mathsf{f} &= N \cdot f_{xy}, \\ \mathsf{g} &= N \cdot f_{yy}, \end{aligned}$$

where N is the unit surface normal and f_{xy} denotes the second partial derivative along directions x and y.

At this point we might want to stop and ask: how does a matrix like IS transform with respect to a change of basis? The first term, I, is a bilinear form, but the second term S is a linear map! As emphasized above, we can't determine the answer by just staring at the matrices themselves—we need to remember what they represent. In this case, we know that IS corresponds to the second fundamental form, so it should transform like any other bilinear form: $\mathsf{IS} \mapsto \mathsf{P}^{-\mathsf{T}} I S \mathsf{P}^{-1}$.

Finally, we can verify that classical geometric expressions using matrices correspond to the expressions we derived earlier using the differential. For instance, the classical expression for normal curvature is

$$\kappa_n(u) = \frac{I\!I(u,u)}{I(u,u)},$$

which we can rewrite as

$$\frac{\mathsf{u}^{\mathsf{T}}\mathbb{II}\mathsf{u}}{\mathsf{u}^{\mathsf{T}}\mathsf{I}\mathsf{u}} = \frac{\mathsf{u}^{\mathsf{T}}\mathsf{I}\mathsf{S}\mathsf{u}}{\mathsf{u}^{\mathsf{T}}\mathsf{I}\mathsf{u}} = \frac{(\mathsf{J}\mathsf{u})^{\mathsf{T}}(\mathsf{J}\mathsf{S}\mathsf{u})}{(\mathsf{J}\mathsf{u})^{\mathsf{T}}(\mathsf{J}\mathsf{u})} = \frac{df(u) \cdot dN(u)}{|df(u)|^2}.$$

Up to a choice of sign, this expression is the same one we obtained earlier by considering a curve embedded in the surface.

CHAPTER 4

A Quick and Dirty Introduction to Exterior Calculus

Many important concepts in differential geometry can be nicely expressed in the language of *exterior calculus*. Initially these concepts will look exactly like objects you know and love from *vector calculus*, and you may question the value of giving them funky new names. For instance, scalar fields are no longer called scalar fields, but are now called *0-forms!* In the long run we'll see that this new language makes it easy to generalize certain ideas from vector calculus—a central example being *Stokes' theorem*, which in turn is intimately related to discretization, and ultimately, computation.

The basic story of exterior calculus can be broken up into a few pieces:

- **Linear Algebra: Little Arrows.** If you've ever studied linear algebra, you probably remember that it has something to do with "little arrows"—also known as *vectors*. In fact, if that's all you can remember about linear algebra, *now would be an extremely good time to go back and do a review!* We're not going to do one here.
- **Vector Calculus: How do Little Arrows Change?** Likewise, if you've ever studied vector calculus, then you remember it has to do with how "little arrows" change over space and time (*e.g.*, how fast the direction of the wind is changing). In other words, vector calculus tells us how to *differentiate* vectors. We're not going to review that either!
- **Exterior Algebra: Little Volumes.** Linear algebra explored a bunch of things you can do with vectors: you can add them, you can scale them, you can take inner products, and outer products, and so forth. *Exterior algebra* just adds a couple more operations to this list which make it easy to talk about things like area and volume. In particular, the operations let us build up things called *k-vectors*, which can be thought of as "little k-dimensional volumes."
- **Exterior Calculus: How do Little Volumes Change?** Finally, if *vector calculus* is the study of how "little arrows" change over space and time, then *exterior calculus* is the study of how "little volumes" change over space and time. In other words, exterior calculus tells us how to differentiate k-vectors.

That's the big picture: exterior calculus is to exterior algebra what vector calculus is to linear algebra. And little volumes are useful because they help us talk about integration in a very general context. If that story still sounds a bit fuzzy, then read on!

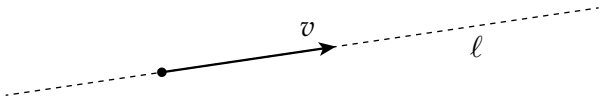
4.1. Exterior Algebra

As alluded to above, just as linear algebra is the natural language of "little arrows," *exterior algebra* is the natural language of "little volumes" which we will call *k-vectors*. The letter "k" denotes the dimension, for instance, a 1-vector represents a "little length," a 2-vector represents a "little area," and so on. A fundamental thing to remember about ordinary vectors is that they encode two basic pieces of information: *direction*, and *magnitude*. Likewise, k-vectors will also have a direction and a magnitude, though the notion of "direction" for k-dimensional volumes is a little bit trickier than for one-dimensional vectors. In its full generality, exterior algebra makes sense in any vector space V, but to keep things simple for now we'll just stick to familiar examples like the plane $\mathbb{R}^2$, three-dimensional space $\mathbb{R}^3$, or more generally, n-dimensional space $\mathbb{R}^n$.

4.1.1. Warm Up: 1-Vectors and 2-Vectors. How do you describe a volume in $\mathbb{R}^n$? The basic idea of exterior algebra is that, roughly speaking, k-dimensional volumes can be described by a list of k vectors. In linear algebra we had a somewhat similar idea: k vectors can be used to describe a k-dimensional linear subspace via the *span* (one vector spans a line; two vectors span a plane, and so forth). In either case the particular choice of vectors is not so important: for instance, just as many different pairs of vectors can span the same plane, many different pairs of vectors can be used to describe the same 2-vector. Overall, the k-vectors that appear in exterior algebra are not so different from linear subspaces, except that

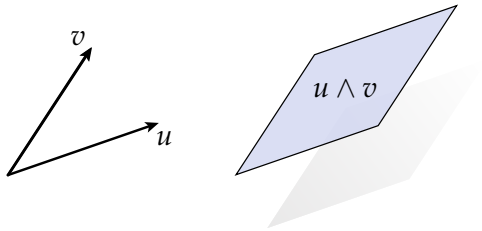
(1) they have "finite extent", *i.e.*, they have a *magnitude* and
(2) they have an *orientation*.

What do we mean by "orientation?" A good analogy is to think about the difference between a line ℓ and a vector v:

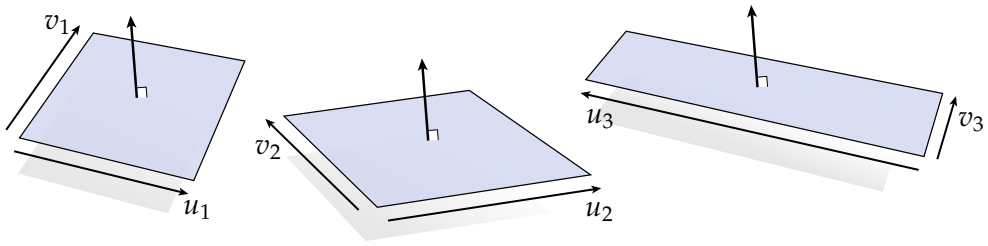


A line encodes a direction but with no sense of orientation, *i.e.*, no notion of which way along the line is "forward" or "backward." In contrast, a vector encodes a direction and a definite orientation (*e.g.*, $+v$ and $-v$ point in opposite directions); moreover, a vector has a definite magnitude, given by its length. The analogy between lines and vectors capture the basic idea behind k-vectors: a k-vector is to a k-dimensional linear subspace what a vector is to a line. In fact, ordinary vectors provide our first example of an object in exterior algebra: *a 1-vector is just an ordinary vector*.

What about 2-vectors? A pretty good visualization of a *2-vector* is to associate any two vectors u, v in three-dimensional space $\mathbb{R}^3$ with the volume spanned by a little parallelogram:



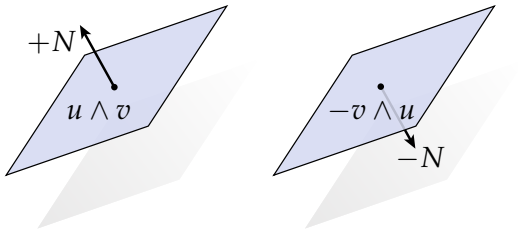
As a shorthand, we will denote this little parallelogram or 2-vector as $u \wedge v$ (here the $\wedge$ symbol is pronounced "wedge"). As with ordinary vectors, two 2-vectors are considered "the same" if they have the same magnitude and direction. For instance, all the parallelograms in the picture below have been carefully constructed to have identical area. All three therefore depict the same 2-vector, even though they are skewed and stretched by different amounts:



$$u_1 \wedge v_1 = u_2 \wedge v_2 = u_3 \wedge v_3$$

In this sense, our parallelogram drawings are merely "cartoons" of a 2-vector, since they each depict only one of many possibilities. However, since parallelograms faithfully represent many of the features of 2-forms, we can use them to investigate the way general 2-forms behave.

First and foremost, how do we define orientation for a 2-vector? For 1-vectors, this was an easy idea: the two (1-)vectors $+u$ and $-u$ have opposite orientation because they point in opposite directions along the same line. Likewise, we can think of a 2-vector in $\mathbb{R}^3$ as having two possible orientations: "up" or "down", corresponding to the two possible unit normals for the plane it sits in: $+N$ or $-N$. We will therefore distinguish between the two expressions $u \wedge v$ or $v \wedge u$, writing $u \wedge v = -v \wedge u$ to indicate that they have opposite orientation:

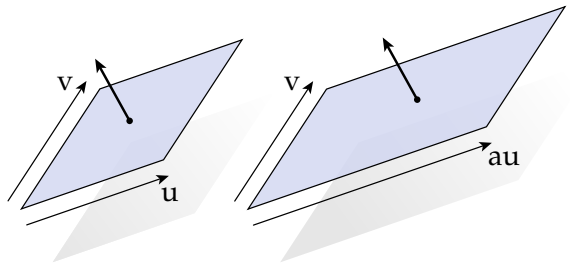


What behavior can we observe by playing around with little parallelograms? For one thing, it seems it must be the case that

$$u \wedge u = 0,$$

since the "parallelogram" described by two copies of the same vector has no area at all! This idea corresponds nicely with the idea that $u \wedge v = -v \wedge u$, since when $u = v$ we get $u \wedge u = -u \wedge u$.

Another thing we can notice is that scaling just one of the vectors by a factor $a \in \mathbb{R}$ will scale the area of our parallelogram by the same amount:

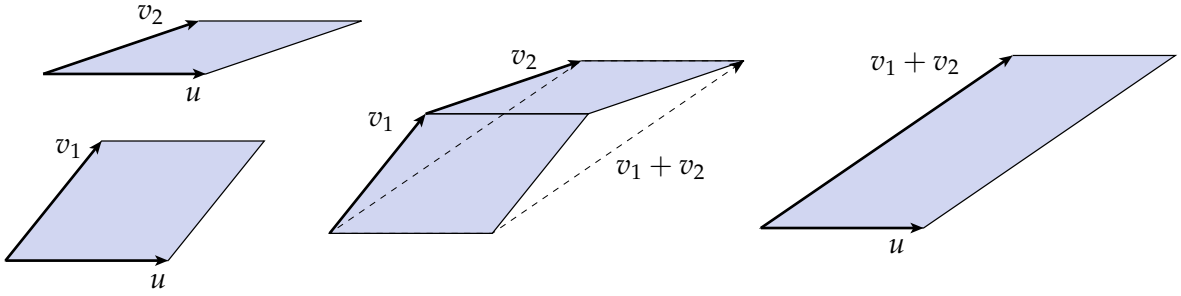


We might therefore encode this behavior via the rule

$$(au) \wedge v = a(u \wedge v).$$

Of course, the same kind of thing will happen if we scale the second vector rather than the first, *i.e.*, $u \wedge (av) = a(u \wedge v)$.

What can we say about the behavior of parallelograms when we *add* vectors? The following picture helps answer this question:

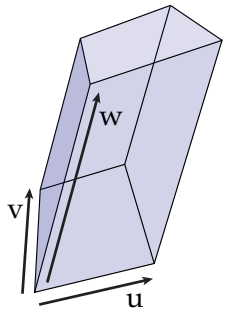


The sum of the two areas on the left can be expressed as $u \wedge v_1 + u \wedge v_2$; the area on the right is $u \wedge (v_1 + v_2)$. The middle image suggests that these two quantities are equal, since the area we lose is identical to the area we gain. In other words, it seems that

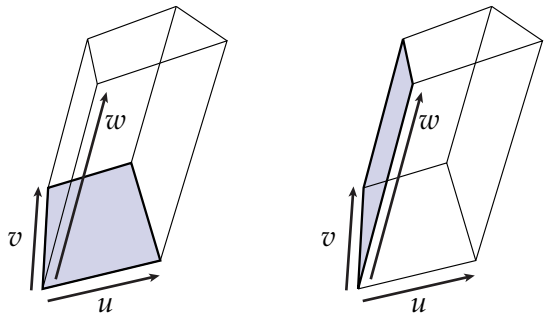
$$u \wedge v_1 + u \wedge v_2 = u \wedge (v_1 + v_2).$$

(Can you come up with a similar picture in 3D?)

To observe one final property, we must consider *volumes* rather than areas, which we will depict as little parallelepipeds:



Just as with 2-vectors, we can denote this little volume or *3-vector* as $u \wedge v \wedge w$. Moreover, we can think of this 3-vector as being constructed by first using two vectors to construct a little parallelogram, and then extruding this parallelogram along a third vector:



Notice that the order doesn't really seem to matter here: we can build the 2-vector $u \wedge v$ and then extend it along w, or we can first build $v \wedge w$ and then extend it along u. We can summarize this observation by saying that

$$(u \wedge v) \wedge w = u \wedge (v \wedge w),$$

which means that we can simply write $u \wedge v \wedge w$ without any ambiguity about which volume we mean. What would happen, however, if we flipped the order of the two vectors used to build the initial parallelogram? Earlier we said that $u \wedge v = -v \wedge u$, *i.e.*, swapping the order of vectors swaps the orientation of a 2-vector. Hence, we get $(u \wedge v) \wedge w = -(v \wedge u) \wedge w$, or just

$$u \wedge v \wedge w = -v \wedge u \wedge w.$$

Ok, but what does this statement mean geometrically? The minus sign seems to indicate that the two little volumes are identical *up to orientation*. But what does orientation mean for a volume? For vectors we had two orientations ($+u$ and $-u$) corresponding to "forward" and "backward"; for 2-vectors we had two orientations ($u \wedge v$ and $v \wedge u$) corresponding to "up" and "down" orientations of the plane. Likewise, we can imagine that a little volume has either an "inward" or "outward" orientation—for instance, you might imagine that the normal to the boundary points in and out, or that one side of the boundary is painted red and the other is painted blue. In either case there are just *two* orientations. By playing around a bit more we can notice that every time we swap a pair of consecutive vectors in a 3-vector the orientation switches; if we swap another pair the orientation switches back. Hence, *any even permutation of vectors preserves orientation; any odd permutation reverses orientation*. In other words, the three 3-vectors

$$u \wedge v \wedge w = v \wedge w \wedge u = w \wedge u \wedge v$$

all have the same orientation, and the three-vectors

$$w \wedge v \wedge u = v \wedge u \wedge w = u \wedge w \wedge v$$

all have the same orientation, but these two groups of three have opposite orientation.

4.1.2. The Wedge Product. Already we've established a bunch of rules about how little volumes appear to behave, which start to provide a definition for the *wedge product* $\wedge$. In particular, for any collection of vectors $u, v, w \in \mathbb{R}^n$ and scalars $a, b \in \mathbb{R}$ we have

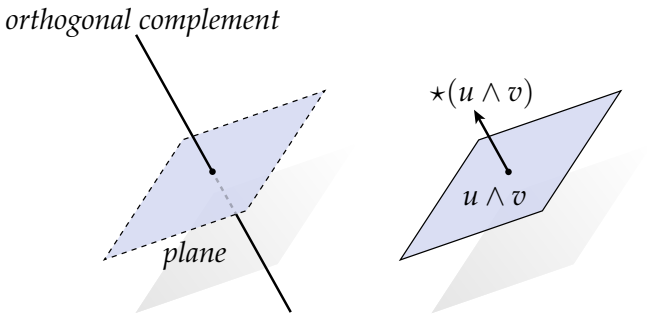
- **(Antisymmetry)** $u \wedge v = -v \wedge u$
- **(Associativity)** $(u \wedge v) \wedge w = u \wedge (v \wedge w)$
- **(Distributivity over addition)** $u \wedge (v + w) = u \wedge v + u \wedge w$
- **(Distributivity of scalar multiplication)** $(au) \wedge (bv) = ab(u \wedge v)$

In fact, these rules provide the right impression of how the wedge product behaves in any vector space, for any number of vectors. For now we'll hold off on a full-blown formal definition—the more important thing to remember is *where these rules came from*. In other words, how did the behavior of "little volumes" motivate us to write down this list in the first place? If you can get your head around the geometric picture, the rules should follow naturally. (And conversely, if you don't take a minute to think about the geometry behind the wedge product, you may be forever perplexed!)

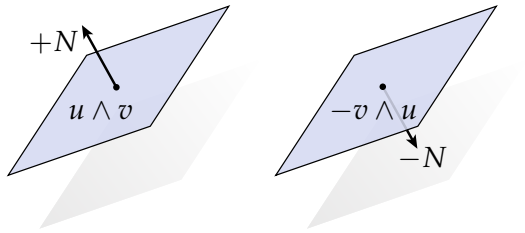
Working out some concrete examples (*e.g.*, in your homework) should also help to build up some intuition for k-vectors and the wedge product. A bit later on we'll revisit the wedge product in the context of a somewhat different vector space: rather than individual vectors in $\mathbb{R}^n$, we'll be thinking about whole *vector fields*, leading to the idea of *differential forms*.

4.1.3. The Hodge Star. Often, it's easiest to specify a set by instead specifying its complement. For instance, if you asked me, *"what foods do you like?"* it would be much easier to say, *"I like everything except for nattō* [1] *and doogh* [2]*"* rather than saying, *"I like pizza, and apples, and hamburgers, and sushi, and fesenjān, and chicken & waffles, and . . . "*. In linear algebra, a good example of this idea is the *orthogonal complement:* if I want to specify a k-dimensional linear subspace $W \subset V$ of an n-dimensional linear space V, I can provide either a collection of vectors $w_1, \ldots, w_n$ that span W, or alternatively, I can provide a collection of vectors $\tilde{w}_1, \ldots, \tilde{w}_k$ spanning the vectors that are *not* in W, *i.e.*, its *orthogonal complement*. For instance, a plane in $\mathbb{R}^3$ can be specified either by two vectors that span it, or a single vector giving its normal:

[1] Natto is a Japanese dish consisting of sticky, fermented soy beans.
[2] Doogh is a salty Persian yogurt drink.



In exterior algebra, the *Hodge star* $\star$ (pronounced "star") provides a sort of orthogonal complement for k-vectors. In particular, if we have a k-vector v in $\mathbb{R}^n$, then $\star v$ will be an $(n-k)$-vector that is in some sense "complementary." What exactly do we mean by complementary? A good first example is a 2-vector in $\mathbb{R}^3$:



Just as a plane in $\mathbb{R}^3$ can be identified with its unit normal (which spans its orthogonal complement), a 2-vector $u \wedge v$ can also be identified with some vector in the normal direction. But which vector? Unlike a linear subspace, we need to pick a definite magnitude and direction for the 1-vector $\star(u \wedge v)$. Here there is no "best" choice; we simply need to adopt a convention and stick with it—a good analogy is the *right hand rule* used to determine the direction of a cross product $u \times v$. For a 2-vector $u \wedge v$, we'll ask that

$$\det(u, v, \star(u \wedge v)) > 0,$$

i.e., the determinant of the two vectors comprising $u \wedge v$ and the third vector given by its Hodge star should be positive. In fact, this rule corresponds to the usual right-hand rule in the sense that $\star(u \wedge v)$ points in the same direction as $u \times v$. What about the magnitude? Again we have a rule based on the determinant—in particular, in the special case of two *orthonormal* vectors u_1, u_2, we ask that

$$\det(u_1, u_2, \star(u_1 \wedge u_2)) = 1.$$

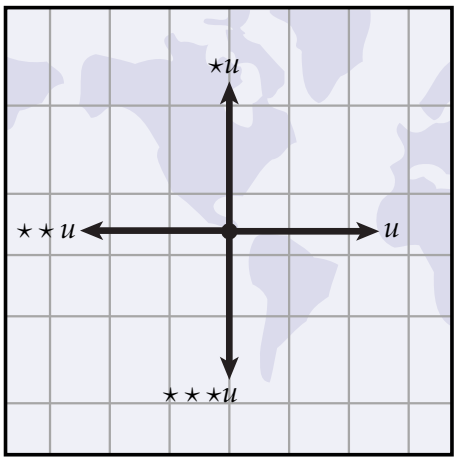
Since vectors in $\mathbb{R}^n$ can always be expressed in an orthonormal basis, this rule uniquely pins down the Hodge star for any 2-vector. In particular, we now really have $\star(u \wedge v) = u \times v$, *i.e.*, for two vectors in Euclidean $\mathbb{R}^3$ applying the wedge and then the star is equivalent to taking the cross product. (But it will not be this easy in general!)

More generally, suppose $e_1, \ldots, e_n$ is an orthonormal basis for $\mathbb{R}^n$. If we start out with k orthonormal vectors $u_1, \ldots, u_k$, then the Hodge star is uniquely determined by the relationship

$$(u_1 \wedge \cdots \wedge u_k) \wedge \star(u_1 \wedge \cdots \wedge u_k) = e_1 \wedge \cdots \wedge e_n.$$

In short: if we wedge together a k-dimensional "unit volume" with the complementary $(n-k)$-dimensional unit volume," we should get the one and only n-dimensional unit volume on $\mathbb{R}^n$.

An important special case (especially for thinking about surfaces) is the Hodge star of 1-vectors in $\mathbb{R}^2$, *i.e.*, the Hodge star of ordinary vectors in the plane. Here things are easy to visualize: if we have a 1-vector u, then its Hodge star $\star u$ will be an $(n-k)$-vector. But since $n-k=2-1=1$, we just get another 1-vector, orthogonal to u. For instance, if u points "east" on a map, then $\star u$ will point "north":



As we continue to apply the Hodge star, we get a vector that points west, then south, then back to east again. In other words, in 2D the Hodge star is just a quarter-rotation in the counter-clockwise direction.

Finally, we can think about the interaction between the Hodge star and the wedge product. For instance, for two 1-vectors u, v in $\mathbb{R}^3$, we have

$$\star(u+v) = \star u + \star v,$$

since adding two vectors and then rotating them by 90 degrees is no different from rotating them each individually and then adding them. More generally, this same identity holds for any two k-vectors in any dimension, *i.e.*, the Hodge star distributes over addition (can you draw other pictures that make this idea clearer?).

4.2. Examples of Wedge and Star in $\mathbb{R}^n$

To make all these ideas a bit more concrete, let's consider some concrete examples. These examples aren't meant to be particularly "deep," but rather just demonstrate the basic mechanics of doing calculations with k-vectors. (You'll see some more interesting examples in your homework!) Here we'll express (1-)vectors v in an orthonormal basis $e_1, \ldots, e_n$. For instance, in 2D $v := e_1 + e_2$ is a vector of length $\sqrt{2}$ making a 45° angle with the horizontal.

EXAMPLE 1. Let $u := e_1 + 2e_2$ and $v := e_1 + e_2 - e_3$ be 1-vectors in $\mathbb{R}^3$. Then their wedge product is given by

$$\begin{aligned} u \wedge v &= (e_1 + 2e_2) \wedge (e_1 + e_2 - e_3) \\ &= e_1 \wedge (e_1 + e_2 - e_3) + 2e_2 \wedge (e_1 + e_2 - e_3) \\ &= \cancelto{0}{e_1 \wedge e_1} + e_1 \wedge e_2 - e_1 \wedge e_3 + 2e_2 \wedge e_1 + \cancelto{0}{2e_2 \wedge e_2} - 2e_2 \wedge e_3 \\ &= e_1 \wedge e_2 - 2e_1 \wedge e_2 - e_1 \wedge e_3 - 2e_2 \wedge e_3 \\ &= -e_1 \wedge e_2 - e_1 \wedge e_3 - 2e_2 \wedge e_3. \end{aligned}$$

There are a couple things to notice in this calculation. First, any term $e_i \wedge e_i$ cancels to zero. Do you remember why? It's essentially because the parallelogram spanned by two copies of the same vector has zero area. Also notice that at one point we replace $2e_2 \wedge e_1$ with $-2e_1 \wedge e_2$. Why did we do that? Because $e_1 \wedge e_2$ and $e_2 \wedge e_1$ describe the same 2-vector, but with opposite orientation.

EXAMPLE 2. Let $w := -e_1 \wedge e_2 - e_1 \wedge e_3 - 2e_2 \wedge e_3$ be the 2-vector from the previous example. Its Hodge star is given by

$$\begin{aligned} \star w &= \star(-e_1 \wedge e_2 - e_1 \wedge e_3 - 2e_2 \wedge e_3) \\ &= -\star(e_1 \wedge e_2) - \star(e_1 \wedge e_3) - 2\star(e_2 \wedge e_3) \\ &= -e_3 - (-e_2) - 2e_1 \\ &= -2e_1 + e_2 - e_3. \end{aligned}$$

The main thing we did here was use the *right hand rule* to determine which direction the wedge of two basis vectors points. For instance, just as $e_1 \times e_2 = e_3$ when working with the cross product, $\star(e_1 \wedge e_2) = e_3$ when working with the wedge product and the Hodge star. A more detailed discussion of these relationships, and about bases in exterior algebra, can be found in Section 4.5.1.

EXAMPLE 3. Let $u := e_1 + e_2 + e_3$, $v := e_1 + 2e_2 + 3e_3$, and $w := e_1 - e_3$ be 1-vectors in $\mathbb{R}^3$, and suppose we want to compute $u \wedge v \wedge w$. Since the wedge product is associative, we can start out by just computing either $u \wedge v$ or $v \wedge w$, and then wedging the result with the remaining 1-vector. For instance, we have

$$\begin{aligned} v \wedge w &= (e_1 + 2e_2 + 3e_3) \wedge (e_1 - e_3) \\ &= \cancelto{0}{e_1 \wedge e_1} - e_1 \wedge e_3 + 2e_2 \wedge e_1 - 2e_2 \wedge e_3 + 3e_3 \wedge e_1 - \cancelto{0}{3e_3 \wedge e_3} \\ &= -2e_1 \wedge e_2 - 4e_1 \wedge e_3 - 2e_2 \wedge e_3. \end{aligned}$$

Wedging with u then yields

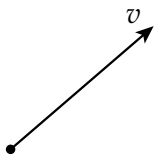
$$\begin{aligned} u \wedge (v \wedge w) &= (e_1 + e_2 + e_3) \wedge (-2e_1 \wedge e_2 - 4e_1 \wedge e_3 - 2e_2 \wedge e_3) \\ &= -2e_1 \wedge e_2 \wedge e_3 - 4e_2 \wedge e_1 \wedge e_3 - 2e_3 \wedge e_1 \wedge e_2 \\ &= -2e_1 \wedge e_2 \wedge e_3 + 4e_1 \wedge e_2 \wedge e_3 - 2e_1 \wedge e_2 \wedge e_3 \\ &= 0. \end{aligned}$$

In the second calculation we avoided a lot of work by noticing that any term involving multiple copies of the same basis 1-vector (*e.g.*, $e_1 \wedge e_1 \wedge e_2$) would have zero volume, since two of the edges of the corresponding little parallelepiped would be parallel. Hence, we can just write down the three remaining terms where all three bases show up (*e.g.*, $e_2 \wedge e_3 \wedge e_1$). By repeatedly swapping pairs of bases, we can put all such 3-vectors into a canonical form (*e.g.*, $e_2 \wedge e_3 \wedge e_1 = -e_2 \wedge e_1 \wedge e_3 = e_1 \wedge e_2 \wedge e_3$), at which point we just have several copies of the unit 3-vector $e_1 \wedge e_2 \wedge e_3$ scaled by some magnitude. In this case, the magnitudes of all the terms summed to zero. What does that mean geometrically? It must mean that our original 1-vectors u, v, and w are not linearly independent, *i.e.*, they describe a "flat" 3-vector with zero volume.

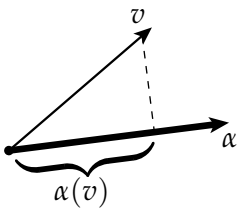
4.3. Vectors and 1-Forms

Now that we have a basic language for working with "little volumes," we continue with the second part of our story, about exterior calculus.

Once upon a time there was a vector named v:



What information does v encode? One way to inspect a vector is to determine its extent or *length* along a given direction. For instance, we can pick some arbitrary direction α and record the length of the shadow cast by v along α:



The result is simply a number, which we can denote $\alpha(v)$. The notation here is meant to emphasize the idea that α is a function: in particular, it's a *linear* function that eats a vector and produces a scalar. Any such function is called a *1-form* (also known as a *covector*).

Of course, it's clear from the picture that the space of all 1-forms looks a lot like the space of all vectors: we just had to pick some direction to measure along. But often there is good reason to distinguish between vectors and 1-forms—the distinction is not unlike the one made between *row vectors* and *column vectors* in linear algebra. For instance, even though rows and column both represent "vectors," we only allow ourselves to multiply rows with columns:

$$\begin{bmatrix} \alpha_1 & \cdots & \alpha_n \end{bmatrix} \begin{bmatrix} v^1 \\ \vdots \\ v^n \end{bmatrix}.$$

If we wanted to multiply, say, two column vectors, we would first have to take the *transpose* of one of them to convert it into a row:

$$v^T v = \begin{bmatrix} v^1 & \cdots & v^n \end{bmatrix} \begin{bmatrix} v^1 \\ \vdots \\ v^n \end{bmatrix}.$$

Same deal with vectors and 1-forms, except that now we have two different operations: *sharp* ($\sharp$), which converts a 1-form into a vector, and *flat* ($\flat$) which converts a vector into a 1-form. For instance, it's perfectly valid to write $v^\flat(v)$ or $\alpha(\alpha^\sharp)$, since in either case we're feeding a vector to a 1-form. The operations $\sharp$ and $\flat$ are called the *musical isomorphisms*.

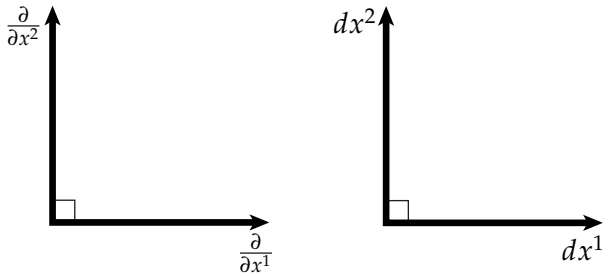
All this fuss over 1-forms versus vectors (or even row versus column vectors) may seem like much ado about nothing. And indeed, in a *flat* space like the plane, the difference between the two is pretty superficial. In *curved* spaces, however, there's an important distinction between vectors and 1-forms—in particular, we want to make sure that we're taking "measurements" in the right space. For instance, suppose we want to measure the length of a vector v along the direction of another vector u. It's important to remember that tangent vectors get stretched out by the map $f : \mathbb{R}^2 \supset M \to \mathbb{R}^3$ that takes us from the plane to some surface in $\mathbb{R}^3$. Therefore, the operations $\sharp$ and $\flat$ should satisfy relationships like

$$u^\flat(v) = g(u, v)$$

where g is the metric induced by f. This way we're really measuring how things behave in the "stretched out" space rather than the initial domain M.

4.3.1. Coordinates. Until now we've intentionally avoided the use of *coordinates*—in other words, we've tried to express geometric relationships without reference to any particular *coordinate system* $x_1, \ldots, x_n$. Why avoid coordinates? Several reasons are often cited (people will mumble something about "invariance"), but the real reason is quite simply that coordinate-free expressions tend to be shorter, sweeter, and easier to extract meaning from. This approach is also particularly valuable in geometry processing, because many coordinate-free expressions translate naturally to basic operations on meshes.

Yet coordinates are still quite valuable in a number of situations. Sometimes there's a special coordinate basis that greatly simplifies analysis—recall our discussion of principal curvature directions, for instance. At other times there's simply no obvious way to prove something *without* coordinates. For now we're going to grind out a few basic facts about exterior calculus in coordinates; at the end of the day we'll keep whatever nice coordinate-free expressions we find and politely forget that coordinates ever existed!



Let's setup our coordinate system. For reasons that will become clear later, we're going to use the symbols $\frac{\partial}{\partial x^1}, \ldots, \frac{\partial}{\partial x^n}$ to represent an orthonormal basis for vectors in $\mathbb{R}^n$, and use $dx^1, \ldots, dx^n$ to denote the corresponding 1-form basis. In other words, any vector v can be written as a linear combination

$$v = v^1 \frac{\partial}{\partial x^1} + \cdots + v^n \frac{\partial}{\partial x^n},$$

and any 1-form can be written as a linear combination

$$\alpha = \alpha_1 dx^1 + \cdots + \alpha_n dx^n.$$

To keep yourself sane at this point, you should *completely ignore the fact* that the symbols $\frac{\partial}{\partial x^i}$ and dx^i look like derivatives—they're simply collections of unit-length orthogonal bases, as depicted above.

The two bases dx^i and $\frac{\partial}{\partial x^i}$ are often referred to as *dual bases*, meaning they satisfy the relationship

$$dx^i\left(\frac{\partial}{\partial x^j}\right) = \delta^i_j = \begin{cases} 1, & i = j \\ 0, & \text{otherwise.} \end{cases}$$

This relationship captures precisely the behavior we're looking for: a vector $\frac{\partial}{\partial x^i}$ "casts a shadow" on the 1-form dx^j only if the two bases point in the same direction. Using this relationship, we can work out that

$$\alpha(v) = \sum_i \alpha_i dx^i \left(\sum_j v^j \frac{\partial}{\partial x^j}\right) = \sum_i \alpha_i v^i$$

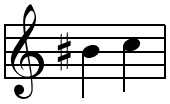
i.e., the *pairing* of a vector and a 1-form looks just like the standard Euclidean inner product.

4.3.2. Notation. It's worth saying a few words about notation. First, vectors and vector fields tend to be represented by letters from the end of the Roman alphabet (u, v, w or X, Y, Z, respectively), whereas 1-forms are given lowercase letters from the beginning of the Greek alphabet (α, β, γ, etc.). Although one often makes a linguistic distinction between a "vector" (meaning a single arrow) and a "vector field" (meaning an arrow glued to every point of a space), there's an unfortunate precedent to use the term "1-form" to refer to both ideas—sadly, nobody ever says "1-form field!" Scalar fields or *0-forms* are often given letters from the middle of the Roman alphabet (f, g, h) or maybe lowercase Greek letters from somewhere near the end (ϕ, ψ, etc.).

You may also notice that we've been very particular about the placement of indices: coefficients v^i of vectors have indices *up*, coefficients α_i of 1-forms have indices *down*. Similarly, vector bases $\frac{\partial}{\partial x^i}$ have indices down (they're in the denominator), and 1-form bases dx^i have indices up. The reason for being so neurotic is to take advantage of *Einstein summation notation*: any time a pair of variables is indexed by the same letter i in both the "up" and "down" position, we interpret this as a sum over all possible values of i:

$$\alpha_i v^i = \sum_i \alpha_i v^i.$$

The placement of indices also provides a cute mnemonic for the musical isomorphisms $\sharp$ and $\flat$. In musical notation $\sharp$ indicates a half-step increase in pitch, corresponding to an upward movement on the staff. For instance, both notes below correspond to a "C" with the same pitch[3]:



Therefore, to go from a 1-form to a vector we *raise* the indices. For instance, in a *flat* space we don't have to worry about the metric and so a 1-form

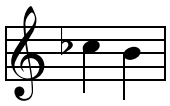
$$\alpha = \alpha_1 dx^1 + \cdots + \alpha_n dx^n$$

becomes a vector

$$\alpha^\sharp = \alpha^1 \frac{\partial}{\partial x^1} + \cdots + \alpha^n \frac{\partial}{\partial x^n}.$$

Similarly, $\flat$ indicates a decrease in pitch and a downward motion on the staff:

[3]At least on a *tempered* instrument!



and so ♭ *lowers* the indices of a vector to give us a 1-form—e.g.,

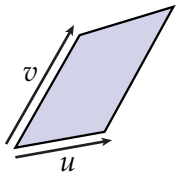
$$v = v^1 \frac{\partial}{\partial x^1} + \cdots + v^n \frac{\partial}{\partial x^n}.$$

becomes

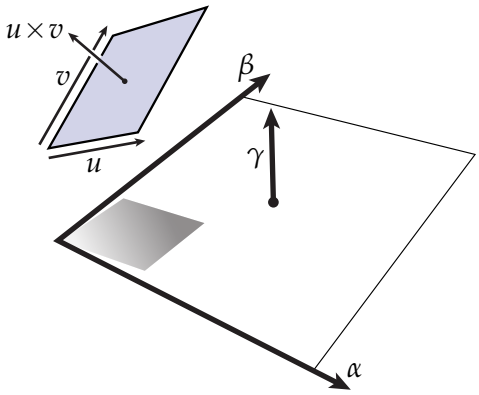
$$v^\flat = v_1 dx^1 + \cdots + v_n dx^n.$$

4.4. Differential Forms and the Wedge Product

In the last subsection we measured the length of a vector by projecting it onto different coordinate axes; this measurement process effectively defined what we call a *1-form*. But what happens if we have a collection of vectors? For instance, consider a pair of vectors u, v sitting in $\mathbb{R}^3$:



We can think of these vectors as defining a *parallelogram*, and much like we did with a single vector we can measure this parallelogram by measuring the size of the "shadow" it casts on some plane:



For instance, suppose we represent this plane via a pair of unit orthogonal 1-forms α and β. Then the projected vectors have components

$$\begin{aligned} u' &= (\alpha(u), \beta(u))^T, \\ v' &= (\alpha(v), \beta(v))^T, \end{aligned}$$

hence the (signed) projected area is given by the cross product

$$u' \times v' = \alpha(u)\beta(v) - \alpha(v)\beta(u).$$

Since we want to measure a lot of projected volumes in the future, we'll give this operation the special name "$\alpha \wedge \beta$":

$$\alpha \wedge \beta(u, v) := \alpha(u)\beta(v) - \alpha(v)\beta(u).$$

As you may have already guessed, $\alpha \wedge \beta$ is what we call a *2-form*. Ultimately we'll interpret the symbol $\wedge$ (pronounced "wedge") as a binary operation on differential forms called the *wedge product*. Algebraic properties of the wedge product follow *directly* from the way signed volumes behave. For instance, notice that if we reverse the order of our axes α, β the sign of the area changes. In other words, the wedge product is *antisymmetric*:

$$\alpha \wedge \beta = -\beta \wedge \alpha.$$

An important consequence of antisymmetry is that the wedge of any 1-form with itself is zero:

$$\begin{aligned} &\alpha \wedge \alpha = -\alpha \wedge \alpha \\ &\Rightarrow \alpha \wedge \alpha = 0. \end{aligned}$$

But don't let this statement become a purely algebraic fact! Geometrically, why should the wedge of two 1-forms be zero? Quite simply because it represents projection onto a plane of zero area! (I.e., the plane spanned by α and α.)

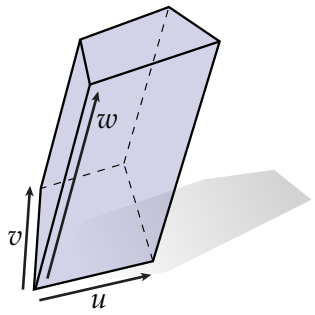
Next, consider the projection onto two different planes spanned by α, β and α, γ. The sum of the projected areas can be written as

$$\begin{aligned} \alpha \wedge \beta(u,v) + \alpha \wedge \gamma(u,v) &= \alpha(u)\beta(v) - \alpha(v)\beta(u) + \alpha(u)\gamma(v) - \alpha(v)\gamma(u) \\ &= \alpha(u)(\beta(v) + \gamma(v)) - \alpha(v)(\beta(u) + \gamma(u)) \\ &= (\alpha \wedge (\beta + \gamma))(u,v), \end{aligned}$$

or in other words $\wedge$ distributes over $+$:

$$\alpha \wedge (\beta + \gamma) = \alpha \wedge \beta + \alpha \wedge \gamma.$$

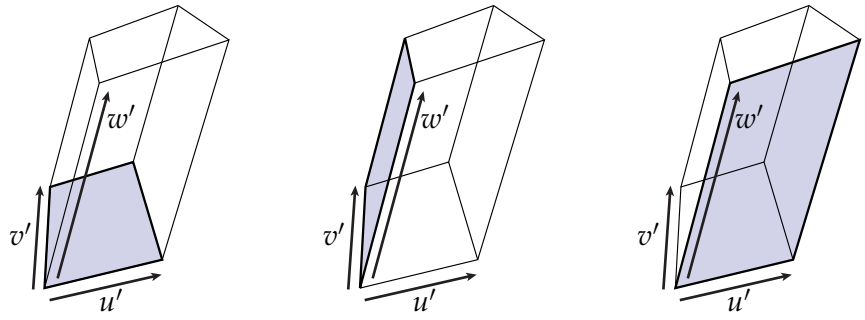
Finally, consider three vectors u, v, w that span a volume in $\mathbb{R}^3$:



We'd like to consider the projection of this volume onto the volume spanned by three 1-forms α, β, and γ, but the projection of one volume onto another is a bit difficult to visualize! For now you can just cheat and imagine that $\alpha = dx^1$, $\beta = dx^2$, and $\gamma = dx^3$ so that the mental picture for the projected volume looks just like the volume depicted above. One way to write the projected volume is as the determinant of the projected vectors u', v', and w':

$$\alpha \wedge \beta \wedge \gamma(u,v,w) := \det\left(\begin{bmatrix} u' & v' & w' \end{bmatrix}\right) = \det\left(\begin{bmatrix} \alpha(u) & \alpha(v) & \alpha(w) \\ \beta(u) & \beta(v) & \beta(w) \\ \gamma(u) & \gamma(v) & \gamma(w) \end{bmatrix}\right).$$

(Did you notice that the determinant of the upper-left 2x2 submatrix also gives us the wedge product of two 1-forms?) Alternatively, we could express the volume as the area of one of the faces times the length of the remaining edge:



Thinking about things this way, we might come up with an alternative definition of the wedge product in terms of the *triple product*:

$$\begin{aligned} \alpha \wedge \beta \wedge \gamma(u, v, w) &= (u' \times v') \cdot w' \\ &= (v' \times w') \cdot u' \\ &= (w' \times u') \cdot v' \end{aligned}$$

The important thing to notice here is that *order* is not important—we always get the same volume, regardless of which face we pick (though we still have to be a bit careful about *sign*). A more algebraic way of saying this is that the wedge product is *associative*:

$$(\alpha \wedge \beta) \wedge \gamma = \alpha \wedge (\beta \wedge \gamma).$$

In summary, the wedge product of k 1-forms gives us a k-form, which measures the projected volume of a collection of k vectors. As a result, the wedge product has the following properties for any k-form α, l-form β, and m-form γ:

- **Antisymmetry**: $\alpha \wedge \beta = (-1)^{kl} \beta \wedge \alpha$
- **Associativity**: $\alpha \wedge (\beta \wedge \gamma) = (\alpha \wedge \beta) \wedge \gamma$

and in the case where β and γ have the same degree (*i.e.*, $l = m$) we have

- **Distributivity**: $\alpha \wedge (\beta + \gamma) = \alpha \wedge \beta + \alpha \wedge \gamma$

A separate fact is that a k-form is *antisymmetric* in its arguments—in other words, swapping the relative order of two "input" vectors changes only the *sign* of the volume. For instance, if α is a 2-form then $\alpha(u, v) = -\alpha(v, u)$. In general, an *even* number of swaps will preserve the sign; an *odd* number of swaps will negate it. (One way to convince yourself is to consider what happens to the determinant of a matrix when you exchange two of its columns.) Finally, you'll often hear people say that k-forms are "multilinear"—all this means is that if you keep all but one of the vectors fixed, then a k-form looks like a linear map. Geometrically this makes sense: k-forms are built up from k *linear* measurements of length (essentially just k different dot products).

4.4.1. Vector-Valued Forms. Up to this point we've considered only *real-valued* k-forms—for instance, $\alpha(u)$ represents the length of the vector u along the direction α, which can be expressed as a single real number. In general, however, a k-form can "spit out" all kinds of different values. For instance, we might want to deal with quantities that are described by complex numbers ($\mathbb{C}$) or vectors in some larger vector space (e.g., $\mathbb{R}^n$).

A good example of a vector-valued k-form is our map $f : M \to \mathbb{R}^3$ which represents the geometry of a surface. In the language of exterior calculus, f is an $\mathbb{R}^3$*-valued 0-form*: at each point p of M, it takes *zero* vectors as input and produces a point $f(p)$ in $\mathbb{R}^3$ as output. Similarly, the differential df is an $\mathbb{R}^3$-valued 1-form: it takes *one* vector (some direction u in the plane) and maps it to a value $df(u)$ in $\mathbb{R}^3$ (representing the "stretched out" version of u).

More generally, if E is a vector space then an *E-valued k-form* takes k vectors to a single value in E. However, we have to be a bit careful here. For instance, think about our definition of a 2-form:

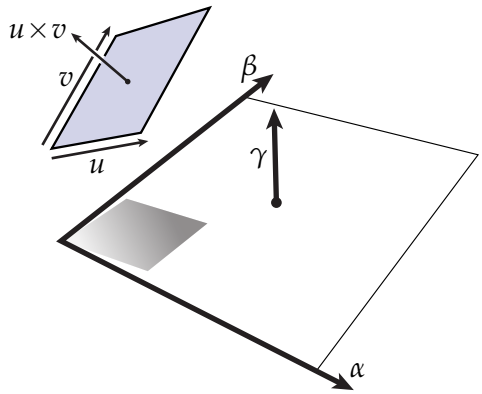
$$\alpha \wedge \beta(u, v) := \alpha(u)\beta(v) - \alpha(v)\beta(u).$$

If α and β are both E-valued 1-forms, then $\alpha(u)$ and $\beta(v)$ are both *vectors* in E. But how do you multiply two vectors? In general there may be no good answer: not every vector space comes with a natural notion of multiplication.

However, there are plenty of spaces that *do* come with a well-defined product—for instance, the product of two complex numbers $a + bi$ and $c + di$ is given by $(ac - bd) + (ad + bc)i$, so we have no trouble explicitly evaluating the expression above. In other cases we simply have to say which product we want to use—in $\mathbb{R}^3$ for instance we could use the cross product $\times$, in which case an $\mathbb{R}^3$-valued 2-form looks like this:

$$\alpha \wedge \beta(u, v) := \alpha(u) \times \beta(v) - \alpha(v) \times \beta(u).$$

4.5. Hodge Duality

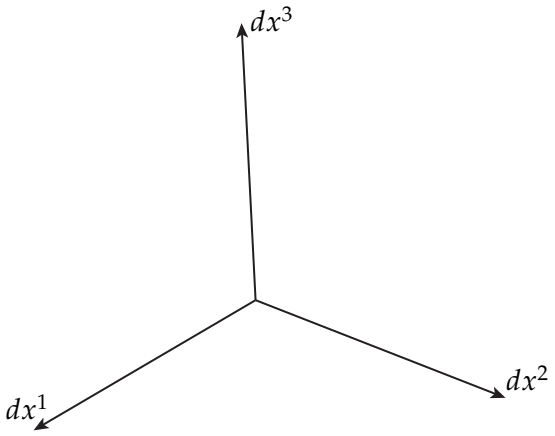


Previously we saw that a k-form measures the (signed) projected volume of a k-dimensional parallelepiped. For instance, a 2-form measures the area of a parallelogram projected onto some plane, as depicted above. But here's a nice observation: a plane in $\mathbb{R}^3$ can be described *either* by a pair of basis directions (α, β), *or* by a normal direction γ. So rather than measuring projected area, we could instead measure how well the normal of a parallelogram (u, v) lines up with the normal of our plane. In other words, we could look for a 1-form γ such that

$$\gamma(u \times v) = \alpha \wedge \beta(u, v).$$

This observation captures the idea behind *Hodge duality*: a k-dimensional volume in an n-dimensional space can be specified either by k directions or by a complementary set of $(n-k)$ directions. There should therefore be some kind of natural correspondence between k-forms and $(n-k)$-forms.

4.5.1. Differential Forms and the Hodge Star. Let's investigate this idea further by constructing an explicit basis for the space of 0-forms, 1-forms, 2-forms, etc.—to keep things manageable we'll work with $\mathbb{R}^3$ and its standard coordinate system (x^1, x^2, x^3). 0-forms are easy: any 0-form can be thought of as some function times the *constant* 0-form, which we'll denote "1." We've already seen the 1-form basis dx^1, dx^2, dx^3, which looks like the standard orthonormal basis of a vector space:



What about 2-forms? Well, consider that any 2-form can be expressed as the wedge of two 1-forms:

$$\alpha \wedge \beta = (\alpha_i dx^i) \wedge (\beta_j dx^j) = \alpha_i \beta_j dx^i \wedge dx^j.$$

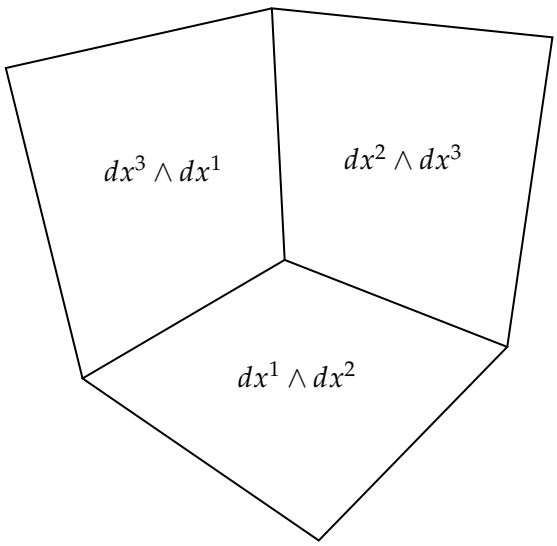
In other words, any 2-form looks like some linear combination of the basis 2-forms $dx^i \wedge dx^j$. How many of these bases are there? Initially it looks like there are a bunch of possibilities:

$$\begin{array}{ccc} dx^1 \wedge dx^1 & dx^1 \wedge dx^2 & dx^1 \wedge dx^3 \\ dx^2 \wedge dx^1 & dx^2 \wedge dx^2 & dx^2 \wedge dx^3 \\ dx^3 \wedge dx^1 & dx^3 \wedge dx^2 & dx^3 \wedge dx^3 \end{array}$$

But of course, not all of these guys are distinct: remember that the wedge product is antisymmetric ($\alpha \wedge \beta = -\beta \wedge \alpha$), which has the important consequence $\alpha \wedge \alpha = 0$. So really our table looks more like this:

$$\begin{array}{ccc} 0 & dx^1 \wedge dx^2 & -dx^3 \wedge dx^1 \\ -dx^1 \wedge dx^2 & 0 & dx^2 \wedge dx^3 \\ dx^3 \wedge dx^1 & -dx^2 \wedge dx^3 & 0 \end{array}$$

and we're left with only three distinct bases: $dx^2 \wedge dx^3$, $dx^3 \wedge dx^1$, and $dx^1 \wedge dx^2$. Geometrically all we've said is that there are three linearly-independent "planes" in $\mathbb{R}^3$:



How about 3-form bases? We certainly have at least one:

$$dx^1 \wedge dx^2 \wedge dx^3.$$

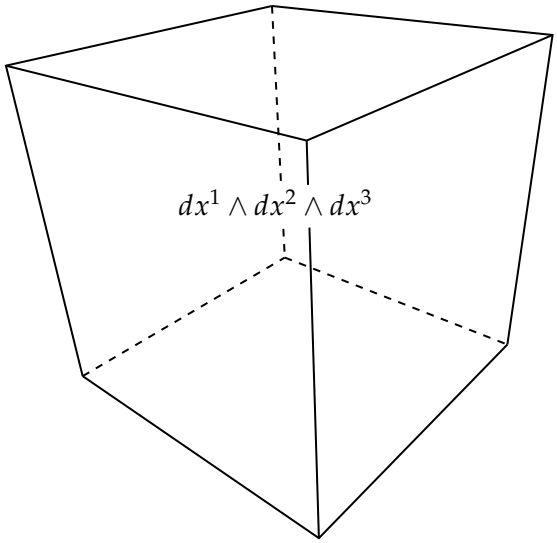
Are there any others? Again the antisymmetry of $\wedge$ comes into play: many potential bases are just permutations of this first one:

$$dx^2 \wedge dx^3 \wedge dx^1 = -dx^2 \wedge dx^1 \wedge dx^3 = dx^1 \wedge dx^2 \wedge dx^3,$$

and the rest vanish due to the appearance of repeated 1-forms:

$$dx^2 \wedge dx^1 \wedge dx^2 = -dx^2 \wedge dx^2 \wedge dx^1 = 0 \wedge dx^1 = 0.$$

In general there is only *one* basis n-form $dx^1 \wedge \cdots \wedge dx^n$, which measures the usual Euclidean volume of a parallelpiped:



Finally, what about 4-forms on $\mathbb{R}^3$? At this point it's probably pretty easy to see that there are none, since we'd need to pick four *distinct* 1-form bases from a collection of only three. Geometrically: there are no four-dimensional volumes contained in $\mathbb{R}^3$! (Or volumes of any greater dimension, for that matter.) The complete list of k-form bases on $\mathbb{R}^3$ is then

- **0-form bases:** 1
- **1-form bases:** dx^1, dx^2, dx^3
- **2-form bases:** $dx^2 \wedge dx^3$, $dx^3 \wedge dx^1$, $dx^1 \wedge dx^2$
- **3-form bases:** $dx^1 \wedge dx^2 \wedge dx^3$,

which means the *number* of bases is 1, 3, 3, 1. In fact you may see a more general pattern here: the number of k-form bases on an n-dimensional space is given by the binomial coefficient

$$\begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!(n-k)!}$$

(i.e., "n choose k"), since we want to pick k distinct 1-form bases and don't care about the order. An important identity here is

$$\begin{pmatrix} n \\ k \end{pmatrix} = \begin{pmatrix} n \\ n-k \end{pmatrix},$$

which, as anticipated, means that we have a one-to-one relationship between k-forms and $(n-k)$-forms. In particular, we can identify any k-form with its complement. For example, on $\mathbb{R}^3$ we have

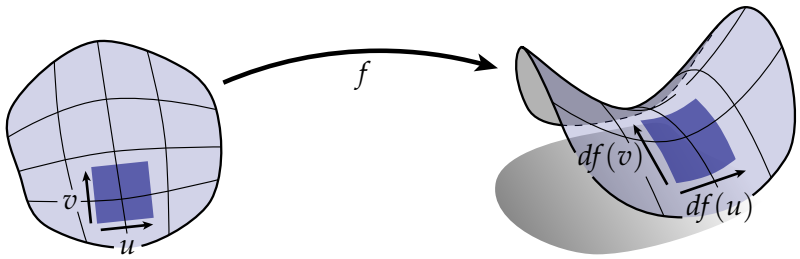
$$\begin{aligned}
\star 1 &= dx^1 \wedge dx^2 \wedge dx^3 \\
\star\, dx^1 &= dx^2 \wedge dx^3 \\
\star\, dx^2 &= dx^3 \wedge dx^1 \\
\star\, dx^3 &= dx^1 \wedge dx^2 \\
\star(dx^1 \wedge dx^2) &= dx^3 \\
\star(dx^2 \wedge dx^3) &= dx^1 \\
\star(dx^3 \wedge dx^1) &= dx^2 \\
\star(dx^1 \wedge dx^2 \wedge dx^3) &= 1
\end{aligned}$$

The map $\star$ (pronounced "star") is called the *Hodge star* and captures this idea that planes can be identified with their normals and so forth. More generally, on any *flat* space we have

$$\star(dx^{i_1} \wedge dx^{i_2} \wedge \cdots \wedge dx^{i_k}) = dx^{i_{k+1}} \wedge dx^{i_{k+2}} \wedge \cdots \wedge dx^{i_n},$$

where $(i_1, i_2, \ldots, i_n)$ is any *even* permutation of $(1, 2, \ldots, n)$.

4.5.2. The Volume Form.



So far we've been talking about measuring volumes in *flat* spaces like $\mathbb{R}^n$. But how do we take measurements in a curved space? Let's think about our usual example of a surface $f : \mathbb{R}^2 \supset M \to \mathbb{R}^3$. If we consider a region of our surface spanned by a pair of orthogonal unit vectors $u, v \in \mathbb{R}^2$, it's clear that we don't want the area $dx^1 \wedge dx^2(u, v) = 1$ since that just gives us the area in the plane. What we really want is the area of this region after it's been "stretched-out" by the map f. In other words, we want the size of the corresponding parallelogram in $\mathbb{R}^3$, spanned by the vectors $df(u)$ and $df(v)$.

EXERCISE 4.1

Letting $u, v \in \mathbb{R}^2$ be orthonormal (as above), show that

$$|df(u) \times df(v)| = \sqrt{\det(g)},$$

i.e., show that the "stretching factor" as we go from the plane to the surface is given by the square root of the *determinant* of the metric

$$\det(g) := g(u, u)g(v, v) - g(u, v)^2.$$

Hint: remember that the induced metric is just the usual Euclidean dot product of the embedded vectors: $g(u, v) := df(u) \cdot df(v)$.

Therefore, we can measure the area of *any* little region on our surface by simply scaling the volume in the plane by the determinant of the metric, *i.e.*, by applying the 2-form $\sqrt{\det(g)}dx^1 \wedge dx^2$ to two vectors u, v spanning the region of interest. More generally, the n-form

$$\omega := \sqrt{\det(g)}dx^1 \wedge \cdots \wedge dx^n$$

is called the *volume form*, and will play a key role when we talk about integration.

On curved spaces, we'd also like the Hodge star to capture the fact that volumes have been stretched out. For instance, it makes a certain amount of sense to identify the constant function 1 with the volume form ω, since ω really represents *unit* volume on the curved space:

$$\star 1 = \omega$$

4.5.3. The Inner Product on k-Forms. More generally we'll ask that any n-form constructed from a pair of k-forms α and β satisfies

$$\alpha \wedge \star\beta = \langle\langle \alpha, \beta \rangle\rangle \omega,$$

where $\langle\langle \alpha, \beta \rangle\rangle = \sum_i \alpha_i \beta_i$ is the inner product on k-forms. In fact, some authors use this relationship as the *definition* of the wedge product—in other words, they'll start with something like, "the wedge product is the unique binary operation on k-forms such that $\alpha \wedge \star\beta = \langle\langle \alpha, \beta \rangle\rangle \omega$," and from there derive all the properties we've established above. This treatment is a bit abstract, and makes it far too easy to forget that the wedge product has an extraordinarily concrete geometric meaning. (It's certainly not the way Hermann Grassmann thought about it when he invented exterior algebra!). In practice, however, this identity is quite useful. For instance, if u and v are vectors in $\mathbb{R}^3$, then we can write

$$u \cdot v = \star \left(u^\flat \wedge \star v^\flat \right),$$

i.e., on a flat space we can express the usual Euclidean inner product via the wedge product. Is it clear *geometrically* that this identity is true? Think about what it says: the Hodge star turns v into a plane with v as a normal. We then build a volume by extruding this plane along the direction u. If u and v are nearly parallel the volume will be fairly large; if they're nearly orthogonal the volume will be quite shallow. (But to be sure we really got it right, you should try verifying this identity in coordinates!) Similarly, we can express the Euclidean cross product as just

$$u \times v = (\star(u^\flat \wedge v^\flat))^\sharp,$$

i.e., we can create a plane with normal $u \times v$ by wedging together the two basis vectors u and v. (Again, working this fact out in coordinates may help soothe your paranoia.)

4.6. Differential Operators

Originally we set out to develop *exterior calculus*. The objects we've looked at so far—k-forms, the wedge product $\wedge$ and the Hodge star $\star$—actually describe a more general structure called an *exterior algebra*. To turn our algebra into a *calculus*, we also need to know how quantities *change*, as well as how to *measure* quantities. In other words, we need some tools for *differentiation* and *integration*. Let's start with differentiation.

In our discussion of surfaces we briefly looked at the *differential* df of a surface $f : M \to \mathbb{R}^3$, which tells us something about the way tangent vectors get "stretched out" as we move from the domain M to a curved surface sitting in $\mathbb{R}^3$. More generally d is called the *exterior derivative* and is responsible for building up many of the differential operators in exterior calculus. The basic idea is that d tells us how quickly a k-form changes along *every possible direction*. But how exactly is it defined? So far we've seen only a high-level geometric description.

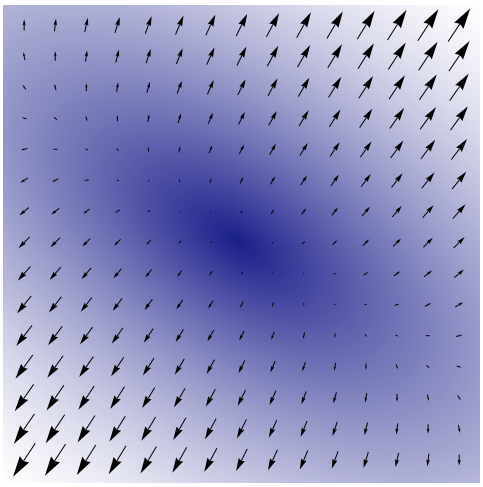
4.6.1. Div, Grad, and Curl. Before jumping into the exterior derivative, it's worth reviewing what the basic vector derivatives div, grad, and curl do, and more importantly, what they *look like*. The key player here is the operator ∇ (pronounced "nabla") which can be expressed in coordinates as the vector of all partial derivatives:

$$\nabla := \left(\frac{\partial}{\partial x^1}, \dots, \frac{\partial}{\partial x^n} \right)^T.$$

For instance, applying ∇ to a scalar function $\phi : \mathbb{R}^n \to \mathbb{R}$ yields the *gradient*

$$\nabla \phi = \left(\frac{\partial f}{\partial x^1}, \dots, \frac{\partial f}{\partial x^n} \right)^T,$$

which can be visualized as the direction of steepest ascent on some terrain:



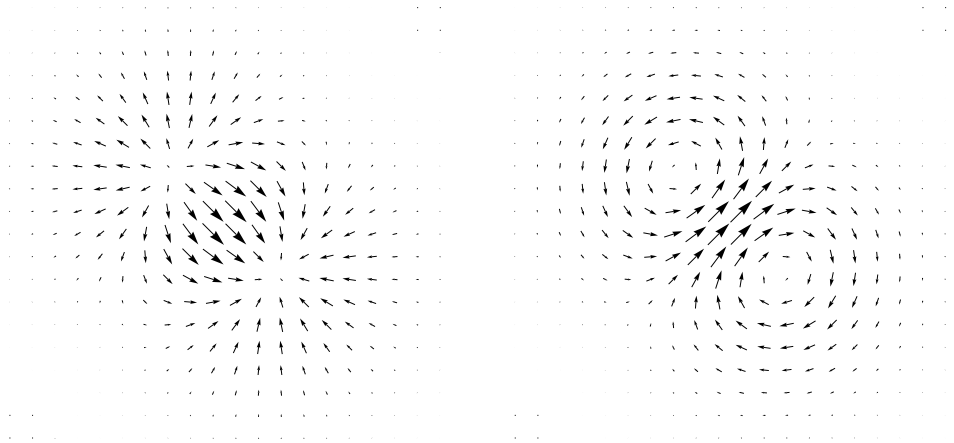
We can also apply ∇ to a vector field X in two different ways. The dot product gives us the *divergence*

$$\nabla \cdot X = \frac{\partial X^1}{\partial x^1} + \dots + \frac{\partial X^n}{\partial x^n}$$

which measures how quickly the vector field is "spreading out", and on $\mathbb{R}^3$ the cross product gives us the *curl*

$$\nabla \times X = \left(\frac{\partial X^3}{\partial x^2} - \frac{\partial X^2}{\partial x^3}, \frac{\partial X^1}{\partial x^3} - \frac{\partial X^3}{\partial x^1}, \frac{\partial X^2}{\partial x^1} - \frac{\partial X^1}{\partial x^2} \right),$$

which indicates how much a vector field is "spinning around." For instance, here's a pair of vector fields with a lot of divergence and a lot of curl, respectively:



(Note that in this case one field is just a 90-degree rotation of the other!) On a typical day it's a lot more useful to think of div, grad and curl in terms of these kinds of pictures rather than the ugly expressions above.

4.6.2. Think Differential. Not surprisingly, we can express similar notions using exterior calculus. However, these notions will be a bit easier to generalize (for instance, what does "curl" mean for a vector field in $\mathbb{R}^4$, where no cross product is defined?). Let's first take a look at the exterior derivative of 0-forms (i.e., functions), which is often just called the *differential*. To keep things simple, we'll start with *real*-valued functions $\phi : \mathbb{R}^n \to \mathbb{R}$. In coordinates, the differential is defined as

$$d\phi := \frac{\partial \phi}{\partial x^1} dx^1 + \cdots + \frac{\partial \phi}{\partial x^n} dx^n.$$

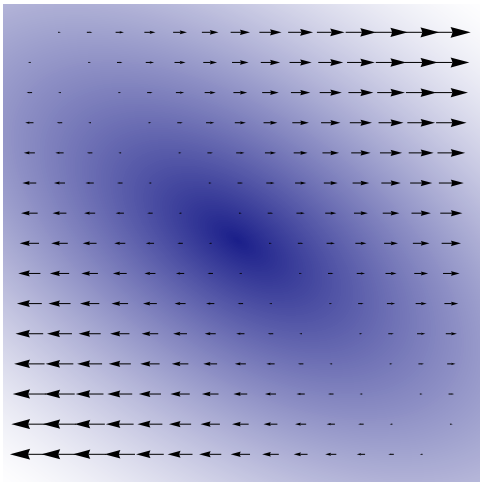
It's important to note that the terms $\frac{\partial \phi}{\partial x^i}$ actually correspond to partial derivatives of our function ϕ, whereas the terms dx^i simply denote an orthonormal basis for $\mathbb{R}^n$. In other words, you can think of $d\phi$ as just a list of all the partial derivatives of ϕ. Of course, this object looks a lot like the *gradient* $\nabla\phi$ we saw just a moment ago. And indeed the two are closely related, except for the fact that $\nabla\phi$ is a vector field and $d\phi$ is a 1-form. More precisely,

$$\nabla\phi = (d\phi)^\sharp.$$

4.6.3. Directional Derivatives. Another way to investigate the behavior of the exterior derivative is to see what happens when we stick a vector u into the 1-form df. In coordinates we get something that looks like a dot product between the gradient of f and the vector u:

$$df(u) = \frac{\partial f}{\partial x^1} u^1 + \cdots + \frac{\partial f}{\partial x^n} u^n.$$

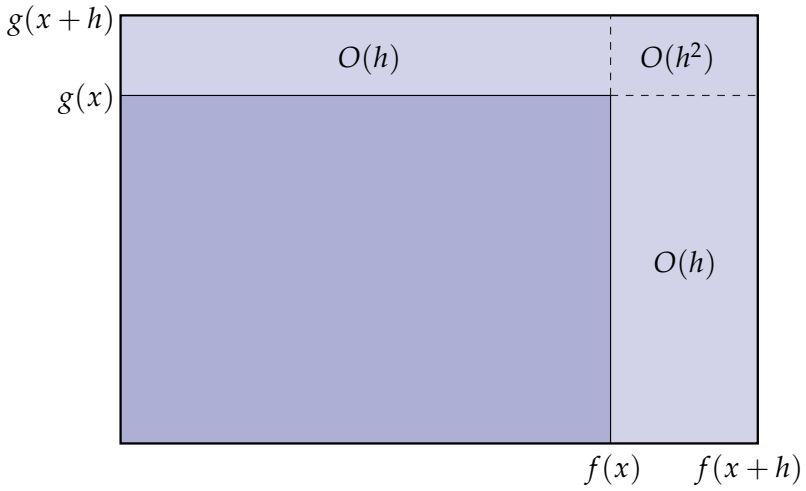
For instance, in $\mathbb{R}^2$ we could stick in the unit vector $u = (1, 0)$ to get the partial derivative $\frac{\partial f}{\partial x^1}$ along the first coordinate axis:



(Compare this picture to the picture of the gradient we saw above.) In general, $df(u)$ represents the *directional derivative* of f along the direction u. In other words, it tells us how quickly f changes if we take a short walk in the direction u. Returning again to vector calculus notation, we have

$$df(u) = u \cdot \nabla f.$$

4.6.4. Properties of the Exterior Derivative. How do derivatives of arbitrary k-forms behave? For one thing, we expect d to be *linear*—after all, a derivative is just the limit of a *difference*, and differences are certainly linear! What about the derivative of a wedge of two forms? Harkening back to good old-fashioned calculus, here's a picture that explains the typical product rule $\frac{\partial}{\partial x}(f(x)g(x)) = f'(x)g(x) + f(x)g'(x)$:



The dark region represents the value of fg at x; the light blue region represents the *change* in this value as we move x some small distance h. As h gets smaller and smaller, the contribution of the upper-right quadrant becomes negligible and we can write the derivative as the change in

f times g plus the change in g times f. (Can you make this argument more rigorous?) Since a k-form also measures a (signed) volume, this intuition also carries over to the exterior derivative of a wedge product. In particular, if α is a k-form then d obeys the rule

$$d(\alpha \wedge \beta) = d\alpha \wedge \beta + (-1)^k \alpha \wedge d\beta.$$

which says that the rate of change of the overall volume can be expressed in terms of changes in the constituent volumes, exactly as in the picture above.

4.6.5. Exterior Derivative of 1-Forms. To be a little more concrete, let's see what happens when we differentiate a 1-form on $\mathbb{R}^3$. Working things out in coordinates turns out to be a total mess, but in the end you may be pleasantly surprised with the simplicity of the outcome! (Later on we'll see that these ideas can also be expressed quite nicely *without* coordinates using *Stokes' theorem*, which paves the way to differentiation in the discrete setting.) Applying the linearity of d, we have

$$\begin{aligned} d\alpha &= d(\alpha_1 dx^1 + \alpha_2 dx^2 + \alpha_3 dx^3) \\ &= d(\alpha_1 dx^1) + d(\alpha_2 dx^2) + d(\alpha_3 dx^3). \end{aligned}$$

Each term $\alpha_j dx^j$ can really be thought of a wedge product $\alpha_j \wedge dx^j$ between a 0-form α_j and the corresponding basis 1-form dx^j. Applying the exterior derivative to one of these terms we get

$$d(\alpha_j \wedge dx^j) = (d\alpha_j) \wedge dx^j + \alpha_j \wedge \underbrace{(ddx^j)}_{=0} = \frac{\partial \alpha_j}{\partial x^i} dx^i \wedge dx^j.$$

To keep things short we used the Einstein summation convention here, but let's really write out all the terms:

$$\begin{aligned} d\alpha = &\tfrac{\partial \alpha_1}{\partial x^1} dx^1 \wedge dx^1 + \tfrac{\partial \alpha_1}{\partial x^2} dx^2 \wedge dx^1 + \tfrac{\partial \alpha_1}{\partial x^3} dx^3 \wedge dx^1 + \\ &\tfrac{\partial \alpha_2}{\partial x^1} dx^1 \wedge dx^2 + \tfrac{\partial \alpha_2}{\partial x^2} dx^2 \wedge dx^2 + \tfrac{\partial \alpha_2}{\partial x^3} dx^3 \wedge dx^2 + \\ &\tfrac{\partial \alpha_3}{\partial x^1} dx^1 \wedge dx^3 + \tfrac{\partial \alpha_3}{\partial x^2} dx^2 \wedge dx^3 + \tfrac{\partial \alpha_3}{\partial x^3} dx^3 \wedge dx^3. \end{aligned}$$

Using the fact that $\alpha \wedge \beta = -\beta \wedge \alpha$, we get a much simpler expression

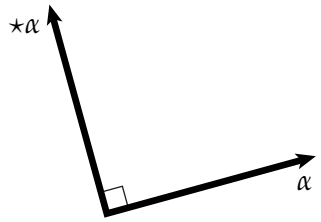
$$\begin{aligned} d\alpha = &(\tfrac{\partial \alpha_3}{\partial x^2} - \tfrac{\partial \alpha_2}{\partial x^3}) dx^2 \wedge dx^3 + \\ &(\tfrac{\partial \alpha_1}{\partial x^3} - \tfrac{\partial \alpha_3}{\partial x^1}) dx^3 \wedge dx^1 + \\ &(\tfrac{\partial \alpha_2}{\partial x^1} - \tfrac{\partial \alpha_1}{\partial x^2}) dx^1 \wedge dx^2. \end{aligned}$$

Does this expression look familiar? If you look again at our review of vector derivatives, you'll recognize that $d\alpha$ basically looks like the *curl* of $\alpha^\sharp$, except that it's expressed as a 2-form instead of a vector field. Also remember (from our discussion of Hodge duality) that a 2-form and a 1-form are not so different here—geometrically they both specify some direction in $\mathbb{R}^3$. Therefore, we can express the curl of any vector field X as

$$\nabla \times X = \left(\star dX^\flat \right)^\sharp.$$

It's worth stepping through the sequence of operations here to check that everything makes sense: $\flat$ converts the vector field X into a 1-form $X^\flat$; d computes something that looks like the curl, but expressed as a 2-form $dX^\flat$; $\star$ turns this 2-form into a 1-form $\star dX^\flat$; and finally $\sharp$ converts this 1-form back into the vector field $\left(\star dX^\flat\right)^\sharp$. The take-home message here, though, is that *the exterior derivative of a 1-form looks like the curl of a vector field*.

So far we know how to express the gradient and the curl using d. What about our other favorite vector derivative, the divergence? Instead of grinding through another tedious derivation, let's make a simple geometric observation: in $\mathbb{R}^2$ at least, we can determine the divergence of a vector field by rotating it by 90 degrees and computing its curl (consider the example we saw earlier). Moreover, in $\mathbb{R}^2$ the Hodge star on 1-forms represents a rotation by 90 degrees, since it identifies any *line* with the direction orthogonal to that line:



Therefore, we might suspect that divergence can be computed by first applying the Hodge star, then applying the exterior derivative:

$$\nabla \cdot X = \star d \star X^\flat.$$

The leftmost Hodge star accounts for the fact that $d \star X^\flat$ is an n-form instead of a 0-form—in vector calculus divergence is viewed as a scalar quantity. Does this definition really work? Let's give it a try in coordinates on $\mathbb{R}^3$. First, we have

$$\begin{aligned} \star X^\flat &= \star(X_1 dx^1 + X_2 dx^2 + X_3 dx^3) \\ &= X_1 dx^2 \wedge dx^3 + X_2 dx^3 \wedge dx^1 + X_3 dx^1 \wedge dx^2. \end{aligned}$$

Differentiating we get

$$\begin{aligned} d \star X^\flat = \; & \tfrac{\partial X_1}{\partial x^1} dx^1 \wedge dx^2 \wedge dx^3 + \\ & \tfrac{\partial X_2}{\partial x^2} dx^2 \wedge dx^3 \wedge dx^1 + \\ & \tfrac{\partial X_3}{\partial x^3} dx^3 \wedge dx^1 \wedge dx^2, \end{aligned}$$

but of course we can rearrange these wedge products to simply

$$d \star X^\flat = \left(\frac{\partial X_1}{\partial x^1} + \frac{\partial X_2}{\partial x^2} + \frac{\partial X_3}{\partial x^3} \right) dx^1 \wedge dx^2 \wedge dx^3.$$

A final application of the Hodge star gives us the divergence

$$\star d \star X^\flat = \frac{\partial X^1}{\partial x^1} + \frac{\partial X^2}{\partial x^2} + \frac{\partial X^3}{\partial x^3}$$

as desired.

In summary, for any scalar field ϕ and vector field X we have

$$\begin{aligned} \nabla\phi &= (d\phi)^\sharp \\ \nabla \times X &= \left(\star d X^\flat\right)^\sharp \\ \nabla \cdot X &= \star d \star X^\flat \end{aligned}$$

One cute thing to notice here is that (in $\mathbb{R}^3$) grad, curl, and div are just d applied to a $0-$, $1-$ and $2-$ form, respectively.

4.6.6. The Laplacian. Another key differential operator from vector calculus is the scalar *Laplacian* which (confusingly!) is often denoted by Δ or ∇^2, and is defined as

$$\Delta := \nabla \cdot \nabla,$$

i.e., the divergence of the gradient. Although the Laplacian may seem like yet another in a long list of derivatives, it deserves your utmost respect: the Laplacian is central to fundamental physical laws (any diffusion process and all forms of wave propagation, including the Schrödinger equation); its eigenvalues capture *almost* everything there is to know about a given piece of geometry (can you hear the shape of a drum?). Heavy tomes and entire lives have been devoted to the Laplacian, and in the discrete setting we'll see that this one simple operator can be applied to a diverse array of tasks (surface parameterization, surface smoothing, vector field design and decomposition, distance computation, fluid simulation... you name it, we got it!).

Fortunately, now that we know how to write div, grad and curl using exterior calculus, expressing the scalar Laplacian is straightforward: $\Delta = \star d \star d$. More generally, the *k-form Laplacian* is given by

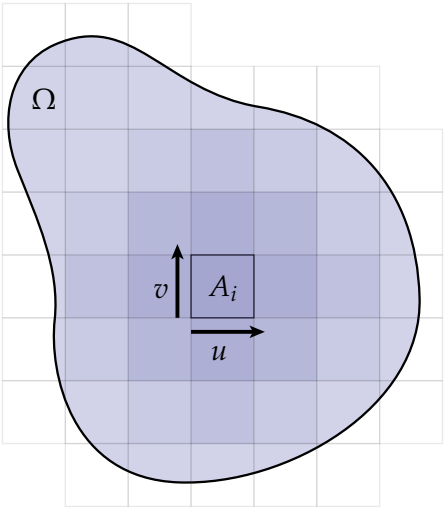
$$\Delta := \star d \star d + d \star d \star .$$

The name "Laplace-*Beltrami*" is used merely to indicate that the domain may have some amount of curvature (encapsulated by the Hodge star). Some people like to define the operator $\delta := \star d \star$, called the *codifferential*, and write the Laplacian as $\Delta = \delta d + d\delta$.

One question you might ask is: why is the Laplacian for 0-forms different from the general k-form Laplacian? Actually, it's not—consider what happens when we apply the term $d \star d \star$ to a 0-form ϕ: $\star\phi$ is an n-form, and so $d \star \phi$ must be an $(n+1)$-form. But there are no $(n+1)$-forms on an n-dimensional space! So this term is often omitted when writing the scalar Laplacian.

4.7. Integration and Stokes' Theorem

In the previous section we talked about how to *differentiate* k-forms using the exterior derivative d. We'd also like some way to *integrate* forms. Actually, there's surprisingly little to say about integration given the setup we already have. Suppose we want to compute the total area A_Ω of a region Ω in the plane:



If you remember back to calculus class, the basic idea was to break up the domain into a bunch of little pieces that are easy to measure (like squares) and add up their areas:

$$A_\Omega \approx \sum_i A_i.$$

As these squares get smaller and smaller we get a better and better approximation, ultimately achieving the true area

$$A_\Omega = \int_\Omega dA.$$

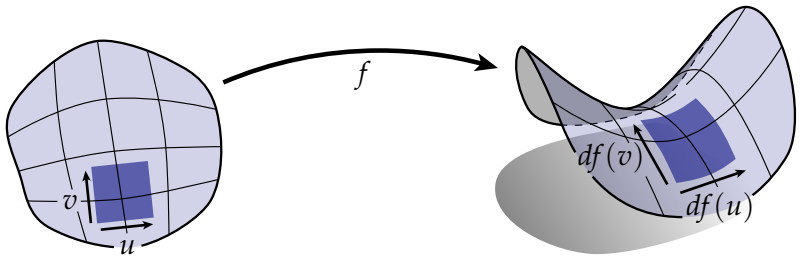
Alternatively, we could write the individual areas using differential forms—in particular, $A_i = dx^1 \wedge dx^2(u, v)$. Therefore, the area element dA is really nothing more than the standard volume form $dx^1 \wedge dx^2$ on $\mathbb{R}^2$. (Not too surprising, since the whole point of k-forms was to measure volume!)

To make things more interesting, let's say that the contribution of each little square is weighted by some scalar function ϕ. In this case we get the quantity

$$\int_\Omega \phi \, dA = \int_\Omega \phi \, dx^1 \wedge dx^2.$$

Again the integrand $\phi \, dx^1 \wedge dx^2$ can be thought of as a 2-form. In other words, you've been working with differential forms your whole life, even if you didn't realize it! More generally, integrands on an n-dimensional space are always n-forms, since we need to "plug in" n orthogonal vectors representing the local volume. For now, however, looking at surfaces (i.e., 2-manifolds) will give us all the intuition we need.

4.7.1. Integration on Surfaces.



If you think back to our discussion of the Hodge star, you'll remember the *volume form*

$$\omega = \sqrt{\det(g)}dx^1 \wedge dx^2,$$

which measures the area of little parallelograms on our surface. The factor $\sqrt{\det(g)}$ reminds us that we can't simply measure the volume in the domain M—we also have to take into account any "stretching" induced by the map $f : M \to \mathbb{R}^2$. Of course, when we integrate a function on a surface, we should also take this stretching into account. For instance, to integrate a function $\phi : M \to \mathbb{R}$, we would write

$$\int_\Omega \phi\omega = \int_\Omega \phi\sqrt{\det(g)}\, dx^1 \wedge dx^2.$$

In the case of a *conformal* parameterization things become even simpler—since $\sqrt{\det(g)} = a$ we have just

$$\int_\Omega \phi a\, dx^1 \wedge dx^2,$$

where $a : M \to \mathbb{R}$ is the scaling factor. In other words, we scale the value of ϕ up or down depending on the amount by which the surface locally "inflates" or "deflates." In fact, this whole story gives a nice geometric interpretation to good old-fashioned integrals: you can imagine that $\int_\Omega \phi\, dA$ represents the area of some suitably deformed version of the initially planar region Ω.

4.7.2. Stokes' Theorem. The main reason for studying integration on manifolds is to take advantage of the world's most powerful tool: *Stokes' theorem*. Without further ado, Stokes' theorem says that

$$\int_\Omega d\alpha = \int_{\partial\Omega} \alpha,$$

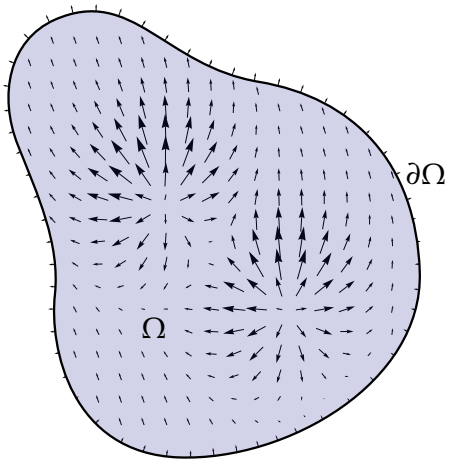
where α is any $n-1$-form on an n-dimensional domain Ω. In other words, integrating a differential form over the boundary of a manifold is the same as integrating its derivative over the entire domain.

If this trick sounds familiar to you, it's probably because you've seen it time and again in different contexts and under different names: the *divergence theorem*, *Green's theorem*, *the fundamental theorem of calculus*, *Cauchy's integral formula*, etc. Picking apart these special cases will really help us understand the more general meaning of Stokes' theorem.

4.7.3. Divergence Theorem. Let's start with the divergence theorem from vector calculus, which says that

$$\int_\Omega \nabla \cdot X\, dA = \int_{\partial\Omega} n \cdot X\, d\ell,$$

where X is a vector field on Ω and n represents the unit normal field along the boundary of Ω. A better name for this theorem might have been the "what goes in must come out theorem", because if you think about X as the flow of water throughout the domain Ω then it's clear that the amount of water being pumped into Ω (via pipes in the ground) must be the same as the amount flowing out of its boundary at any moment in time:



Let's try writing this theorem using exterior calculus. First, remember that we can write the divergence of X as $\nabla \cdot X = \star d \star X^\flat$. It's a bit harder to see how to write the right-hand side of the divergence theorem, but think about what integration does here: it takes tangents to the boundary and sticks them into a 1-form. For instance, $\int_{\partial\Omega} X^\flat$ "adds up" the *tangential* components of X. To get the *normal* component we could rotate $X^\flat$ by a quarter turn, which conveniently enough is achieved by hitting it with the Hodge star. Overall we get

$$\int_\Omega d \star X^\flat = \int_{\partial\Omega} \star X^\flat,$$

which, as promised, is just a special case of Stokes' theorem. Alternatively, we can use Stokes' theorem to provide a more geometric interpretation of the divergence operator itself: when integrated over *any* region Ω—no matter how small—the divergence operator gives the total flux through the region boundary. In the discrete case we'll see that this boundary flux interpretation is the *only* notion of divergence—in other words, there's no concept of divergence at a single point.

By the way, why does $d \star X^\flat$ appear on the left-hand side instead of $\star d \star X^\flat$? The reason is that $\star d \star X^\flat$ is a 0-form, so we have to hit it with another Hodge star to turn it into an object that measures areas (i.e., a 2-form). Applying this transformation is no different from appending dA to $\nabla \cdot X$—we're specifying how volume should be measured on our domain.

EXERCISE 4.2

Show that Stokes' theorem also implies *Green's theorem*, which says that

$$\int_\Omega \nabla \times X \, dA = \int_{\partial\Omega} t \cdot X \, d\ell,$$

where Ω is a region in the plane and t is a continuous unit vector field tangent to its boundary $\partial\Omega$.

4.7.4. Fundamental Theorem of Calculus. The fundamental theorem of calculus is in fact *so* fundamental that you may not even remember what it is. It basically says that for a real-valued function $\phi : \mathbb{R} \to \mathbb{R}$ on the real line

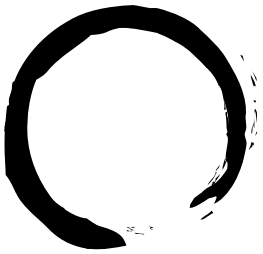
$$\int_a^b \frac{\partial \phi}{\partial x} dx = \phi(b) - \phi(a).$$

In other words, the total change over an interval $[a, b]$ is (as you might expect) how much you end up with minus how much you started with. But soft, behold! All we've done is written Stokes' theorem once again:

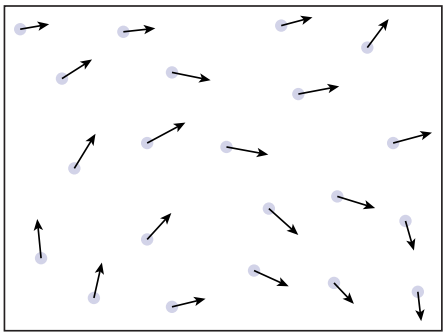
$$\int_{[a,b]} d\phi = \int_{\partial[a,b]} \phi,$$

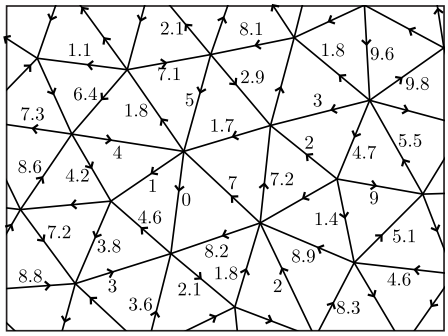
since the boundary of the interval $[a, b]$ consists only of the two endpoints a and b.

Hopefully these two examples give you a good feel for what Stokes' theorem says. In the end, it reads almost like a Zen kōan: what happens on the outside is purely a function of the change within. (Perhaps it is Stokes' that deserves the name, "fundamental theorem of calculus!")



4.8. Discrete Exterior Calculus





So far we've been exploring exterior calculus purely in the smooth setting. Unfortunately this theory was developed by some old-timers who did not know anything about computers, hence it cannot be used directly by machines that store only a finite amount of information. For instance, if we have a smooth vector field or a smooth 1-form we can't possibly store the direction of every little "arrow" at each point—there are far too many of them! Instead, we need to keep track of a discrete (or really, *finite*) number of pieces of information that capture the essential behavior of the objects we're working with; we call this scheme *discrete exterior calculus* (or *DEC* for short). The big secret about DEC is that it's literally *nothing more* than the good-old fashioned (continuous) exterior calculus we've been learning about, except that we integrate differential forms over elements of our mesh.

4.8.1. Discrete Differential Forms. One way to encode a 1-form might be to store a finite collection of "arrows" associated with some subset of points. Instead, we're going to do something a bit different: we're going to *integrate* our 1-form over each edge of a mesh, and store the resulting *numbers* (remember that the integral of an n-form always spits out a single number) on the corresponding edges. In other words, if α is a 1-form and e is an edge, then we'll associate the number

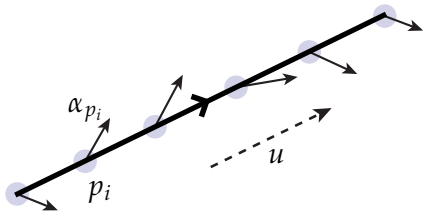
$$\hat{\alpha}_e := \int_e \alpha$$

with e, where the ($\hat{}$) is meant to suggest a *discrete* quantity (not to be confused with a unit-length vector).

Does this procedure seem a bit abstract to you? It shouldn't! Think about what this integral represents: it tells us how strongly the 1-form α "flows along" the edge e *on average*. More specifically, remember how integration of a 1-form works: at each point along the edge we take the vector *tangent* to the edge, stick it into the 1-form α, and sum up the resulting values—each value tells us something about how well α "lines up" with the direction of the edge. For instance, we could approximate the integral via the sum

$$\int_e \alpha \approx |e| \left(\frac{1}{N} \sum_{i=1}^{N} \alpha_{p_i}(u) \right),$$

where $|e|$ denotes the length of the edge, $\{p_i\}$ is a sequence of points along the edge, and $u := e/|e|$ is a unit vector tangent to the edge:



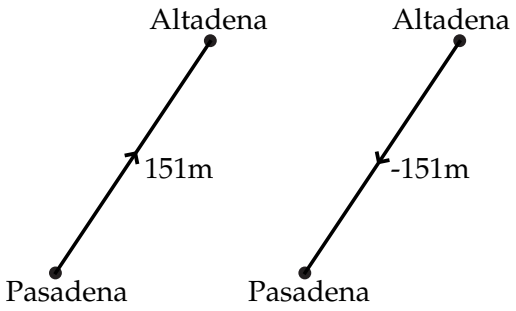
Of course, this quantity tells us *absolutely nothing* about the strength of the "flow" *orthogonal* to the edge: it could be zero, it could be enormous! We don't really know, because we didn't take any measurements along the orthogonal direction. However, the hope is that some of this information will still be captured by nearby edges (which are most likely not parallel to e).

More generally, a k-form that has been integrated over each k-dimensional cell (edges in 1D, faces in 2D, etc.) is called a *discrete differential k-form*. (If you ever find the distinction confusing, you might find it helpful to substitute the word "integrated" for the word "discrete.") In practice, however, not every discrete differential form has to originate from a continuous one—for instance, a bunch of arbitrary values assigned to each edge of a mesh is a perfectly good discrete 1-form.

4.8.2. Orientation. One thing you may have noticed in all of our illustrations so far is that each edge is marked with a little arrow. Why? Well, one thing to remember is that *direction* matters when you integrate. For instance, the fundamental theorem of calculus (and common sense) tells us that the total change as you go from a to b is the opposite of the total change as you go from b to a:

$$\int_a^b \frac{\partial \phi}{\partial x} dx = \phi(b) - \phi(a) = -(\phi(a) - \phi(b)) = -\int_b^a \frac{\partial \phi}{\partial x} dx.$$

Said in a much less fancy way: the elevation gain as you go from Pasadena to Altadena is 151 meters, so the elevation "gain" in the other direction must be -151 meters! Just keeping track of the number 151 does you little good—you have to say what that quantity represents.



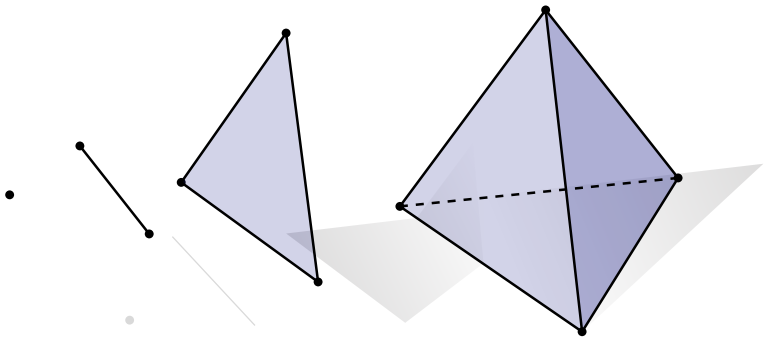
Therefore, when we store a discrete differential form it's not enough to just store a number: we also have to specify a canonical *orientation* for each element of our mesh, corresponding to the orientation we used during integration. For an edge we've already seen that we can think about orientation as a little arrow pointing from one vertex to another—we could also just think of an edge as an *ordered* pair (i, j), meaning that we always integrate from i to j.

More generally, suppose that each element of our mesh is an *oriented k-simplex* σ, i.e., a collection of $k+1$ vertices $p_i \in \mathbb{R}^n$ given in some fixed order $(p_1, \ldots, p_{k+1})$. The geometry associated with σ

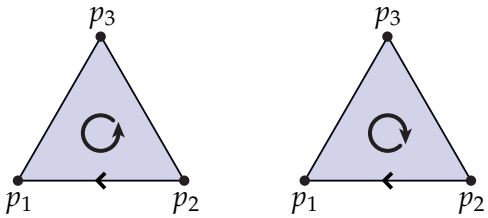
is the *convex combination* of these points:

$$\left\{ \sum_{i=1}^{k+1} \lambda_i p_i \,\middle|\, \sum_{i=1}^{k+1} \lambda_i = 1, \lambda_i \geq 0 \right\} \subset \mathbb{R}^n$$

(Convince yourself that a 0-simplex is a vertex, a 1-simplex is an edge, a 2-simplex is a triangle, and a 3-simplex is a tetrahedron.)



Two oriented k-simplices have the same orientation if and only if the vertices of one are an *even* permutation of the vertices of another. For instance, the triangles (p_1, p_2, p_3) and (p_2, p_3, p_1) have the same orientation; (p_1, p_2, p_3) and (p_2, p_1, p_3) have opposite orientation.



If a simplex σ_1 is a (not necessarily proper) subset of another simplex σ_2, then we say that σ_1 is a *face* of σ_2. For instance, every vertex, edge, and triangle of a tetrahedron σ is a face of σ; as is σ itself! Moreover, the orientation of a simplex *agrees with* the orientation of one of its faces as long as we see an even permutation on the shared vertices. For instance, the orientations of the edge (p_2, p_1) and the triangle (p_1, p_3, p_2) agree. Geometrically all we're saying is that the two "point" in the same direction (as depicted above, right). To keep yourself sane while working with meshes, the most important thing is to *pick an orientation and stick with it*!

So in general, how do we integrate a k-form over an oriented k-simplex? Remember that a k-form is going to "eat" k vectors at each point and spit out a number—a good canonical choice is to take the ordered collection of edge vectors $(p_2 - p_1, \ldots, p_{k+1} - p_1)$ and orthonormalize them (using, say the Gram-Schmidt algorithm) to get vectors $(u_1, \ldots, u_k)$. This way the sign of the integrand changes whenever the orientation changes. Numerically, we can then approximate the integral via a sum

$$\int_\sigma \alpha \approx \frac{|\sigma|}{N} \sum_{i=1}^{N} \alpha_{p_i}(u_1, \ldots, u_k)$$

where $\{p_i\}$ is a (usually carefully-chosen) collection of sample points. (Can you see why the orientation of σ affects the sign of the integrand?) Sounds like a lot of work, but in practice one rarely constructs discrete differential forms via integration: more often, discrete forms are constructed via input data that is already discrete (e.g., vertex positions in a triangle mesh).

By the way, what's a discrete 0-form? Give up? Well, it must be a 0-form (i.e., a function) that's been integrated over every 0-simplex v_i (i.e., vertex) of a mesh:

$$\hat{\phi}_i = \int_{v_i} \phi$$

By convention, the integral of a function over a zero-dimensional set is simply the value of the function at that point: $\hat{\phi}_i = \phi(v_i)$. In other words, in the case of 0-forms there is no difference between storing point samples and storing integrated quantities: the two coincide. Note that the orientation of a 0-simplex is always *positive*, since the identity map on one vertex is an even permutation.

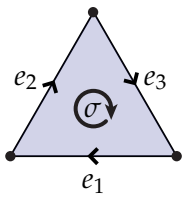
It's also important to remember that differential forms don't have to be *real*-valued. For instance, we can think of a map $f : M \to \mathbb{R}^3$ that encodes the geometry of a surface as an $\mathbb{R}^3$-valued 0-form; its differential df is then an $\mathbb{R}^3$-valued 1-form, etc. Likewise, when we say that a discrete differential form is a *number* stored on every mesh element, the word "number" is used in a fairly loose sense: a number could be a real value, a vector, a complex number, a quaternion, etc. For instance, the collection of (x, y, z) vertex coordinates of a mesh can be viewed as an $\mathbb{R}^3$-valued discrete 0-form (namely, one that discretizes the map f). The only requirement, of course, is that we store the same *type* of number on each mesh element.

4.8.3. The Discrete Exterior Derivative. One of the main advantages of working with integrated (i.e., "discrete") differential forms instead of point samples is that we can easily take advantage of Stokes' theorem. Remember that Stokes' theorem says

$$\int_\Omega d\alpha = \int_{\partial\Omega} \alpha,$$

for any k-form α and $k+1$-dimensional domain Ω. In other words, we can integrate the derivative of a differential form as long as we know its integral along the boundary. But that's *exactly* the kind of information encoded by a discrete differential form! For instance, if $\hat{\alpha}$ is a discrete 1-form stored on the three edges of a triangle σ, then we have

$$\int_\sigma d\alpha = \int_{\partial\sigma} \alpha = \sum_{i=1}^{3} \int_{e_i} \alpha = \sum_{i=1}^{3} \hat{\alpha}_i.$$



In other words, we can *exactly* evaluate the integral on the left by just adding up three numbers. Pretty cool! In fact, the thing on the left is *also* a discrete differential form: it's the 2-form $d\alpha$ integrated over the only triangle in our mesh. So for convenience, we'll call this guy "$\hat{d}\hat{\alpha}$", and we'll call the operation $\hat{d}$ the *discrete exterior derivative*. (In the future we will drop the hats from our

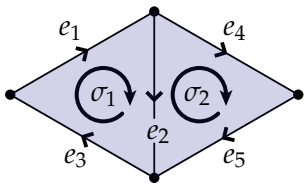
notation when the meaning is clear from context.) In other words, the discrete exterior derivative takes a k-form that has *already been integrated* over each k-simplex and applies Stokes' theorem to get the integral of the derivative over each $k+1$-simplex.

In practice (i.e., in code) you can see how this operation might be implemented by simply taking local sums over the appropriate mesh elements. However, in the example above we made life particularly easy on ourselves by giving each edge an orientation that agrees with the orientation of the triangle. Unfortunately assigning a consistent orientation to every simplex is not always possible, and in general we need to be more careful about *sign* when adding up our piecewise integrals. For instance, in the example below we'd have

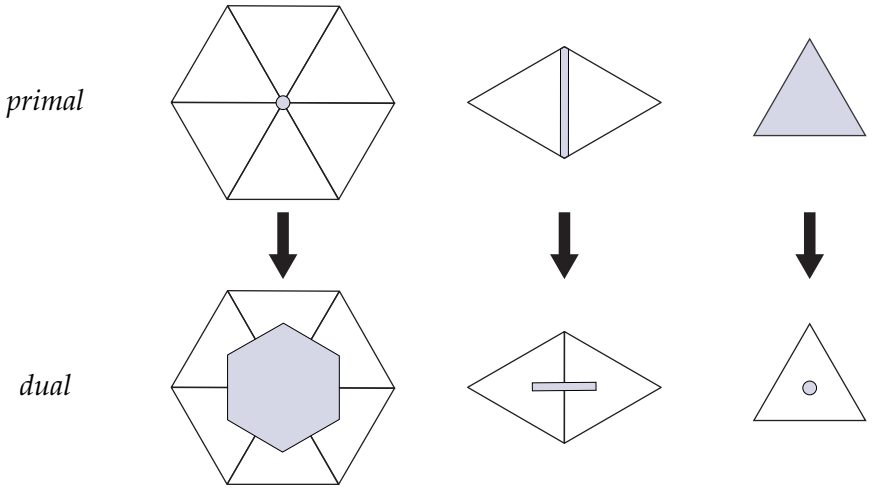
$$(\hat{d}\hat{\alpha})_1 = \hat{\alpha}_1 + \hat{\alpha}_2 + \hat{\alpha}_3$$

and

$$(\hat{d}\hat{\alpha})_2 = \hat{\alpha}_4 + \hat{\alpha}_5 - \hat{\alpha}_2.$$



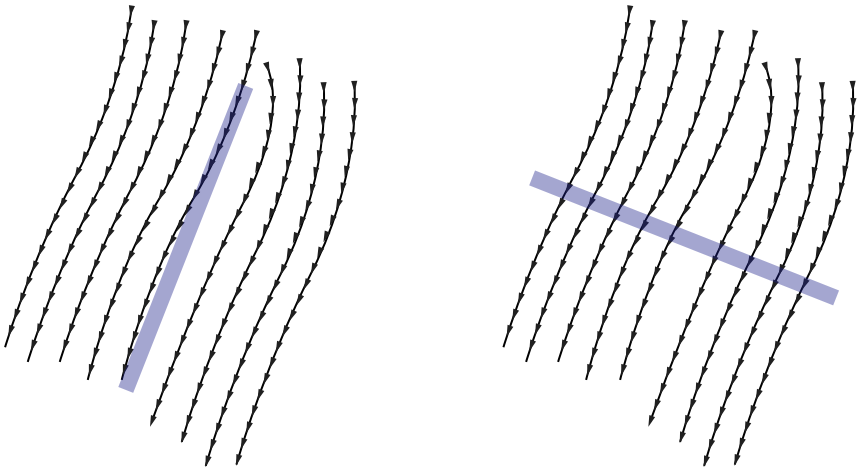
4.8.4. Discrete Hodge Star.



As hinted at above, a discrete k-form captures the behavior of a continuous k-form along k directions, but not along the remaining $n-k$ directions—for instance, a discrete 1-form in 2D captures the flow along edges but not in the orthogonal direction. If you paid close attention to the discussion of Hodge duality, this story starts to sound familiar! To capture Hodge duality in the discrete setting, we'll need to define a *dual mesh*. In general, the dual of an n-dimensional simplicial mesh identifies every k-simplex in the primal (i.e., original) mesh with a unique $(n-k)$-cell in the dual mesh. In a two-dimensional *simplicial* mesh, for instance, primal vertices are identified with dual faces, primal edges are identified with dual edges, and primal faces are identified with dual vertices. Note, however, that the dual cells are not always simplices! (See above.) A dual mesh is an *orthogonal dual* if primal and dual elements are contained in orthogonal linear subspaces. For

instance, on a planar triangle mesh a dual edge would make a right angle with the corresponding primal edge. For curved domains, we ask only that primal and dual elements be orthogonal *intrinsically*, *e.g.*, if one rigidly unfolds a pair of neighboring triangles into the plane, the primal and dual edges should again be orthogonal.

The fact that dual mesh elements are contained in orthogonal linear subspaces leads naturally to a notion of Hodge duality in the discrete setting. In particular, the *discrete Hodge dual* of a (discrete) k-form on the primal mesh is an $(n-k)$-form on the dual mesh. Similarly, the Hodge dual of an k-form on the dual mesh is an $(n-k)$-form on the primal mesh. Discrete forms on the primal mesh are called *primal forms* and discrete forms on the dual mesh are called *dual forms*. Given a discrete form $\hat{\alpha}$ (whether primal or dual), we'll write its Hodge dual as $\hat{\star}\hat{\alpha}$.



primal 1-form (circulation) dual 1-form (flux)

Unlike continuous forms, discrete primal and dual forms live in different places (so for instance, discrete primal k-forms and dual k-forms cannot be added to each other). In fact, primal and dual forms often have different physical interpretations. For instance, a primal 1-form might represent the total circulation along edges of the primal mesh, whereas in the same context a dual 1-form might represent the total flux through the corresponding dual edges (see illustration above).

Of course, these two quantities (flux and circulation) are closely related, and naturally leads into one definition for a discrete Hodge star called the *diagonal Hodge star*. Consider a primal k-form α. If $\hat{\alpha}_i$ is the value of $\hat{\alpha}$ on the k-simplex σ_i, then the diagonal Hodge star is defined by

$$\hat{\star}\hat{\alpha}_i = \frac{|\sigma_i^\star|}{|\sigma_i|}\hat{\alpha}_i$$

for all i, where $|\sigma|$ indicates the (unsigned) volume of σ (which by convention equals *one* for a vertex!) and $|\sigma^\star|$ is the volume of the corresponding dual cell. In other words, to compute the dual form we simply multiply the scalar value stored on each cell by the ratio of corresponding dual and primal volumes.

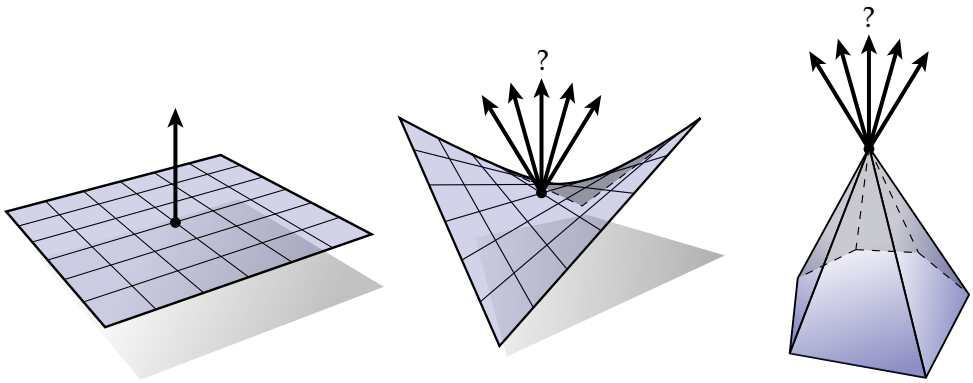
If we remember that a discrete form can be thought of as a continuous form integrated over each cell, this definition for the Hodge star makes perfect sense: the primal and dual quantities should have the same *density*, but we need to account for the fact that they are integrated over cells of different volume. We therefore normalize by a *ratio* of volumes when mapping between primal

and dual. This particular Hodge star is called *diagonal* since the ith element of the dual differential form depends only on the ith element of the primal differential form. It's not hard to see, then, that Hodge star taking dual forms to primal forms (the *dual Hodge star*) is the inverse of the one that takes primal to dual (the *primal Hodge star*).

4.8.5. That's All, Folks! Hey, wait a minute, what about our other operations, like the wedge product ($\wedge$)? These operations can certainly be defined in the discrete setting, but we won't go into detail here—the basic recipe is to integrate, integrate, integrate. Actually, even in *continuous* exterior calculus we omitted a couple operations like the *Lie derivative* ($\mathcal{L}_X$) and the *interior product* (i_α). Coming up with a complete discrete calculus where the whole cast of characters d, $\wedge$, $\star$, $\mathcal{L}_X$, i_α, *etc.*, plays well together is an active and ongoing area of research.

CHAPTER 5

Curvature of Discrete Surfaces

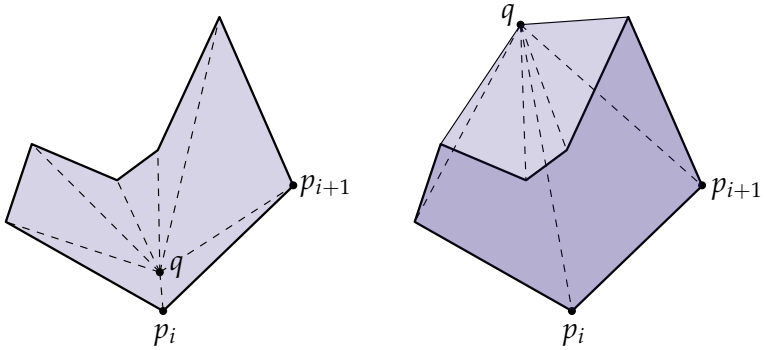


For a smooth surface in $\mathbb{R}^3$, the *normal* direction is easy to define: it is the unique direction orthogonal to all tangent vectors—in other words, it's the direction sticking "straight out" of the surface. For discrete surfaces the story is not so simple. If a mesh has *planar* faces (all vertices lie in a common plane) then of course the normal is well-defined: it is simply the normal of the plane. But if the polygon is *nonplanar*, or if we ask for the normal at a *vertex*, then it is not as clear how the normal should be defined.

In practice there are a number of different possibilities, which arise from different ways of looking at the smooth geometry. But before jumping in, let's establish a few basic geometric facts.

5.1. Vector Area

Here's a simple question: how do you compute the area of a polygon in the plane? Suppose our polygon has vertices $p_1, p_2, \ldots, p_n$. One way to compute the area is to stick another point q in the middle and sum up the areas of triangles q, p_i, p_{i+1} as done on the left:



A cute fact is that if we place q *anywhere* and sum up the *signed* triangle areas, we still recover the polygon area! (Signed area just means *negative* if our vertices are oriented clockwise; *positive* if they're counter-clockwise.) You can get an idea of why this happens just by looking at the picture: positive triangles that cover "too much" area get accounted for by negative triangles.

The proof is an application of Stokes' theorem—consider a different expression for the area A of a planar polygon P:

$$A = \int_P dx \wedge dy.$$

Noting that $dx \wedge dy = d(x \wedge dy) = -d(y \wedge dx)$, we can also express the area as

$$A = \frac{1}{2}\int_P d(x \wedge dy) - d(y \wedge dx) = \frac{1}{2}\int_{\partial P} x \wedge dy - y \wedge dx,$$

where we've applied Stokes' theorem in the final step to convert our integral over the entire surface into an integral over just the boundary. Now suppose that our polygon vertices have coordinates $p_i = (x_i, y_i)$. From here we can explicitly work out the boundary integral by summing up the integrals over each edge e_{ij}:

$$\int_{\partial P} x \wedge dy - y \wedge dx = \sum \int_{e_{ij}} x \wedge dy - y \wedge dx.$$

Since the coordinate functions x and y are *linear* along each edge (and their differentials dx and dy are therefore *constant*), we can write these integrals as

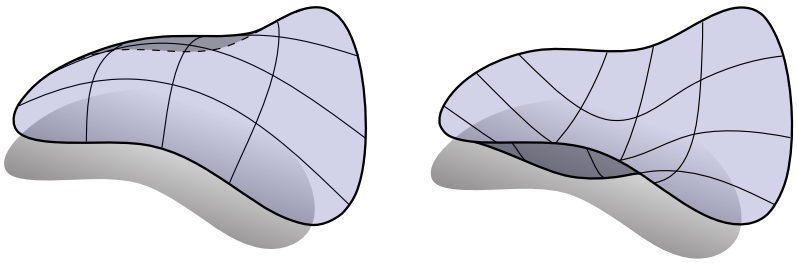
$$\begin{aligned}
\textstyle\sum \int_{e_{ij}} x \wedge dy - y \wedge dx &= \textstyle\sum \frac{x_i + x_j}{2}(y_j - y_i) - \frac{y_i + y_j}{2}(x_j - x_i) \\
&= \textstyle\frac{1}{2}\sum (p_i + p_j) \times (p_j - p_i) \\
&= \textstyle\frac{1}{2}\sum p_i \times p_j - p_i \times p_i - p_j \times p_j - p_j \times p_i \\
&= \textstyle\sum p_i \times p_j.
\end{aligned}$$

In short, we've shown that the area of a polygon can be written as simply

$$A = \frac{1}{2}\sum_i p_i \times p_j.$$

EXERCISE 5.1

Complete the proof by showing that for any point q the signed areas of triangles (q, p_i, p_{i+1}) sum to precisely the expression above.

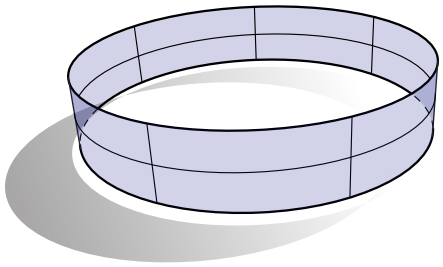


A more general version of the situation we just looked at with polygon areas is the *vector area* of a surface patch $f : M \to \mathbb{R}^3$, which is defined as the integral of the surface normal over the entire domain:

$$N_{\mathcal{V}} := \int_M N dA.$$

A very nice property of the vector area is that it depends only on the shape of the boundary ∂M (as you will demonstrate in the next exercise). As a result, two surfaces that look very different (such as the ones above) can still have the same vector area—the physical intuition here is that the vector area measures the total *flux* through the boundary curve.

For a *flat* region the normal is constant over the surface and we get just the usual area times the unit normal vector. Things get more interesting when the surface is not flat—for instance, the vector area of a circular band is *zero* since opposing normals cancel each-other out:

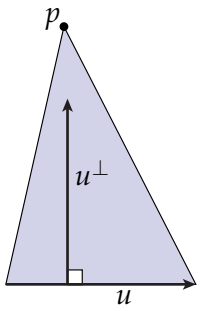


EXERCISE 5.2

Using Stokes' theorem, show that the vector area can be written as

$$N_{\mathcal{V}} = \frac{1}{2} \int_{\partial M} f \wedge df,$$

where the product of two vectors in $\mathbb{R}^3$ is given by the usual cross product $\times$.



Here's another fairly basic question: consider a triangle sitting in $\mathbb{R}^3$, and imagine that we're allowed to pull on one of its vertices p. What's the quickest way to increase its area A? In other words, what's the *gradient* of A with respect to p?

EXERCISE 5.3

Show that the area gradient is given by

$$\nabla_p A_\sigma = \frac{1}{2}\mathbf{u}^\perp$$

where $\mathbf{u}^\perp$ is the edge vector across from p rotated by an angle $\pi/2$ in the plane of the triangle (such that it points toward p).

You should require only a few *very simple* geometric arguments—there's no need to write things out in coordinates, etc.

5.2. Area Gradient

With these facts out of the way let's take a look at some different ways to define vertex normals. There are essentially only two definitions that arise naturally from the smooth picture: the *area gradient* and the *volume gradient*; we'll start with the former.

The area gradient asks, "which direction should we 'push' the surface in order to increase its total area A as quickly as possible?" Sliding all points tangentially along the surface clearly doesn't change anything: we just end up with the same surface. In fact, the only thing we can do to increase surface area is move the surface in the *normal* direction. The idea, then, is to *define* the vertex normal as the gradient of area with respect to a given vertex.

Since we already know how to express the area gradient for a single triangle σ, we can easily express the area gradient for the entire surface:

$$\nabla_p A = \sum_\sigma \nabla_p A_\sigma.$$

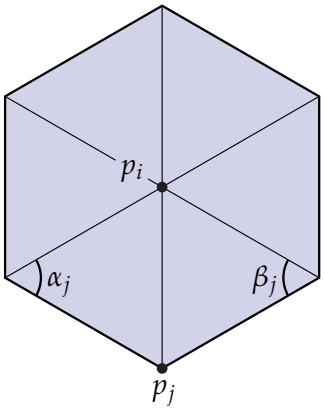
Of course, a given vertex p influences only the areas of the triangles touching p. So we can just sum up the area gradients over this small collection of triangles.

EXERCISE 5.4

Show that the gradient of surface area with respect to vertex p_i can be expressed as

$$\nabla_{p_i} A = \frac{1}{2}\sum_j (\cot\alpha_j + \cot\beta_j)(p_j - p_i)$$

where p_j is the coordinate of the jth neighbor of p_i and α_j and β_j are the angles across from edge (p_i, p_j).



5.2.1. Mean Curvature Vector. The expression for the area gradient derived in the last exercise shows up all over discrete differential geometry, and is often referred to as the *cotan formula*. Interestingly enough this same expression appears when taking a completely different approach to defining vertex normals, by way of the *mean curvature vector* HN. In particular, for a smooth surface $f : M \to \mathbb{R}^3$ we have

$$\Delta f = 2HN$$

where H is the mean curvature, N is the unit surface normal (which we'd like to compute), and Δ is the Laplace-Beltrami operator (see below). Therefore, another way to define vertex normals for a discrete surface is to simply apply a discrete Laplace operator to the vertex positions and normalize the resulting vector.

The question now becomes, "how do you discretize the Laplacian?" We'll take a closer look at this question in the future, but the remarkable fact is that the most straightforward discretization of Δ leads us right back to the cotan formula! In other words, the vertex normals we get from the mean curvature vector are precisely the same as the ones we get from the area gradient.

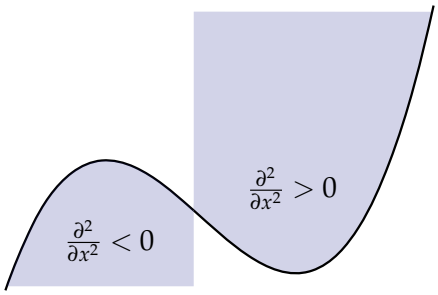
This whole story also helps us get better intuition for the Laplace-Beltrami operator Δ itself. Unfortunately, there's no really nice way to write Δ—the standard coordinate formula you'll find in a textbook on differential geometry is $\Delta\phi = \frac{1}{\sqrt{|g|}}\frac{\partial}{\partial x^i}(\sqrt{|g|}g^{ij}\frac{\partial}{\partial x^j}\phi)$, where g is the metric. However, this obfuscated expression provides little intuition about what Δ really does, and is damn-near useless when it comes to discretization since for a triangle mesh we *never* have a coordinate representation of g! Earlier, we saw that the (0-form) Laplacian can be expressed as $\Delta = \star d \star d$, which leads to a fairly straightforward discretization. But for now, we'll make use of another tool we learned about earlier: *conformal parameterization*. Remember that if f is a conformal map, then lengths on M and lengths on $f(M)$ are related by a positive scaling e^u. In other words, $|df(X)| = e^u|X|$ for some real-valued function u on M. Moreover, a conformal parameterization always exists—in other words, we don't have to make any special assumptions about our geometry in order to use conformal coordinates in proofs or other calculations. The reason conformal coordinates are useful when talking about Laplace-Beltrami is that we can write Δ as simply a rescaling of the standard Laplacian in the plane, i.e., as the sum of second partial

derivatives divided by the metric scaling factor e^{2u}:

$$\Delta\phi = \frac{d(d\phi(X))(X) + d(d\phi(Y))(Y)}{e^{2u}},$$

where X and Y are any pair of unit, orthogonal directions.

What's the geometric meaning here? Remember that for a good old-fashioned function $\phi : \mathbb{R} \to \mathbb{R}$ in 1D, second derivatives basically tell us about the *curvature* of a function, e.g., is it concave or convex?



Well, since Δ is a *sum* of second derivatives, it's no surprise that it tells us something about the *mean* curvature!

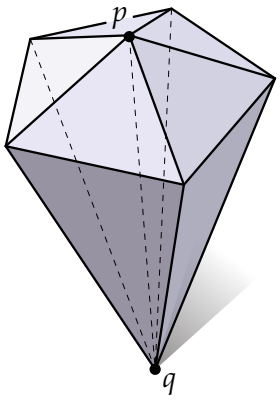
EXERCISE 5.5

Show that the relationship $\Delta f = 2HN$ holds.

5.3. Volume Gradient

An alternative way to come up with normals is to look at the *volume gradient*. Suppose that our surface encloses some region of space with total volume $\mathcal{V}$. As before, we know that sliding the surface along itself tangentially doesn't really change anything: we end up with the same surface, which encloses the same region of space. Therefore, the quickest way to increase $\mathcal{V}$ is to again move the surface in the normal direction. A somewhat surprising fact is that, in the discrete case, the volume gradient actually yields a *different* definition for vertex normals than the one we got from the area gradient. To express this gradient, we'll use three-dimensional versions of our "basic facts" from above.

First, much like we broke the area of a polygon into triangles, we're going to decompose the volume enclosed by our surface into a collection of tetrahedra. Each tetrahedron includes exactly one face of our discrete surface, along with a new point q. For instance, here's what the volume might look like in the vicinity of a vertex p:



Just as in the polygon case the location of q makes no difference, as long as we work with the *signed* volume of the tetrahedra. (Can you prove it?)

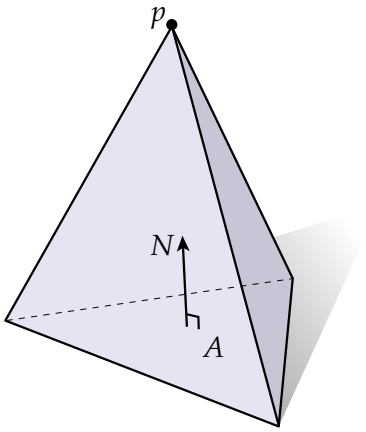
Next, what's the volume gradient for a single tetrahedron? One way to write the volume of a tet is as

$$\mathcal{V} = \frac{1}{3}Ah,$$

where A is the area of the base triangle and h is the height. Then using the same kind of geometric reasoning as in the triangle case, we know that

$$\nabla_p \mathcal{V} = \frac{1}{3}AN,$$

where N is the unit normal to the base.

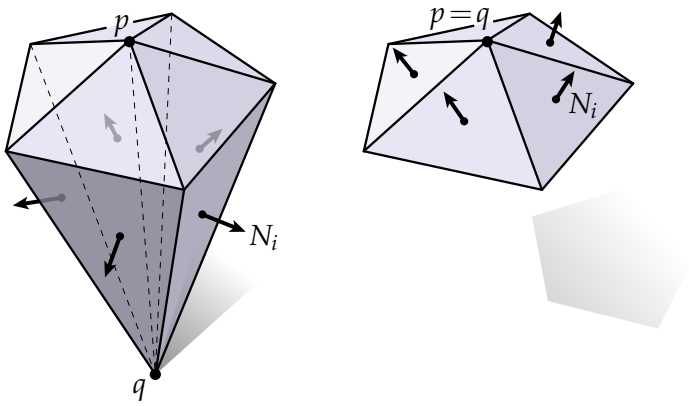


To express the gradient of the enclosed volume with respect to a given vertex p, we simply sum up the gradients for the tetrahedra containing p:

$$\nabla_p \mathcal{V} = \sum_i \mathcal{V}_i = \frac{1}{3}\sum_i A_i N_i.$$

At first glance this sum does not lead to a nice expression for $\Delta_p \mathcal{V}$—for instance, it uses the normals N_i of faces that have little to do with our surface geometry. However, remember that we can place q anywhere we please and still get the same expression for volume. In particular, if we put q directly

on top of p, then the N_i and A_i coincide with the normals and areas (respectively) of the faces containing p from our original surface:



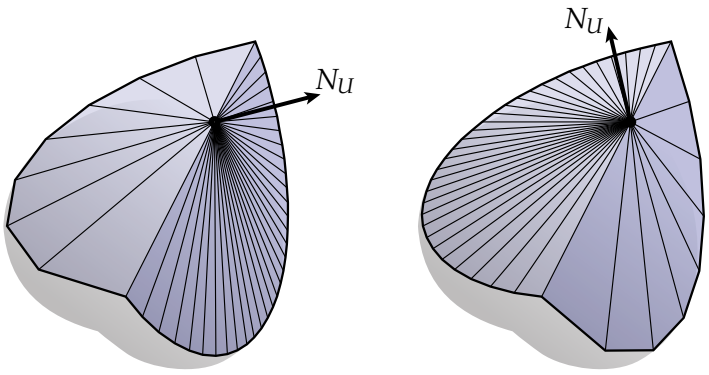
EXERCISE 5.6

Show that the volume gradient points in the same direction as the vector area N_V (i.e., show that they are the same up to a constant factor).

5.4. Other Definitions

So far we've only looked at definitions for vertex normals that arise from some smooth definition. This way of thinking captures the essential spirit of discrete differential geometry: relationships from the smooth setting should persist unperturbed in the discrete setting (e.g., $\Delta f = 2HN$ should be true independent of whether Δ, H, and N are smooth objects or discrete ones). Nonetheless, there are a number of common definitions for vertex normals that do not have a known origin in the smooth world. (Perhaps you can find one?)

5.4.1. Uniform Weighting.

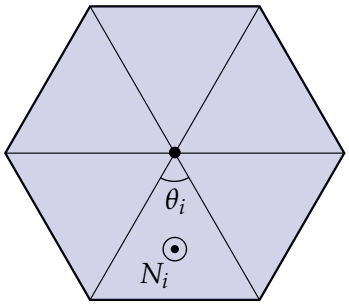


Perhaps the simplest way to get vertex normals is to just add up the neighboring face normals:

$$N_U := \sum_i N_i$$

The main drawback to this approach is that two different tessellations of the same geometry can produce very different vertex normals, as illustrated above.

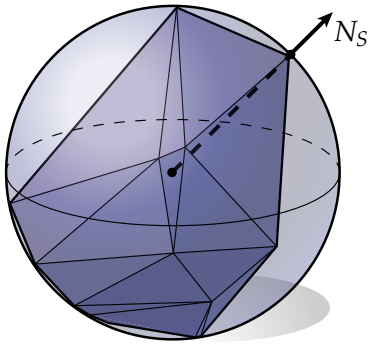
5.4.2. Tip-Angle Weights.



A simple way to reduce dependence on the tessellation is to weigh face normals by their corresponding *tip angles* θ, i.e., the interior angles incident on the vertex of interest:

$$N_\theta := \sum_i \theta_i N_i$$

5.4.3. Sphere-Inscribed Polytope.



Here's another interesting approach to vertex normals: consider the sphere S^2 consisting of all points unit distance from the origin in $\mathbb{R}^3$. A nice fact about the sphere is that the unit normal N at a point $x \in S^2$ is simply the point itself! I.e., $N(x) = x$. So if we start out with a polytope whose vertices all sit on the sphere, one reasonable way to define vertex normals is to simply use the vertex positions.

In fact, it's not too hard to show that the direction of the normal at a vertex p_i can be expressed purely in terms of the edge vectors $e_j = p_j - p_i$, where p_j are the immediate neighbors of p_i. In particular, we have

$$N_S = \frac{1}{c} \sum_{j=0}^{n-1} \frac{e_j \times e_{j+1}}{|e_j|^2 |e_{j+1}|^2}$$

where the constant $c \in \mathbb{R}$ can be ignored since we're only interested in the *direction* of the normal. (For a detailed derivation of this expression, see Max, *"Weights for Computing Vertex Normals from Facet Normals."*) Of course, since this expression depends only on the edge vectors it can be evaluated on any mesh (not just those inscribed in a sphere).

CODING 10. For the coding portion of this assignment, you will implement the vertex normals derived above, as well as the angle defect expression you derived when studying topological invariants of discrete surfaces. In particular, you should write methods that compute:

- the vertex normal N_U using uniform weights
- the vertex normal N_V using face area weights
- the vertex normal N_θ using tip angle weights
- the mean curvature normal Δf
- the sphere-inscribed normal N_S
- the Gaussian curvature K using tip angle defect
- the total Gaussian curvature of the entire mesh
- the Euler characteristic of the mesh

Once you've successfully implemented these methods, test them out on some meshes. Do you notice that some definitions work better than others? When? Why? Can you explain the behavior of the mean curvature normal? Does your total angle defect agree with the discrete Gauss-Bonnet theorem? (You can check, using the value you computed for the Euler characteristic.)

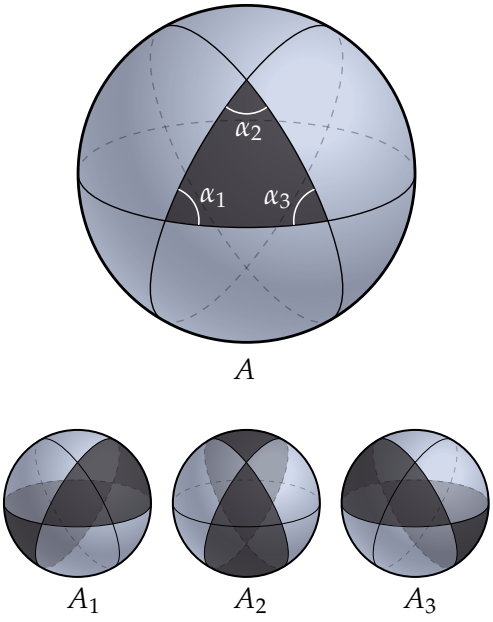
5.5. Gauss-Bonnet

EXERCISE 5.7

Area of a Spherical Triangle. Show that the area of a spherical triangle on the unit sphere with interior angles $\alpha_1, \alpha_2, \alpha_3$ is

$$A = \alpha_1 + \alpha_2 + \alpha_3 - \pi.$$

Hint: consider the areas A_1, A_2, A_3 of the three shaded regions (called "diangles") pictured below.



EXERCISE 5.8

Area of a Spherical Polygon. Show that the area of a spherical polygon with consecutive interior angles $\beta_1, \ldots, \beta_n$ is

$$A = (2 - n)\pi + \sum_{i=1}^{n} \beta_i.$$

Hint: use the expression for the area of a spherical triangle you just derived!

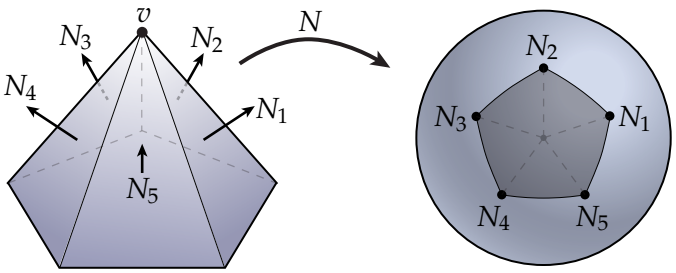
EXERCISE 5.9

Angle Defect. Recall that for a discrete planar curve we can define the curvature at a vertex as the distance on the unit circle between the two adjacent normals. For a discrete *surface* we can define *discrete Gaussian curvature* at a vertex v as the *area* on the unit sphere bounded by a spherical polygon whose vertices are the unit normals of the faces around v. Show that this area is equal to

the *angle defect*

$$d(v) = 2\pi - \sum_{f \in F_v} \angle_f(v)$$

where F_v is the set of faces containing v and $\angle_f(v)$ is the interior angle of the face f at vertex v. *Hint: consider planes that contain two consecutive normals and their intersection with the unit sphere.*



EXERCISE 5.10

Discrete Gauss-Bonnet Theorem. Consider a (connected, orientable) simplicial surface K with finitely many vertices V, edges E and faces F. Show that a discrete analog of the Gauss-Bonnet theorem holds for simplicial surfaces, namely

$$\sum_{v \in V} d(v) = 2\pi\chi$$

where $\chi = |V| - |E| + |F|$ is the *Euler characteristic* of the surface.

This fact perfectly mirrors the smooth Gauss-Bonnet theorem, namely that the total Gaussian curvature K is always a constant multiple of the Euler characteristic:

$$\int_M K \, dA = 2\pi\chi.$$

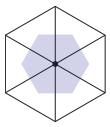
5.6. Numerical Tests and Convergence

In the coding part of this assignment, we will verify that the discrete Gauss-Bonnet theorem derived in the previous exercise really produces an exact value for the total Gaussian curvature in practice; we will also perform a numerical comparison between angle defect and other ways of approximating pointwise Gaussian curvature. The basic idea in all of our coding exercises will be to re-implement key methods within a larger mesh processing framework. We will describe the relevant inputs and outputs (*i.e.*, the *interface*) and ask you to fill in the rest. Methods are typically referred to by the class name, followed by two colons, followed by the method name. For example, `Mesh::read()` refers to a method called `read` in the class `Mesh`. The implementation of this method can be found in a file called `Mesh.cpp`; the routines you are asked to re-implement will typically be marked with a "TODO". When implementing a member function, it's important to realize that the necessary data is most often *not* passed to the method directly—instead, you will likely want to access member data from the parent class. For instance, when implementing methods

in the `Face` class, you can access one of the face's halfedges via the variable "`he`"; subsequently, one can access the root vertex of that half edge by writing `he->vertex` (and so on). For more information about commands available in our mesh processing framework, see the user guide included with the code distribution.

5.6.1. Total Curvature.

CODING 11. Implement the method `Vertex::angleDefect()`, which should return the angle defect discussed above, equal to 2π minus the sum of angles incident on the given vertex.



As we will discuss later on, the angle defect at a vertex actually describes the total Gaussian curvature in a neighborhood around the vertex—in particular, the boundary of this neighborhood can be drawn by connecting the consecutive midpoints of triangles incident on the vertex of interest. Therefore, if we want to visualize the Gaussian curvature itself, we must divide the angle defect by the area of the vertex neighborhood.

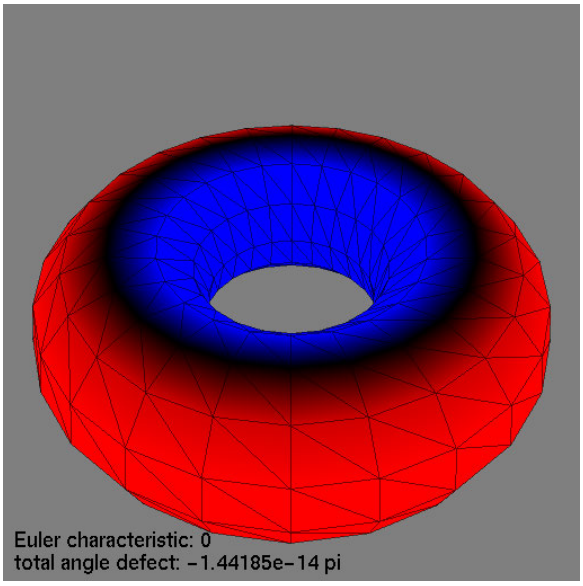
CODING 12. Implement the method `Face::area()`, which should return the triangle area.

CODING 13. Implement the method `Vertex::area()`, which should return one third the sum of the areas of all incident triangles.

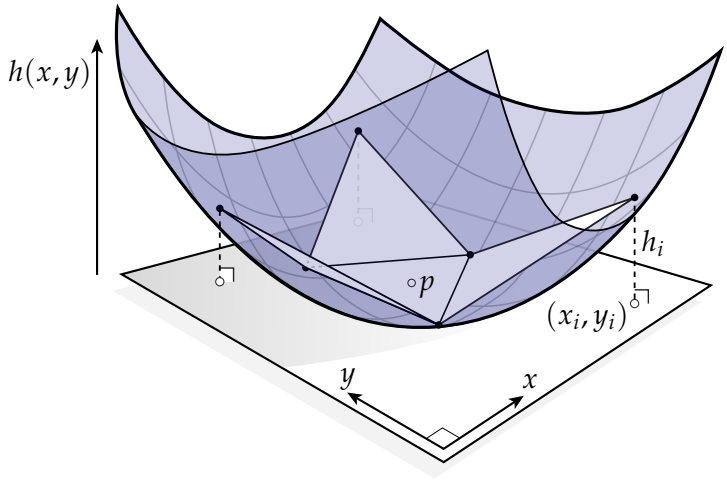
(These routines will automatically be used by the viewer; you do not have to call them explicitly.)

CODING 14. Implement the method `Mesh::totalGaussianCurvature()`, which returns the sum of the angle defect over all vertices.

With these methods implemented, you should now be able to compile and run the code. For each mesh in the `data` directory, you should verify that the total angle defect is equal to 2π times the Euler characteristic of the surface—both of these quantities will be displayed in the lower left corner of the viewer window, and the viewer will warn you (in red) if something is wrong. (Food for thought: why does your code produce a value *slightly* different from zero for the torus?) The angle defect itself can be visualized by pressing the 'a' key. Moreover, the 'd' key will deform the surface—press this key several times and verify that the total angle defect remains unchanged. For reference, the image below shows the correct appearance for the torus mesh; red values indicate positive curvature, blue values indicate negative curvature, and black indicates no curvature.



5.6.2. Pointwise Curvature Estimates. In the previous section, we verified that the total angle defect always gives us the correct value for the *total* Gaussian curvature. In this section we will look at how accurately angle defect approximates the Gaussian curvature at each point, comparing it with a more traditional estimate based on local *interpolation*. The basic idea behind interpolation is to find some nice, smooth function that exactly passes through a given collection of data points. Once we have this function, we can evaluate it and its derivatives at any point analytically. In some sense, it is our "best guess" for what the original object looked like, before it was sampled into a finite collection of points. Of course, this guess may get better (or worse) depending on what kind of functions we use to do the interpolation!



For this particular experiment, we will fit the neighborhood of each triangle to a *quadratic polynomial*; from there we can analytically evaluate the Gaussian curvature at the center of the triangle. A routine for performing this fit is already provided for you; here we give a high-level overview of how this routine was derived. In particular, consider the three vertices of a triangle,

plus the three remaining vertices of the neighboring triangles. We seek a quadratic polynomial

$$h : \mathbb{R}^2 \to \mathbb{R}; (x, y) \mapsto h(x, y)$$

that describes the height of the surface relative to the plane of the center triangle. "Quadratic polynomial" means that h can be expressed as a linear combination of terms involving products of at most two copies of the arguments x and y, *i.e.*,

$$h(x, y) = ax^2 + by^2 + cxy + dx + ey + f,$$

where $a, b, c, d, e, f \in \mathbb{R}$ are unknown coefficients. To determine the value of these coefficients, we can solve a system of equations

$$h(x_i, y_i) = h_i,\ i = 1, \ldots, 6,$$

where $(x_i, y_i) \in \mathbb{R}^2$ is the location of one of our six vertices (expressed relative to the plane of the center triangle), and $h_i \in \mathbb{R}$ is the height of this point over the plane. Notice that even though h is quadratic in x and y, it is *linear* in the unknown values a, b, c, d, e and f. Therefore, we have a small linear system for the unknown coefficients, which can be solved efficiently. Once we have definite values for these coefficients, we can then evaluate the Gaussian curvature of the height function using a well-established formula

$$K = \frac{h_{x,x}h_{y,y} - h_{x,y}^2}{(1 + h_x^2 + h_y^2)^2}.$$

(Can you derive this formula yourself?) Subscripts here denote partial derivatives, *e.g.*, $h_{x,y} = \frac{\partial^2}{\partial x \partial y} h$. In the code, we evaluate K at the *barycenter* p of the middle triangle, *i.e.*, at the average of its three vertex positions.

A common way to compare the accuracy of different numerical methods is to examine the *rate of convergence*. In other words, as we refine the mesh, how quickly does the estimated value approach the correct value? The question of convergence is a subtle one, because it depends both on how we refine the mesh, and how we quantify the error. In the next exercise you will look at convergence of Gaussian curvature estimates on the unit sphere, which is known to have curvature $K = 1$ at every point. In particular you will use two different sequences of progressively finer meshes: *irregular* meshes, which are not structured in any particular way, and *semi-regular* meshes, which are obtained by repeatedly subdividing an initial mesh. Error will be measured in two different norms: the L^2 norm, which essentially measures the *average* error, and the L^∞ norm, which measures the *worst* error. You will also measure the error in total curvature, *i.e.*, the (absolute value of) the difference between the sum of all the numerical values and the value predicted by the Gauss-Bonnet theorem.

CODING 15. Once you have verified that your implementation correctly computes the angle defect (by completing the exercises above), run your code on each of the meshes in the subdirectories `data/irregular` and `data/semiregular`, respectively, using the flag `-sphere`. It is *essential* that you use this flag—otherwise, you will not get the necessary output! For each mesh, record the values for L^2 error, L^∞ error, error in total Gaussian curvature, and mean edge length that are printed on the command line. (For convenience, these values are also printed on a single line at the very end of the output.) Plot each of these error functions against mean edge length, clearly labeling the approximation method, the mesh sequence, and the error norm used in each case. Plots should be drawn on a log-log scale; error functions for a given mesh sequence should all be drawn in the same plot. (Please let us know if you need guidance on suitable plotting software.) What can you infer about the different curvature approximation methods? Do they always converge? Is the rate of convergence always the same? Give an interpretation of the *slope* of an error plot on a log-log scale. If you visualize the two curvature estimates in the viewer (using 'a' and 'q' keys for

angle defect and quadratic fit, respectively), do you notice anything about where error appears? How do you think these two approximations would behave in the presence of noise? Finally, from your experience interacting with the viewer, which one seems cheaper to compute?

To submit the coding portion of your assignment, please send us *only* the files `Vertex.cpp`, `Face.cpp`, and `Mesh.cpp`. These files should compile within the default code template provided with the assignment. Make sure to include your full name in a comment at the top of each file.

CHAPTER 6

The Laplacian

Earlier we mentioned that the Laplace-Beltrami operator (commonly abbreviated as just the *Laplacian*) plays a fundamental role in a variety of geometric and physical equations. In this chapter we'll put the Laplacian to work by coming up with a discrete version of the *Poisson equation* for triangulated surfaces. As in the chapter on vertex normals, we'll see that the same discrete expression for the Laplacian (via the *cotan formula*) arises from two very different ways of looking at the problem: using *test functions* (often known as *Galerkin projection*), or by integrating differential forms (often called *discrete exterior calculus*).

6.1. Basic Properties

Before we start talking about discretization, let's establish a few basic facts about the Laplace operator Δ and the standard *Poisson problem*

$$\Delta\phi = \rho.$$

Poisson equations show up all over the place—for instance, in physics ρ might represent a mass density in which case the solution ϕ would (up to suitable constants) give the corresponding gravitational potential. Similarly, if ρ describes an charge density then ϕ gives the corresponding electric potential (you'll get to play around with these equations in the code portion of this assignment). In geometry processing a surprising number of things can be done by solving a Poisson equation (e.g., smoothing a surface, computing a vector field with prescribed singularities, or even computing the geodesic distance on a surface).

Often we'll be interested in solving Poisson equations on a compact surface M without boundary.

EXERCISE 6.1

A twice-differentiable function $\phi : M \to \mathbb{R}$ is called *harmonic* if it sits in the kernel of the Laplacian, i.e., $\Delta\phi = 0$. Argue that the *only* harmonic functions on a compact connected domain without boundary are the *constant* functions.

Your argument does not have to be incredibly formal—there are just a couple simple observations that capture the main idea. This fact is quite important because it implies that we can add a constant to any solution to a Poisson equation. In other words, if ϕ satisfies $\Delta\phi = \rho$, then so does $\phi + c$ since $\Delta(\phi + c) = \Delta\phi + \Delta c = \Delta\phi + 0 = \rho$.

EXERCISE 6.2

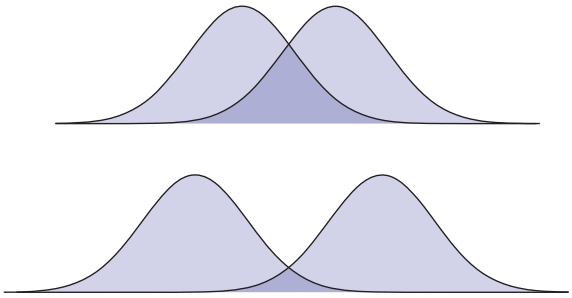
A separate fact is that on a compact domain without boundary, constant functions are not in the image of Δ. In other words, there is no function ϕ such that $\Delta\phi = c$. Why?

This fact is also important because it tells us when a given Poisson equation admits a solution. In particular, if ρ has a constant component then the problem is not well-posed. In some situations, however, it may make sense to simply remove the constant component. I.e., instead of trying to solve $\Delta\phi = \rho$ one can solve $\Delta\phi = \rho - \bar{\rho}$, where $\bar{\rho} := \int_M \rho \, dV / |M|$ and $|M|$ is the total volume of M. However, you *must* be certain that this trick makes sense in the context of your particular problem!

When working with PDEs like the Poisson equation, it's often useful to have an *inner product* between functions. An extremely common inner product is the L^2 *inner product* $\langle \cdot, \cdot \rangle$, which takes the integral of the pointwise product of two functions over the entire domain Ω:

$$\langle f, g \rangle := \int_\Omega f(x) g(x) dx.$$

In spirit, this operation is similar to the usual *dot product* on $\mathbb{R}^n$: it measures the degree to which two functions "line up." For instance, the top two functions have a large inner product; the bottom two have a smaller inner product (as indicated by the dark blue regions):



Similarly, for two vector fields X and Y we can define an L^2 inner product

$$\langle X, Y \rangle := \int_\Omega X(x) \cdot Y(x) dx$$

which measures how much the two fields "line up" at each point.

Using the L^2 inner product we can express an important relationship known as *Green's first identity*. Green's identity says that for any sufficiently differentiable functions f and g

$$\langle \Delta f, g \rangle = -\langle \nabla f, \nabla g \rangle + \langle N \cdot \nabla f, g \rangle_\partial,$$

where $\langle \cdot, \cdot \rangle_\partial$ denotes the inner product on the boundary and N is the outward unit normal.

EXERCISE 6.3

Using exterior calculus, show that Green's identity holds. Hint: apply Stokes' theorem to the $(n-1)$-form $g \star df$.

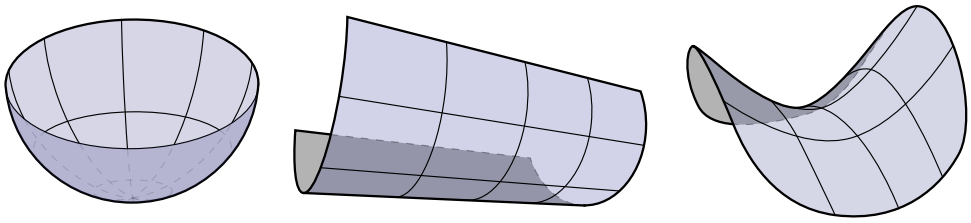
One last key fact about the Laplacian is that it is *positive-semidefinite*, i.e., Δ satisfies

$$\langle \Delta\phi, \phi \rangle \geq 0$$

for all functions ϕ. (By the way, why isn't this quantity *strictly* greater than zero?) Words cannot express the importance of (semi)definiteness. Let's think about a very simple example: functions of the form $\phi(x,y) = ax^2 + bxy + cy^2$ in the plane. Any such function can also be expressed in matrix form:

$$\phi(x,y) = \underbrace{\begin{bmatrix} x & y \end{bmatrix}}_{\mathbf{x}^T} \underbrace{\begin{bmatrix} a & b/2 \\ b/2 & c \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} x \\ y \end{bmatrix}}_{\mathbf{x}} = ax^2 + bxy + cy^2,$$

and we can likewise define positive-semidefiniteness for A. But what does it actually look like? As depicted below, positive-definite matrices ($\mathbf{x}^T A\mathbf{x} > 0$) look like a bowl, positive-semidefinite matrices ($\mathbf{x}^T A\mathbf{x} \geq 0$) look like a half-cylinder, and indefinite matrices ($\mathbf{x}^T A\mathbf{x}$ might be positive or negative depending on $\mathbf{x}$) look like a saddle:



$x^2 + y^2$ *definite* | x^2 *semidefinite* | $x^2 - y^2$ *indefinite*

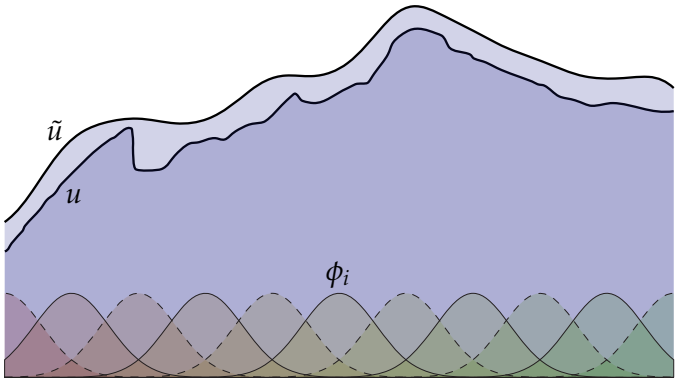
Now suppose you're a back country skiier riding down this kind of terrain in the middle of a howling blizzard. You're cold and exhausted, and you know you parked your truck in a place where the ground levels out, but where exactly is it? The snow is blowing hard and visibility is low—all you can do is keep your fingers crossed and follow the slope of the mountain as you make your descent. (Trust me: this is really how one feels when doing numerical optimization!) If you were smart and went skiing in Pos Def Valley then you can just keep heading down and will soon arrive safely back at the truck. But maybe you were feeling a bit more adventurous that day and took a trip to Semi Def Valley. In that case you'll still get to the bottom, but may have to hike back and forth along the length of the valley before you find your car. Finally, if your motto is "safety second" then you threw caution to the wind and took a wild ride in Indef Valley. In this case you may never make it home!

In short: positive-semidefinite matrices are nice because it's easy to find the minimum of the quadratic functions they describe—many tools in numerical linear algebra are based on this idea. Same goes for positive-semidefinite *linear operators* like the Laplacian Δ, which can often be thought of as sort of infinite-dimensional matrices (if you take some time to read about the spectral theorem, you'll find that this analogy runs even deeper). Given the ubiquity of Poisson equations in geometry and physics, it's a damn good thing Δ is positive-semidefinite!

EXERCISE 6.4

Using Green's first identity, show that Δ is negative-semidefinite on any compact surface M without boundary. From a practical perspective, why are negative semi-definite operators just as good as positive semi-definite ones?

6.2. Discretization via FEM

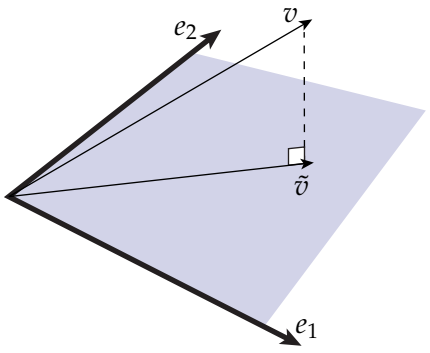


The solution to a geometric or physical problem is often described by a *function*: the temperature at each point on the Earth, the curvature at each point on a surface, the amount of light hitting each point of your retina, etc. Yet the space of *all possible* functions is mind-bogglingly large—too large to be represented on a computer. The basic idea behind the *finite element method* (FEM) is to pick a smaller space of functions and try to find the best possible solution from within this space. More specifically, if u is the true solution to a problem and $\{\phi_i\}$ is a collection of *basis functions*, then we seek the linear combination of these functions

$$\tilde{u} = \sum_i x_i \phi_i, \; x_i \in \mathbb{R}$$

such that the difference $||\tilde{u} - u||$ is as small as possible with respect to some norm. (Above we see a detailed curve u and its best approximation $\tilde{u}$ by a collection of bump-like basis functions ϕ_i.)

Let's start out with a very simple question: suppose we have a vector $v \in \mathbb{R}^3$, and want to find the best approximation $\tilde{v}$ within a plane spanned by two basis vectors $e_1, e_2 \in \mathbb{R}^3$:



Since $\tilde{v}$ is forced to stay in the plane, the best we can do is make sure there's error *only* in the normal direction. In other words, we want the error $\tilde{v} - v$ to be orthogonal to both basis vectors e_1 and e_2:

$$\begin{aligned}(\tilde{v} - v) \cdot e_1 &= 0, \\ (\tilde{v} - v) \cdot e_2 &= 0.\end{aligned}$$

In this case we get a system of two linear equations for two unknowns, and can easily compute the optimal vector $\tilde{v}$.

Now a harder question: suppose we want to solve a standard Poisson problem

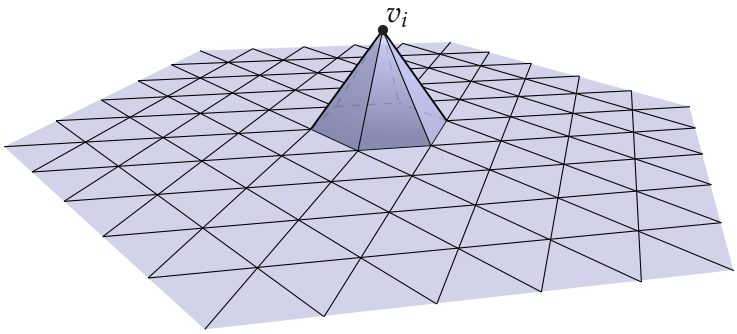
$$\Delta u = f.$$

How do we check whether a given function $\tilde{u}$ is the best possible solution? The basic picture still applies, except that our bases are now *functions* ϕ instead of finite-dimensional vectors e_i, and the simple vector dot product $\cdot$ gets replaced by the L^2 *inner product*. Unfortunately, when trying to solve a Poisson equation we don't know what the correct solution u looks like (otherwise we'd be done already!). So instead of the error $\tilde{u} - u$, we'll have to look at the *residual* $\Delta\tilde{u} - f$, which measures how closely $\tilde{u}$ satisfies our original equation. In particular, we want to "test" that the residual vanishes along each basis direction ϕ_j:

$$\langle \Delta\tilde{u} - f, \phi_j \rangle = 0,$$

again resulting in a system of linear equations. This condition ensures that the solution behaves just as the true solution would over a large collection of possible "measurements."

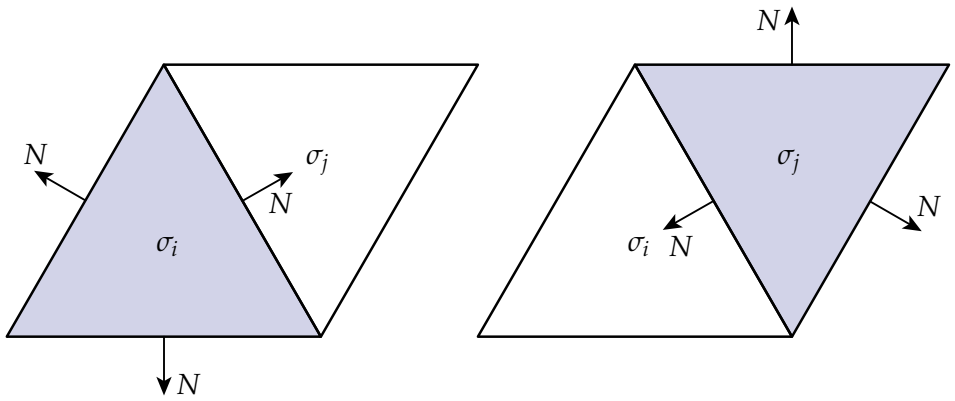
Next, let's work out the details of this system for a triangulated surface. The most natural choice of basis functions are the piecewise linear *hat functions* ϕ_i, which equal one at their associated vertex and zero at all other vertices:



At this point you might object: if all our functions are linear, and Δ is a *second* derivative, aren't we just going to get zero every time we evaluate Δu? Fortunately we're saved by Green's identity—let's see what happens if we apply this identity to our triangle mesh, by breaking up the integral into a sum over individual triangles σ:

$$\begin{aligned}\langle \Delta u, \phi_j \rangle &= \textstyle\sum_k \langle \Delta u, \phi_j \rangle_{\sigma_k} \\ &= \textstyle\sum_k \langle \nabla u, \nabla \phi_j \rangle_{\sigma_k} + \sum_k \langle N \cdot \nabla u, \phi_j \rangle_{\partial\sigma_k}.\end{aligned}$$

If the mesh has no boundary then this final sum will disappear since the normals of adjacent triangles are oppositely oriented, hence the boundary integrals along shared edges cancel each-other out:



In this case, we're left with simply

$$\langle \nabla u, \nabla \phi_j \rangle$$

in each triangle σ_k. In other words, we can "test" Δu as long as we can compute the gradients of both the candidate solution u and each basis function ϕ_j. But remember that u is itself a linear combination of the bases ϕ_i, so we have

$$\langle \nabla u, \nabla \phi_j \rangle = \left\langle \nabla \sum_i x_i \phi_i, \nabla \phi_j \right\rangle = \sum_i x_i \langle \nabla \phi_i, \nabla \phi_j \rangle.$$

The critical thing now becomes the inner product between the gradients of the basis functions in each triangle. If we can compute these, then we can simply build the matrix

$$A_{ij} := \langle \nabla \phi_i, \nabla \phi_j \rangle$$

and solve the problem

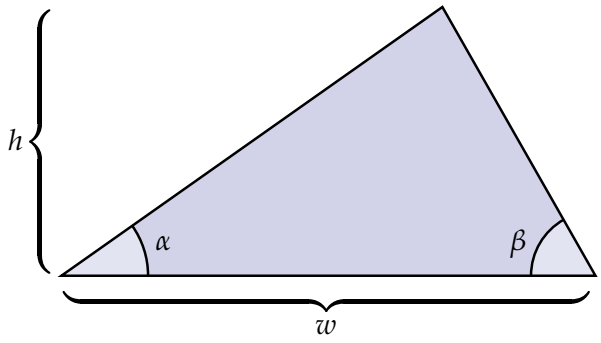
$$Ax = b$$

for the coefficients x, where the entries on the right-hand side are given by $b_i = \langle f, \phi_i \rangle$ (i.e., we just take the same "measurements" on the right-hand side).

EXERCISE 6.5

Show that the aspect ratio of a triangle can be expressed as the sum of the cotangents of the interior angles at its base, i.e.,

$$\frac{w}{h} = \cot \alpha + \cot \beta.$$

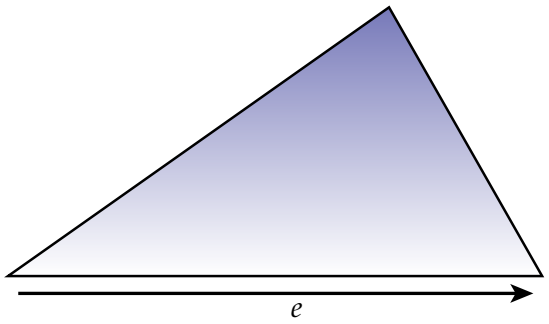


EXERCISE 6.6

Let e be the edge vector along the base of a triangle. Show that on the interior of a triangle, the gradient of the hat function ϕ associated with the opposite vertex is given by

$$\nabla\phi = \frac{e^{\perp}}{2A},$$

where $e^{\perp}$ is the vector e rotated by a quarter turn in the counter-clockwise direction and A is the area of the triangle.

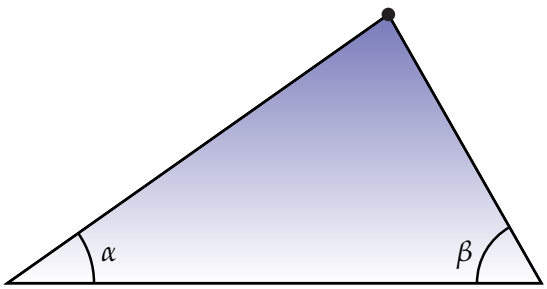


EXERCISE 6.7

Show that for any hat function ϕ associated with a given vertex

$$\langle \nabla\phi, \nabla\phi \rangle = \frac{1}{2}(\cot\alpha + \cot\beta)$$

within a given triangle, where α and β are the interior angles at the remaining two vertices.

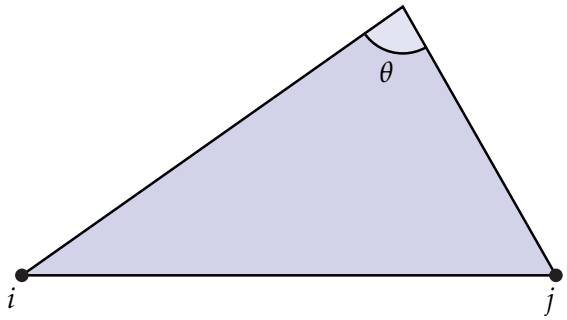


EXERCISE 6.8

Show that for the hat functions ϕ_i and ϕ_j associated with vertices i and j (respectively) of the same triangle, we have

$$\langle \nabla \phi_i, \nabla \phi_j \rangle = -\frac{1}{2} \cot \theta,$$

where θ is the angle between the opposing edge vectors.



Putting all these facts together, we find that we can express the Laplacian of u at vertex i via the infamous *cotan formula*

$$(\Delta u)_i = \frac{1}{2} \sum_j (\cot \alpha_j + \cot \beta_j)(u_j - u_i),$$

where we sum over the immediate neighbors of vertex i.

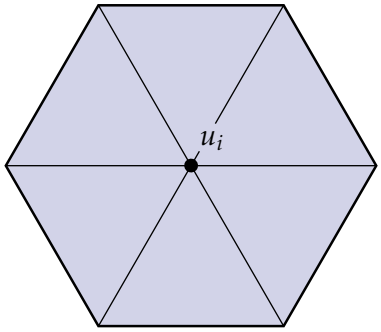
6.3. Discretization via DEC

The FEM approach reflects a fairly standard way to discretize partial differential equations. But let's try a different approach, based on discrete exterior calculus (DEC). Interestingly enough, although these two approaches are quite different, we end up with exactly the same result!

Again we want to solve the Poisson equation $\Delta u = f$, which (if you remember our discussion of differential operators) can also be expressed as

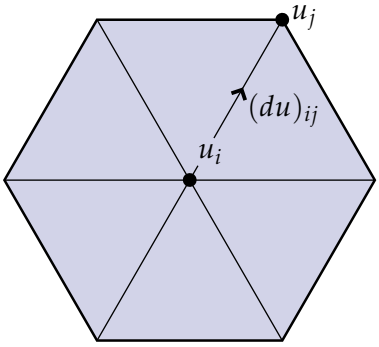
$$\star d \star du = f.$$

We already outlined how to discretize this kind of expression in the notes on discrete exterior calculus, but let's walk through it step by step. We start out with a 0-form u, which is specified as a number u_i at each vertex:



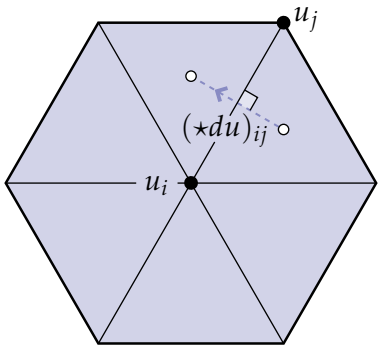
Next, we compute the discrete exterior derivative du, which just means that we want to *integrate* the derivative along each edge:

$$(du)_{ij} = \int_{e_{ij}} du = \int_{\partial e_{ij}} u = u_j - u_i.$$



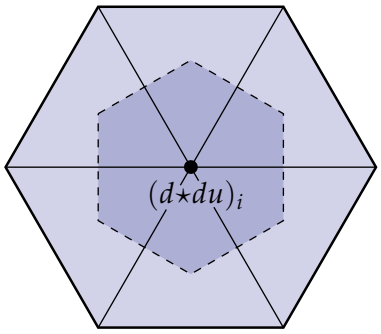
(Note that the boundary ∂e_{ij} of the edge is simply its two endpoints v_i and v_j.) The Hodge star converts a circulation along the edge e_{ij} into the flux through the corresponding dual edge $e_{ij}^\star$. In particular, we take the *total circulation* along the primal edge, divide it by the edge length to get the *average pointwise circulation*, then multiply by the dual edge length to get the *total flux* through the dual edge:

$$(\star du)_{ij} = \frac{|e_{ij}^\star|}{|e_{ij}|}(u_j - u_i).$$



Here $|e_{ij}|$ and $|e^\star_{ij}|$ denote the length of the primal and dual edges, respectively. Next, we take the derivative of $\star du$ and integrate over the whole dual cell:

$$(d \star du)_i = \int_{C_i} d \star du = \int_{\partial C_i} \star du = \sum_j \frac{|e^\star_{ij}|}{|e_{ij}|}(u_j - u_i).$$



The final Hodge star simply divides this quantity by the area of C_i to get the average value over the cell, and we end up with a system of linear equations

$$(\star d \star du)_i = \frac{1}{|C_i|} \sum_j \frac{|e^\star_{ij}|}{|e_{ij}|}(u_j - u_i) = f_i$$

where f_i is the value of the right-hand side at vertex i. In practice, however, it's often preferable to move the area factor $|C_i|$ to the right hand side, since the resulting system

$$(d \star du)_i = \sum_j \frac{|e^\star_{ij}|}{|e_{ij}|}(u_j - u_i) = |C_i| f_i$$

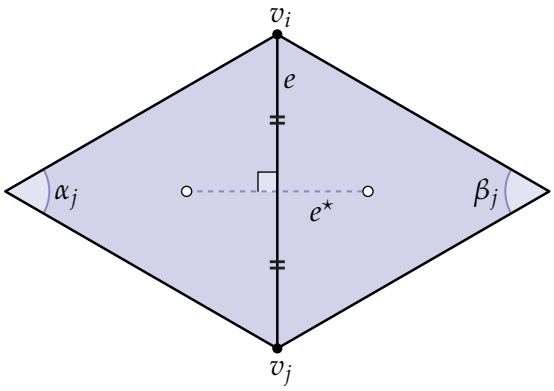
can be represented by a *symmetric* matrix. (Symmetric matrices are often easier to deal with numerically and lead to more efficient algorithms.) Another way of looking at this transformation is to imagine that we discretized the system

$$d \star du = \star f.$$

In other words, we converted an equation in terms of 0-forms into an equation in terms of n-forms. When working with surfaces, the operator $d \star d$ is sometimes referred to as the *conformal Laplacian*, because it does not change when we subject our surface to a conformal transformation. Alternatively, we can think of $d \star d$ as simply an operator that gives us the value of the Laplacian integrated over each dual cell of the mesh (instead of the pointwise value).

EXERCISE 6.9

Consider a simplicial surface and suppose we place the vertices of the dual mesh at the circumcenters of the triangles (i.e., the center of the unique circle containing all three vertices):



Demonstrate that the dual edge $e^\star$ (i.e., the line between the two circumcenters) bisects the primal edge orthogonally, and use this fact to show that

$$\frac{|e^\star_{ij}|}{|e_{ij}|} = \frac{1}{2}(\cot \alpha_j + \cot \beta_j).$$

Hence the DEC discretization yields precisely the same "cotan-Laplace" operator as the Galerkin discretization.

6.4. Meshes and Matrices

So far we've been giving a sort of "algorithmic" description of operators like Laplace. For instance, we determined that the Laplacian of a scalar function u at a vertex i can be approximated as

$$(\Delta u)_i = \frac{1}{2}\sum_j (\cot \alpha_j + \cot \beta_j)(u_j - u_i),$$

where the sum is taken over the immediate neighbors j of i. In code, this sum could easily be expressed as a *loop* and evaluated at any vertex. However, a key aspect of working with discrete differential operators is building their *matrix representation*. The motivation for encoding an operator as a matrix is so that we can solve systems like

$$\Delta u = f$$

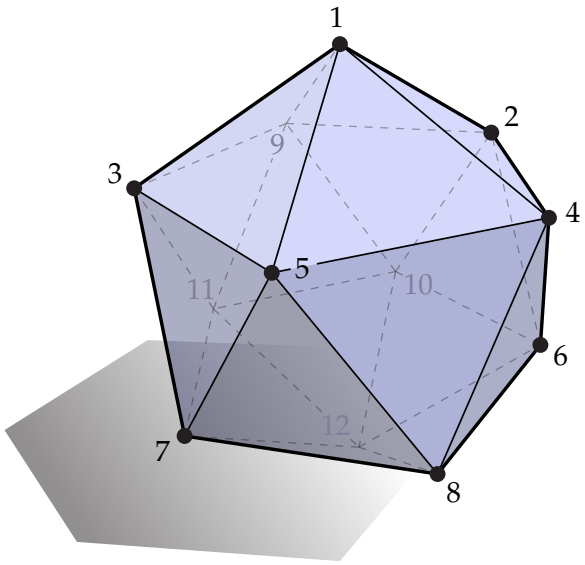
using a standard numerical linear algebra package. (To make matters even more complicated, some linear solvers are perfectly happy to work with algorithmic representations of operators called *callback functions*—in general, however, we'll need a matrix.)

In the case of the Poisson equation, we want to construct a matrix $L \in \mathbb{R}^{|V|\times|V|}$ (where $|V|$ is the number of mesh vertices) such that for any vector $u \in \mathbb{R}^{|V|}$ of values at vertices, the expression Lu effectively evaluates the formula above. But let's start with something simpler—consider an operator B that computes the sum of all neighboring values:

$$(Bu)_i = \sum_j u_j$$

How do we build the matrix representation of this operator? Think of B a machine that takes a vector u of input values at each vertex and spits out another vector Bu of output values. In order

for this story to make sense, we need to know which values correspond to which vertices. In other words, we need to assign a unique *index* to each vertex of our mesh, in the range $1, \ldots, |V|$:



It doesn't matter which numbers we assign to which vertices, so long as there's one number for every vertex and one vertex for every number. This mesh has twelve vertices and vertex 1 is next to vertices 2, 3, 4, 5, and 9. So we could compute the sum of the neighboring values as

$$(Bu)_1 = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \\ u_7 \\ u_8 \\ u_9 \\ u_{10} \\ u_{11} \\ u_{12} \end{bmatrix}.$$

Here we've put a "1" in the jth place whenever vertex j is a neighbor of vertex 1 and a "0" otherwise. Since this row gives the "output" value at the first vertex, we'll make it the first row of the matrix B. The entire matrix looks like this:

$$B = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

(You could verify that this matrix is correct, or you could go outside and play in the sunshine. Your choice.) In practice, fortunately, we don't have to build matrices "by hand"—we can simply start with a matrix of zeros and fill in the nonzero entries by looping over local neighborhoods of our mesh.

Finally, one important thing to notice here is that many of the entries of B are *zero*. In fact, for most discrete operators the number of zeros far outnumbers the number of nonzeros. For this reason, it's usually a good idea to use a *sparse matrix*, i.e., a data structure that stores only the location and value of nonzero entries (rather than explicitly storing every single entry of the matrix). The design of sparse matrix data structures is an interesting question all on its own, but conceptually you can imagine that a sparse matrix is simply a list of triples (i, j, x) where $i, j \in \mathbb{N}$ specify the row and column index of a nonzero entry and $x \in \mathbb{R}$ gives its value.

6.5. The Poisson Equation

In the first part of the coding assignment you'll build the cotan-Laplace operator and use it to solve the scalar Poisson equation

$$\Delta\phi = \rho$$

on a triangle mesh, where ρ can be thought of as a (mass or charge) density and ϕ can be thought of as a (gravitational or electric) potential. Once you've implemented the methods below, you can visualize the results via the Viewer. (If you want to play with the density function ρ, take a look at the method `Viewer::updatePotential`.)

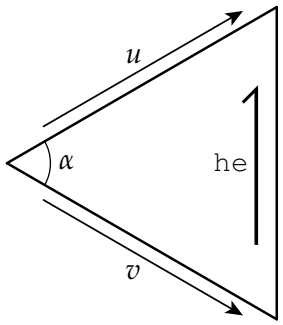
Recall from the end of Section 6.3 that it's often convenient to put the vertex areas $|C_i|$ on the right-hand side, so that the system becomes symmetric. In our matrix representation, this means our Poisson equation becomes a matrix equation

$$\mathsf{L}u = \mathsf{M}\rho, \tag{1}$$

where the matrix $\mathsf{L} \in \mathbb{R}^{n \times n}$ encodes the cotan formula, and $\mathsf{M} \in \mathbb{R}^{n \times n}$ encodes the multiplication of the value ϕ_i at each vertex by the vertex area $|C_i|$. (In the finite element setting, L is often called the *stiffness matrix*, and M is the *mass matrix*.) The Laplace operator can then be expressed as $\mathsf{A} := \mathsf{M}^{-1}\mathsf{L}$, though in practice one typically solves Equation 1 rather than building the matrix A directly.

CODING 16. Implement the method `Mesh::indexVertices()` which assigns a unique ID to each vertex in the range $0, \ldots, |V| - 1$.

CODING 17. Derive an expression for the cotangent of a given angle purely in terms of the two incident edge vectors and the standard Euclidean dot product $(\cdot)$ and cross product $(\times)$. Implement the method `HalfEdge::cotan()`, which computes the cotangent of the angle across from a given halfedge.

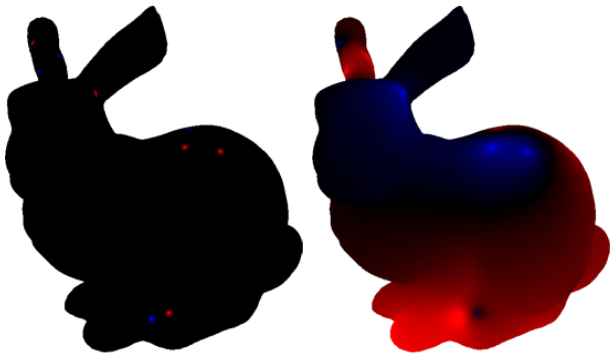


CODING 18. Implement the methods `Face::area()` and `Vertex::dualArea()`. For the dual area of a vertex you can simply use one-third the area of the incident faces—you do not need to compute the area of the circumcentric dual cell. (This choice of area will not affect the order of convergence.)

CODING 19. Using the methods you've written so far, implement the method `Mesh::buildLaplacian()` which builds a sparse matrix representing the cotan-Laplace operator. (Remember to initialize the matrix to the correct size!)

CODING 20. Implement the method `Mesh::solveScalarPoissonProblem()` which solves the problem $\Delta\phi = \rho$ where ρ is a scalar density on vertices (stored in `Vertex::rho`). You can use the method `solve` from `SparseMatrix.h`; ρ and ϕ should be represented by instances of `DenseMatrix` of the appropriate size. Be careful about appropriately incorporating *dual areas* into your computations; also remember that the right-hand side cannot have a constant component!

You should verify that your code produces results that look something like these two images (density on the left; corresponding potential on the right):

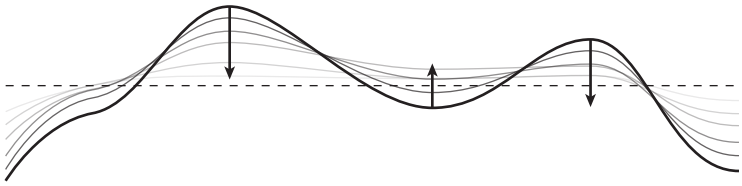


6.6. Implicit Mean Curvature Flow

Next, you'll use nearly identical code to smooth out geometric detail on a surface mesh (also known as *fairing* or *curvature flow*). The basic idea is captured by the *heat equation*, which describes

the way heat diffuses over a domain. For instance, if u is a scalar function describing the temperature at every point on the real line, then the heat equation is given by

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}.$$



Geometrically this equation simply says that concave bumps get pushed down and convex bumps get pushed up—after a long time the heat distribution becomes completely flat. We also could have written this equation using the Laplacian: $\frac{\partial u}{\partial t} = \Delta u$. In fact, this equation is exactly the one we'll use to smooth out a surface, except that instead of considering the evolution of temperature, we consider the flow of the surface $f : M \to \mathbb{R}^3$ itself:

$$\frac{\partial f}{\partial t} = \Delta f.$$

Remember from our discussion of vertex normals that $\Delta f = 2HN$, i.e., the Laplacian of position yields (twice) the mean curvature times the unit normal. Therefore, the equation above reads, "move the surface in the direction of the normal, with strength proportional to mean curvature." In other words, it describes a *mean curvature flow*.

So how do we compute this flow? We already know how to discretize the term Δf—just use the cotangent discretization of Laplace. But what about the time derivative $\frac{\partial f}{\partial t}$? There are all sorts of interesting things to say about discretizing time, but for now let's use a very simple idea: the change over time can be approximated by the *difference* of two consecutive states:

$$\frac{\partial f}{\partial t} \approx \frac{f_h - f_0}{h},$$

where f_0 is the initial state of our system (here the initial configuration of our mesh) and f_h is the configuration after a mean curvature flow of some duration $h > 0$. Our discrete mean curvature flow then becomes

$$\frac{f_h - f_0}{h} = \Delta f.$$

The only remaining question is: which values of f do we use on the right-hand side? One idea is to use f_0, which results in the system

$$f_h = f_0 + h\Delta f_0.$$

This scheme, called *forward Euler*, is attractive because it can be evaluated directly using the known data f_0—we don't have to solve a linear system. Unfortunately, forward Euler is not numerically stable, which means we can take only very small time steps h. One attractive alternative is to use f_h as the argument to Δ, leading to the system

$$\underbrace{(I - h\Delta)}_{A} f_h = f_0,$$

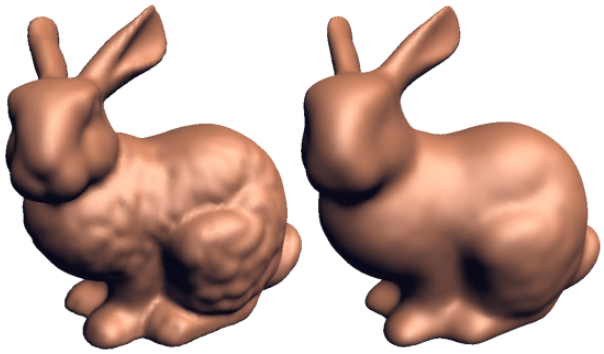
where I is the identity matrix (try the derivation yourself!) This scheme, called *backward Euler*, is far more stable, though we now have to solve a linear system $Af_h = f_0$. Fortunately this system is

highly *sparse*, which means it's not too expensive to solve in practice. (Note that this system is not much different from the Poisson system.)

CODING 21. Implement the method `Mesh::buildFlowOperator()`, which should look very similar to `Mesh::buildLaplacian`.

CODING 22. Implement the method `Mesh::computeImplicitMeanCurvatureFlow()`. Note that you can treat each of the components of f (x, y, and z) as separate scalar quantities.

You should verify that your code produces results that look something like these two images (original mesh on the left; smoothed mesh on the right):



6.7. Boundary Conditions

Boundary conditions are exactly what they sound like: conditions that solutions to a equation (like the Laplace or Poisson equation) must satisfy along the domain boundary. In general, boundary conditions can be a pain in the butt: they're often difficult to work out, and are easy to get wrong in code. From a practical point of view, a nice thing about working with surfaces (rather than regions of $\mathbb{R}^n$, which *must* have boundary) is that you can first consider a domain *without* boundary, then add the additional complexity of boundary conditions once everything is working properly. However, for many problems boundary conditions are critical, because they determine the behavior of the entire solution—consider for instance the Laplace equation, where the solution on the interior is completely determined by the values on the boundary.

Let's take a look at how to derive and implement boundary conditions for a Laplace equation $\Delta u = 0$ on a surface M with boundary ∂M. We will use n to denote the unit normal to the boundary, which by convention we will assume points outward. Two of the most common kinds of boundary conditions are

- **Dirichlet** — along ∂M, the *value* of u must equal a given function $g : \partial M \to \mathbb{R}$. With Dirichlet boundary conditions, a Laplace equation effectively tries to extend or *interpolate* the known boundary values g over the rest of the domain as smoothly as possible, yielding the function u.
- **Neumann** — along ∂M, the *normal derivative* $\partial u / \partial n$ must equal a function $h : \partial M \to \mathbb{R}$. With Neumann conditions, a Laplace equation also seeks a function that is as smooth as possible, but now with prescribed "slope" at the boundary—alternatively, if we think of

the function u as describing the steady-state temperature at each point of the domain, Neumann conditions describe an inflow/outflow of heat at the boundary.

6.7.0.1. *Dirichlet Boundary Conditions.* Let's first consider a Poisson problem with Dirichlet boundary conditions (Laplace is the special case where $f = 0$):

$$\begin{aligned} \Delta u &= f \quad \text{on } M, \\ u &= g \quad \text{on } \partial M. \end{aligned}$$

In the discrete case, this equation will always have a unique solution, no matter which functions f and g we pick. The physical intuition behind the existence and uniqueness of a solution is that this equation describes the steady-state temperature u in a system with a heat source f on the interior, and clamped to a temperature g along the boundary.

Recall that in the discrete case we can express Δu via the cotan formula, yielding an equation

$$\frac{1}{2} \sum_{j \in \mathcal{N}(i)} (\cot \alpha_j + \cot \beta_j)(u_j - u_i) = 0$$

for each vertex i, where the sum is taken over all neighbors $\mathcal{N}(i)$ of vertex i. If j is on the boundary, its value is already given by $u_j = g_j$ for some known constant g_j.

Block matrices. This fact does not really change our expression for the discrete Laplacian (except, perhaps, for swapping out some "u"s for some "g"s), but it does affect the way we assemble the matrix representing our linear system. In general, a nice way to keep track of what's going on with boundary conditions is to express matrices in *block form*, i.e., to write out a matrix where each entry actually corresponds to a smaller matrix. In this case, suppose we index our vertices so that all the interior vertices come first, followed by all the boundary vertices; let $I, B \subset V$ denote the set of interior and boundary vertices, respectively. Then we can write the cotan matrix $\mathsf{L} \in \mathbb{R}^{|V| \times |V|}$ in block form as

$$\mathsf{L} = \begin{bmatrix} \mathsf{L}_{II} & \mathsf{L}_{IB} \\ \mathsf{L}_{BI} & \mathsf{L}_{BB} \end{bmatrix},$$

where L_{II} is the $|I| \times |I|$ block of L, corresponding to entries L_{ij} where both i and j are boundary vertices (and similarly for $\mathsf{L}_{IB} \in \mathbb{R}^{|I| \times |B|}, \mathsf{L}_{BI} \in \mathbb{R}^{|B| \times |I|}, \mathsf{L}_{BB} \in \mathbb{R}^{|B| \times |B|}$). Note also that since L is a symmetric matrix, $\mathsf{L}_{BI} = \mathsf{L}_{IB}^T$. Likewise, consider a discrete Poisson equation

$$\mathsf{L}\mathsf{u} = \mathsf{b},$$

where $\mathsf{u} \in \mathbb{R}^{|V|}$ are the solution values at each vertex, and the right-hand side $\mathsf{b} := \mathsf{M}\mathsf{f}$ gives the source values f_i at each vertex times the vertex areas $|C_i|$ (see discussion around Equation 1). We can write this system in block form as

$$\begin{bmatrix} \mathsf{L}_{II} & \mathsf{L}_{IB} \\ \mathsf{L}_{BI} & \mathsf{L}_{BB} \end{bmatrix} \begin{bmatrix} \mathsf{u}_I \\ \mathsf{u}_B \end{bmatrix} = \begin{bmatrix} \mathsf{b}_I \\ \mathsf{b}_B \end{bmatrix}.$$

For a Dirichlet problem, we already know that $\mathsf{u}_B = \mathsf{g}$, where $\mathsf{g} \in \mathbb{R}^{|B|}$ is the vector of known boundary values. Hence, we just have to solve for the unknown values $\mathsf{u}_I \in \mathbb{R}^{|I|}$, which we can do by solving the first row of our block equation:

$$\mathsf{L}_{II}\mathsf{u}_I = \mathsf{b}_I - \mathsf{L}_{IB}\mathsf{u}_B.$$

Since L_{II} is an $I \times I$ matrix (and has full rank), this smaller system alone is enough to determine all the unknown values. Equivalently, we can imagine that while building row i of the Laplace matrix, we check if the current neighbor j is on the boundary—if so, we subtract the (known) value g_j times the cotan weight for edge ij from the entry of the right-hand side corresponding to vertex i.

An alternative—but completely equivalent—way to enforce Dirichlet conditions is to build a larger system, where we still solve for both u_I and u_B, but force the boundary values to be equal to the given Dirichlet data:

$$\begin{bmatrix} \mathsf{L}_{II} & \mathsf{L}_{IB} \\ 0 & \mathsf{I}_B \end{bmatrix} \begin{bmatrix} \mathsf{u}_I \\ \mathsf{u}_B \end{bmatrix} = \begin{bmatrix} \mathsf{b}_I \\ \mathsf{g} \end{bmatrix},$$

where here I_B notes the $B \times B$ identity matrix. This system is slightly larger and no longer symmetric, and hence may take a bit longer to solve. When faced with alternatives like these, a good maxim is: *get it right, **then** make it fast*. In other words, choose the formulation that *you* find easiest to implement in a bug-free fashion. Once everything is working correctly, *then* you can split hairs over whether the implementation can be made faster using (say) a solver that handles only symmetric matrices.

6.7.0.2. *Neumann Boundary Conditions.* Let's now consider a Laplace problem with Neumann boundary conditions:

$$\begin{aligned} \Delta u &= f \quad \text{on } M, \\ \partial u / \partial n &= h \quad \text{on } \partial M. \end{aligned}$$

The normal derivative $\partial u / \partial n$ can also be written as $n \cdot \nabla u$, which will be helpful for working out the discrete version. Importantly, unlike the Dirichlet problem, *the Neumann problem does not always have a solution!*. Intuitively, f acts as a heat source on the interior of the domain, and h describes how much heat is flowing in and out through the domain boundary. Hence, "what goes in must come out": the two functions must integrate to the same value in order for this equation to be meaningful. This observation is made more explicit via the divergence theorem: the integral over the boundary of the normal derivatives h must be equal to the integral over the whole domain of the source function f:

$$\int_{\partial M} h \, ds = \int_{\partial M} n \cdot \nabla u \, ds = \int_M \nabla \cdot \nabla u \, dA = \int_M \Delta u \, dA = \int_M f \, dA.$$

If f and h are not compatible in this way, then the equation has no solutions[1]. Note also that if a solution does exist, it is unique only up to constant shifts $u + c$ for $c \in \mathbb{R}$, since both the Laplacian and the normal derivative eliminate constants.

Similar to Section 6.3, we can discretize our Poisson equation by asking that Δu and f integrate to the same value over the dual cell C_i associated with each vertex i:

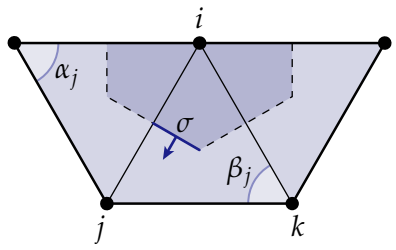
$$\int_{C_i} \Delta u \, dA = \int_{C_i} f \, dA. \tag{2}$$

When i is an interior vertex, the integral on the left-hand side leads to the same discrete expression for $(\Delta u)_i$ as before, via the cotan formula. But at boundary vertices, the cell C_i is "clipped" by the domain boundary, and we must revise our formula. First, we apply the divergence theorem to Equation 2 to obtain an integral over the cell boundary:

$$\int_{C_i} \Delta u \, dA = \int_{C_i} \nabla \cdot \nabla u \, dA = \int_{\partial C_i} n \cdot \nabla u \, ds. \tag{3}$$

The cell boundary ∂C_i can be decomposed into a piece running through the domain interior, and a piece along the domain boundary ∂M. On the interior, we can further break up the cell boundary into linear segments within each triangle ijk, such as the dark blue segment σ in the figure below:

[1] This fact is especially dangerous, because some numerical linear solvers will still return a solution anyway! For instance, the standard "backslash" operator will give the least-squares solution, without reporting any kind of error. The best thing to do, always, is to *check the residual yourself*. I.e., if you think you have a solution to a system $Ax = b$, just take an extra moment to compute $\|Ax - b\|$ and see that it really is equal to (numerical) zero.



Within each triangle ijk, the function u can be expressed as a linear combination of hat functions ϕ_p at the three vertices:

$$u = \sum_{p \in \{i,j,k\}} u_p \phi_p.$$

As shown in Exercise 6, the gradient of ϕ_k is orthogonal to edge ij. Since the normal n_{ij} is parallel to this edge, our integrand over the segment σ becomes just

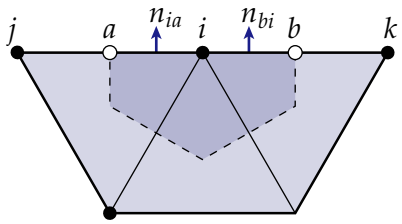
$$n_{ij} \cdot \nabla u = \sum_{p \in \{i,j,k\}} u_p n_{ij} (\cdot \nabla \phi_p) = u_i (n_{ij} \cdot \nabla \phi_i) + u_j (n_{ij} \cdot \nabla \phi_j),$$

i.e., the contribution from σ depends only on the values of u at vertices i and j. Using a calculation very similar to Exercise 7, we then get

$$\int_\sigma n \cdot \nabla u \, ds = \tfrac{1}{2} \cot \beta_j (u_j - u_i).$$

When j is an interior edge, the segment on the other side of edge ij likewise contributes a term $\frac{1}{2} \cot \alpha_j (u_j - u_i)$, and we get the usual cotan weight for this edge. But when ij is a boundary edge, we get a contribution only from one side (and hence do not need to worry about the "unknown" angle in the basic cotan formula, which is not needed).

Most importantly, we still need to account for the contribution along the domain boundary ∂M. Let a and b be the points where the dual cell boundary ∂C_i meets ∂M:



We need to add to our integral the contribution of $n \cdot \nabla u$ along the segments ia and bi—intuitively, we need to account for any flow into or out of the domain. How do we compute these values? Though it is tempting at this moment to, for instance, write down an expression for the gradient "just inside" the domain, the whole point is that *we get to choose this value*, i.e., we get to choose how much "stuff" is flowing in or out of the domain at the vertex i. In particular, if we imagine that we have a continuous function $h : \partial M \to \mathbb{R}$ determining our Neumann boundary conditions, then we can choose a value h_i representing the integral of the normal flux through this piece of the boundary:

$$h_i := \int_{\partial C_i \cap \partial M} n \cdot \nabla u.$$

For instance, suppose we already had values h_{ij} and h_{ki} giving the total flux through boundary edges ij and ki (i.e., the normal derivative *times* the edge length). Then we would want to add the term $h_i = (h_{ij} + h_{ki})/2$ to the overall integral from Equation 3. Importantly, since the Neumann values are known constants that are *added* to our expression for the Laplacian at vertex i, they are *subtracted* from the right-hand side in our final matrix equation.

The construction of the final system can again be made clear by writing it in block form. First, we build the full $|V| \times |V|$ matrix L, making only one minor modification: for edges ij along the domain boundary, we have entries $\mathsf{L}_{ij} = \frac{1}{2}\cot\alpha$ (where α is the one angle we know), rather than $\frac{1}{2}(\cot\alpha + \cot\beta)$. The diagonal entries L_{ii} should still be the sum of all off-diagonal entries, so that the rows sum to zero (hence, constant functions are still in the null space of L). Using this matrix, our Poisson problem with Neumann boundary conditions then becomes simply

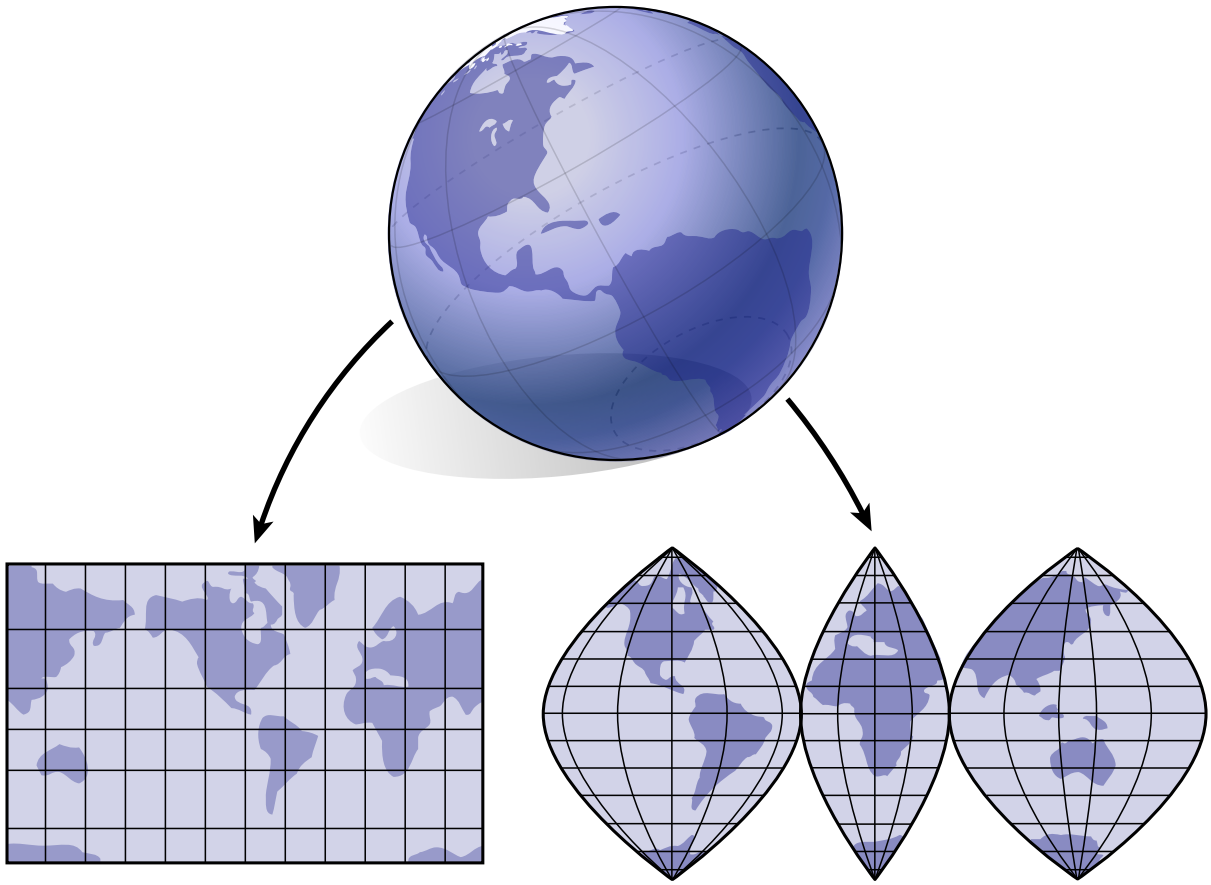
$$\mathsf{L}\mathsf{u} = \begin{bmatrix} \mathsf{b}_I \\ \mathsf{b}_B - \mathsf{h} \end{bmatrix},$$

where $\mathsf{h} \in \mathbb{R}^{|B|}$ is the vector of (integrated) Neumann values along the boundary. Notice that unlike the Dirichlet problem, we still have to solve for *all* values of u—not just the interior values—and hence use the full matrix L.

CHAPTER 7

Surface Parameterization

In this chapter we're going to look at the problem of *surface parameterization*. The basic idea is that we have a surface sitting in space, and we want to "flatten it out" in the plane. The oldest example, perhaps, is making a map of the Earth:

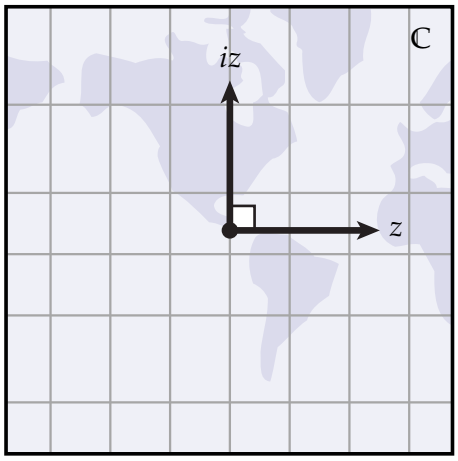


One thing you'll notice about maps of the Earth is that they all look distorted in some way: Greenland looks way too big, or "north" doesn't quite make a right angle with "east." These phenomena reflect a general fact about surface parameterization: it's usually impossible to flatten a surface while perfectly preserving both lengths and angles—in other words, not every surface admits an *isometric* parameterization. It is, however, always possible to find an angle-preserving or *conformal* parameterization, which is what we'll do in this assignment.

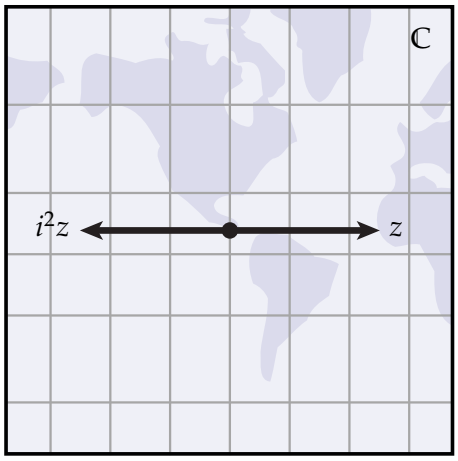
7.0.1. Two Quarter Turns Make a Flip: A Brief Review of Complex Numbers. If you've ever seen the complex numbers, you've probably encountered the totally abysmal description of the imaginary unit i as the "square root" of negative one:

$$i = \sqrt{-1}.$$

Since the square of any *real* number is nonnegative, one argues, the number i must be "*imaginary*." Makes sense, right? The problem with this story is that it neglects the simple geometric meaning of i, which turns out to be quite real! So, let's start over: the symbol i denotes a quarter-turn in the counter-clockwise direction. For instance, if z is a vector pointing east, then iz is a vector pointing north:



What happens if we apply another quarter turn? We get a half turn, of course!

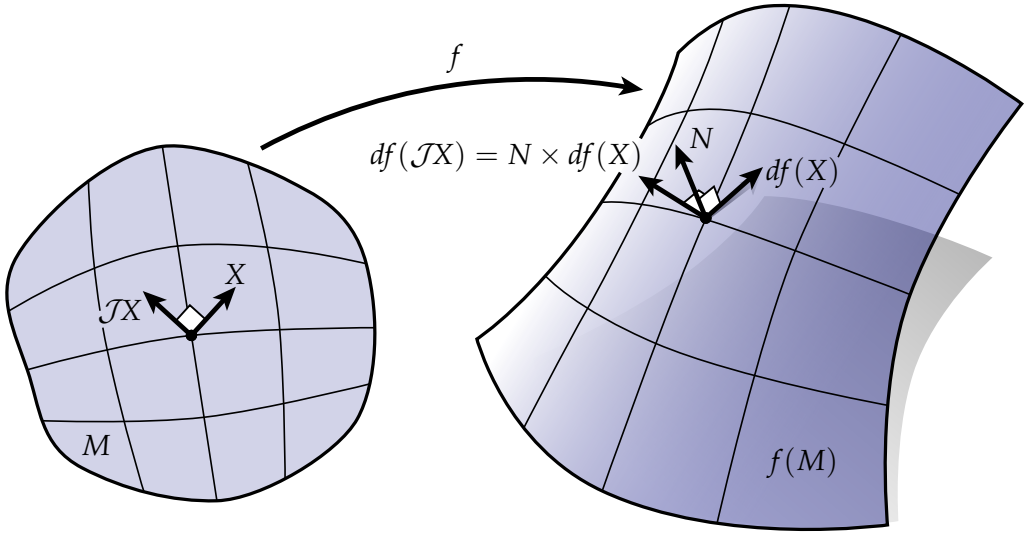


In other words, we have $i(iz) = -z$. We can abbreviate this statement by writing $i^2 z = -z$, which means we must have

$$i^2 = -1,$$

i.e., *two quarter turns make a flip*. That's all. No square roots, and very little imagination required. Thinking of the imaginary unit i as a 90-degree rotation will be essential in our discussion of conformal maps.

7.1. Conformal Structure



For a surface $f\colon M \to \mathbb{R}^3$ sitting in space, we also have a simple way to express 90-degree rotations. In particular, if $df(X)$ is a tangent vector in $\mathbb{R}^3$, we can express a quarter turn in the counter-clockwise direction by taking the cross product with the unit normal N:

$$N \times df(X),$$

Since the vector $N \times df(X)$ is also tangent to the immersed surface $f(M)$, there must be some corresponding tangent vector on the domain M—let's call this vector $\mathcal{J}X$. In other words,

$$df(\mathcal{J}X) = N \times df(X).$$

The map $\mathcal{J}$ is called the *conformal structure* induced by the immersion f. (Some might prefer to call $\mathcal{J}$ an *almost complex structure* or a *linear complex structure*, but for surfaces all of these ideas are essentially the same.) A *Riemann surface* is a surface with a complex structure, *i.e.*, it is a surface where we know how to measure angles between tangent vectors (but possibly not their length).

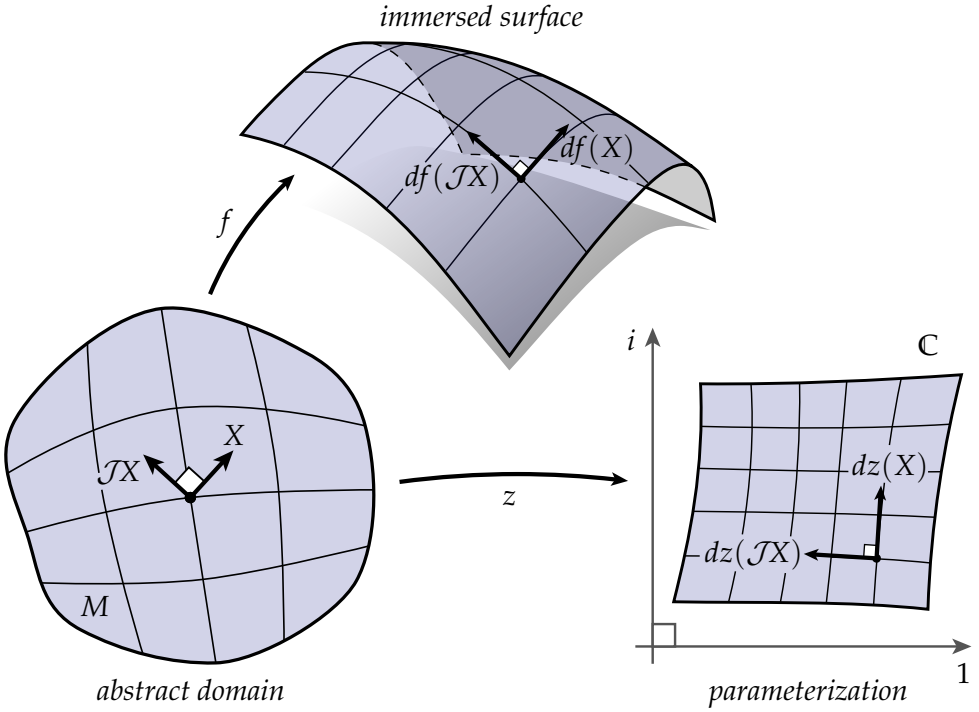
The most important thing to remember about the conformal structure $\mathcal{J}$ is that, like the imaginary unit i, there is nothing strange or mysterious about it: it denotes a quarter turn in the counter-clockwise direction. And, as before, two quarter turns make a flip:

$$\mathcal{J}^2 = -\mathrm{id},$$

where id just denotes the identity.

At this point you might be wondering, "ok, so why do we bother with two different objects, i and $\mathcal{J}$, that do exactly the same thing?" This story is especially confusing given that the domain M in the picture above looks an awful lot like a piece of the (complex) plane. But in general, M does not have to be a piece of the plane—it can be any topological surface (a sphere, a donut, etc.). And in general, *tangent vectors are not complex numbers!* Therefore, it doesn't make much sense to write iX, nor does it make sense to write $\mathcal{J}z$. But there's clearly a relationship we want to capture here, and that relationship is described by our good friends Augustin Cauchy and Bernhard Riemann.

7.2. The Cauchy-Riemann Equation



Remember that, like the cartographers of yore, our goal is to parameterize a given surface over the plane. In particular, we want to find a map that preserves angles. How can we express this condition more explicitly? Well, we know how to express 90-degree rotations on the surface, using the complex structure $\mathcal{J}$. And we know how to express 90-degree rotations in the plane, using the imaginary unit i. Therefore, an angle-preserving or *conformal map* $z : M \to \mathbb{C}$ must satisfy the *Cauchy-Riemann equation*

$$dz(\mathcal{J}X) = i\,dz(X)$$

for all tangent vectors X on M. In other words, rotating a vector by 90-degrees and then mapping it into the plane is no different from mapping it into the plane and then rotating it by 90 degrees. To be more precise, z is a *holomorphic* function, meaning that it preserves both angles *and* orientation ($dz(X) \times dz(\mathcal{J}X)$ sticks "out" of the plane). Maps that preserve angles but *reverse* orientation are called *antiholomorphic*.

Note that the meaning of dz in the Cauchy-Riemann equation is no different from the meaning of df when we talk about an immersion f: it tells us how tangent vectors get "stretched out" as we go from one space to another. In fact, like f, the map z is just another immersion of M—this time into $\mathbb{C}$ instead of $\mathbb{R}^3$. The basic idea of the Cauchy-Riemann equation is that both of these immersions should share an identical notion of angle, as emphasized in the illustration above. One way to look at this picture is to imagine that we start out with the abstract domain M, which "doesn't know" how to measure the angle between two vectors. By immersing M in three-dimensional space (via the map f), we inherit the usual Euclidean notion of angle. We then look for a map z to the complex plane that shares this same notion of angle (but perhaps a different notion of length!).

7.3. Differential Forms on a Riemann Surface

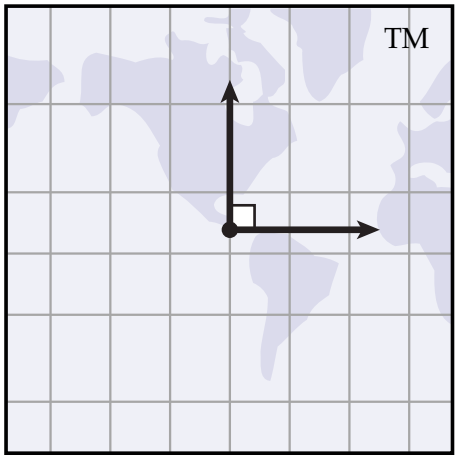
Half of life is knowing what you want. The other half is knowing how to get it. In this case, we know what we want: a map z satisfying the Cauchy-Riemann equation. But how do we actually compute it? In order to connect this question to our existing computational tools, let's rewrite Cauchy-Riemann in terms of exterior calculus. In fact, let's revisit the whole idea of differential forms in the context of surfaces and their conformal structure. As we'll see, this way of thinking can lead to some beautifully simple geometric expressions.

Recall our geometric interpretation of real-valued differential forms: a k-form measures some k-dimensional volume (length, area, etc.). One thing to notice is that on an n-manifold, there are no $n+1$-dimensional volumes to measure. For instance, we can't measure two-dimensional areas on a curve—just one-dimensional lengths. Likewise, we can't measure three-dimensional volumes on a surface—just 1D lengths and 2D areas. For this reason, differential forms on surfaces become particularly simple to understand:

- 0-forms look like scalar functions,
- 1-forms look like vector fields, and
- 2-forms look like scalar multiples of area.

That's where the list ends! There are no 3-forms (or 4-forms, or 5-forms...) to worry about. (A more algebraic way to convince yourself of this fact is to consider the antisymmetry of the wedge product: $\alpha \wedge \beta = -\beta \wedge \alpha$. What happens when you take the wedge of more than two basis 1-forms?)

The Hodge star is also particularly easy to express on surfaces. Recall the basic idea behind the Hodge star: in n-dimensions, we can specify any k-dimensional linear subspace via a complementary $(n-k)$-dimensional subspace. For instance, we can describe a plane in $\mathbb{R}^3$ either by two basis vectors, or by a single normal vector. On surfaces, the most interesting case is perhaps the Hodge star on 1-forms. Roughly speaking, any 1-form α can also be specified via an orthogonal 1-form $\star\alpha$ of equal length:



Look familiar? At this point it becomes clear that, at least on surfaces, the Hodge star on 1-forms is closely connected to the conformal structure $\mathcal{J}$. More precisely, if α is a 1-form on a surface M then

we can define the 1-form Hodge star $\star$ via

$$\star\alpha(X) := \alpha(\mathcal{J}X)$$

for any tangent vector field X. In other words, applying $\star\alpha$ to a vector X is the same as applying α to the rotated vector $\mathcal{J}X$. The Hodge star on 2-forms can also be expressed via the conformal structure. In particular, let ω be any 2-form on a surface M, and let X be any unit vector field. Then we have

$$\star\omega := \omega(X, \mathcal{J}X).$$

In other words, by using our 2-form to measure a little square of unit area, we can determine the associated "scale factor," which is just a scalar function on the surface (i.e., a 0-form).

Notice that we've adopted a particular convention here: there are two equal and opposite directions orthogonal to α, and we could have just as easily adopted the convention that $\star\alpha(X) = -\alpha(\mathcal{J}X)$ (many authors do!). An important thing to be aware of is how this choice affects our expression for the inner product.

EXERCISE 7.1

Like the inner product for vectors, functions, or vector fields, the inner product on 1-forms captures the notion of how well two 1-forms "line up." Any such inner product should be *positive-definite*, i.e., we should have $\langle\langle\alpha, \alpha\rangle\rangle \geq 0$ for any 1-form α. Show that the inner product

$$\langle\langle\alpha, \beta\rangle\rangle = \int_M \star\alpha \wedge \beta.$$

on real-valued 1-forms α, β is positive-definite only if we adopt the convention $\star\alpha(X) = \alpha(\mathcal{J}X)$. Likewise, show that the inner product

$$\langle\langle\alpha, \beta\rangle\rangle = \int_M \alpha \wedge \star\beta.$$

is positive-definite only if we adopt the convention $\star\alpha(X) = -\alpha(\mathcal{J}X)$. *Hint: evaluate the expressions* $\star\alpha \wedge \alpha(X, \mathcal{J}X)$ *and* $\alpha \wedge \star\alpha(X, \mathcal{J}X)$.

Throughout we will adopt the former convention ($\star\alpha(X) := \alpha(\mathcal{J}X)$), and will use double bars $||\cdot||$ to denote the corresponding norm, i.e.,

$$||\alpha|| := \sqrt{\langle\langle\alpha, \alpha\rangle\rangle}$$

EXERCISE 7.2

Show that the Hodge star preserves the norm of any 1-form α, i.e.,

$$||\star\alpha|| = ||\alpha||.$$

What's the geometric intuition?

7.3.1. Complex Differential Forms. In general, a k-form is a multilinear map from k vectors to a scalar. However, a "scalar" does not have to be a real number. For instance, when we looked at the *area vector*, we viewed the map $f : M \to \mathbb{R}^3$ as an $\mathbb{R}^3$-valued 0-form, and its differential $df : TM \to \mathbb{R}^3$ as an $\mathbb{R}^3$-valued 1-form. Likewise, we can also talk about *complex*-valued k-forms,

i.e., functions taking k vectors to a single complex number. In the complex setting, it becomes difficult to interpret k-forms as objects measuring k-dimensional volumes (what's a "complex" volume?), but we still retain all the same algebraic properties (multilinearity, antisymmetry, etc.). We also have a few new tools at our disposal.

EXERCISE 7.3

Let $z = a + bi$ be any complex number. The *complex conjugate* of z is the number $\bar{z} := a - bi$. Show that for any two complex numbers $u, v \in \mathbb{C}$ we have

$$\bar{u}v = u \cdot v + (u \times v)i$$

where on the right-hand side we interpret u, v as vectors in $\mathbb{R}^2$. *Hint: expand the left-hand side in components. Remember that $i^2 = -1$!*

EXERCISE 7.4

In the real setting, inner products are *symmetric*: we can exchange the two arguments without affecting the result. In the complex setting, the analogous concept is that an inner product is *Hermitian*: changing the arguments only *conjugates* the result. In other words, for any Hermitian inner product $\langle \cdot, \cdot \rangle$ on complex numbers, we have

$$\langle u, v \rangle = \overline{\langle v, u \rangle}.$$

Show that the inner product

$$\langle u, v \rangle := \bar{u}v$$

introduced in the previous exercise is Hermitian and positive-definite. *Hint: use the formula you just derived!*

Just as we can conjugate complex numbers, we can conjugate complex-valued 1-forms. As one might expect, this operation simply flips the imaginary part of the result:

$$(\bar{\alpha})(X) := \overline{(\alpha(X))}.$$

Similarly, we can define $(\overline{\alpha \wedge \beta})(X, Y) := \overline{(\alpha \wedge \beta(X, Y))}$, in which case $\overline{\alpha \wedge \beta} = \bar{\alpha} \wedge \bar{\beta}$ (why?).

EXERCISE 7.5

Let α, β be complex 1-forms on M. Show that the inner product

$$\langle\langle \alpha, \beta \rangle\rangle := \operatorname{Re} \int_M \star\bar{\alpha} \wedge \beta$$

is Hermitian and positive-definite. *Hint: evaluate the integrand on a basis $(X, \mathcal{J}X)$; the second part of your proof should be very similar to the real case.*

7.4. Conformal Parameterization

At long last we have all the tools we need to describe our algorithm for conformal parameterization. Remember that we want to find a map $z : M \to \mathbb{C}$ that satisfies the Cauchy-Riemann

equation

$$dz(\mathcal{J}X) = idz(X)$$

for all tangent vectors X. If we interpret dz as a complex-valued 1-form, we can rewrite this relationship as just

$$\star dz = idz.$$

Note that the geometric meaning of this statement hasn't changed: the map $\star dz$ rotates its argument by 90 degrees *before* mapping it to the plane; idz rotates vectors by 90-degrees *after* mapping them into the plane. Ultimately, angles are preserved. We can measure the *failure* of a map to be conformal by measuring the total difference between the expression on the left and the expression on the right:

$$E_C(z) := \tfrac{1}{4}||\star dz - idz||^2.$$

The quantity $E(z)$ is called the *conformal energy*. To compute a conformal map, then, we just need to solve a simple convex quadratic optimization problem

$$\min_z E_C(z),$$

subject to appropriate constraints. First, however, we're going to rewrite this energy in terms of familiar objects like the Laplacian—this formulation will make it particularly simple to setup and solve our optimization problem in the discrete setting.

EXERCISE 7.6

Let u, v be complex functions on M. Assuming that the normal derivative of either function vanishes along the boundary, show the first Green's identity

$$\langle\langle du, dv\rangle\rangle = \langle\langle \Delta u, v\rangle\rangle$$

where $\Delta = -\star d \star d$ is the Laplace-Beltrami operator on 0-forms. *Hint: you already proved this fact in the real case!*

EXERCISE 7.7

Let z be a map from a topological disk M to the complex plane $\mathbb{C}$. Show that the total signed area $\mathcal{A}(z)$ of the region $z(M) \subset \mathbb{C}$ can be expressed as

$$\mathcal{A}(z) = -\frac{i}{2}\int_M d\bar{z} \wedge dz.$$

EXERCISE 7.8

Assuming that z has vanishing normal derivatives along the boundary, show that the conformal energy $E_C(z)$ can be expressed as

$$E_C(z) = E_D(z) - \mathcal{A}(z),$$

where the first term is the *Dirichlet energy* $E_D(z) := \frac{1}{2}\langle\langle \Delta z, z\rangle\rangle$. *Hint: use the results of the last two exercises!*

EXERCISE 7.9

Suppose z is a piecewise linear map on a simplicial disk, *i.e.*, we have one value of z per vertex. Starting with the formula you derived in Exercise 7, show that the signed area of the image $z(M)$ can be expressed as the sum

$$\mathcal{A}(z) = -\frac{i}{4} \sum_{e_{ij} \in E_\partial} \bar{z}_i z_j - \bar{z}_j z_i.$$

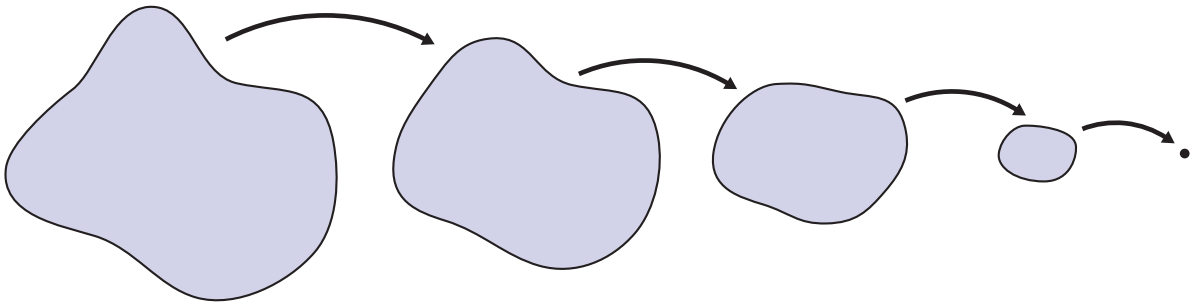
where E_∂ denotes the set of oriented boundary edges. *Hint: first, use Stokes' theorem. Next, break up the integral into an integral over each edge. Finally, think of dz as the pushforward of the edge vectors.*

CODING 23. Implement the method `ConformalParameterization::buildEnergy()`, which builds a $|V| \times |V|$ matrix corresponding to the conformal energy E_C. For the Dirichlet energy, you can reuse the expression you previously derived for the discrete Laplace operator (*i.e.*, the *cotan formula*)—the only difference is that these values are now the entries of a complex matrix rather than a real one (`SparseMatrix<Complex>`). For the area term, subtract the expression your derived in Exercise 9 from the Laplacian. You may find it easiest to simply iterate over the edges of the virtual boundary face (`Mesh::boundaries.begin()`).

Great. So to compute a conformal map we just have to know how to discretize the Laplacian Δ (which already did while studying the Poisson equation) and the signed area $\mathcal{A}$. However, let's take another look at our optimization problem—originally we said we want to solve

$$\min_z ||\star dz - i dz||^2.$$

There's a glaring problem with this formulation, namely that any *constant* map $z(p) \equiv z_0 \in \mathbb{C}$ is a global minimizer. In other words, if we map the whole surface M to a single point z_0 in the complex plane then the conformal energy is zero. (Why? Because the derivative dz is zero everywhere!) Intuitively, we can imagine that we're trying to stretch out a tiny sheet of elastic material (like a small piece of a rubber balloon) over a large region in the plane. If we don't nail this sheet down in enough places, it will simply collapse into a point:



We therefore need to add some additional constraints to make the solution more "interesting." But which constraints should we use? If we use too *few* constraints, the solution may still be uninteresting—for instance, if we just nail our elastic sheet to a single point, it can still collapse around that point. If we use too *many* constraints, there may be no solution at all—in other words, there may be no perfectly conformal map ($\star dz = i dz$) that satisfies all our constraints

simultaneously. To understand this situation better, let's take another look at *harmonic* functions and how they relate to holomorphic maps.

EXERCISE 7.10

Recall that a function is *harmonic* if it sits in the kernel of the Laplace-Beltrami operator Δ. Show that any holomorphic map $z : M \to \mathbb{C}$ is harmonic. *Hint: use the Cauchy-Riemann equation and the expression for Laplace-Beltrami you derived in the homework on vertex normals.*

Another way to investigate the relationship between harmonic and holomorphic functions is to consider our optimization problem

$$\min_z E_C(z) = \tfrac{1}{2}\langle\langle \Delta z, z \rangle\rangle - \mathcal{A}(z).$$

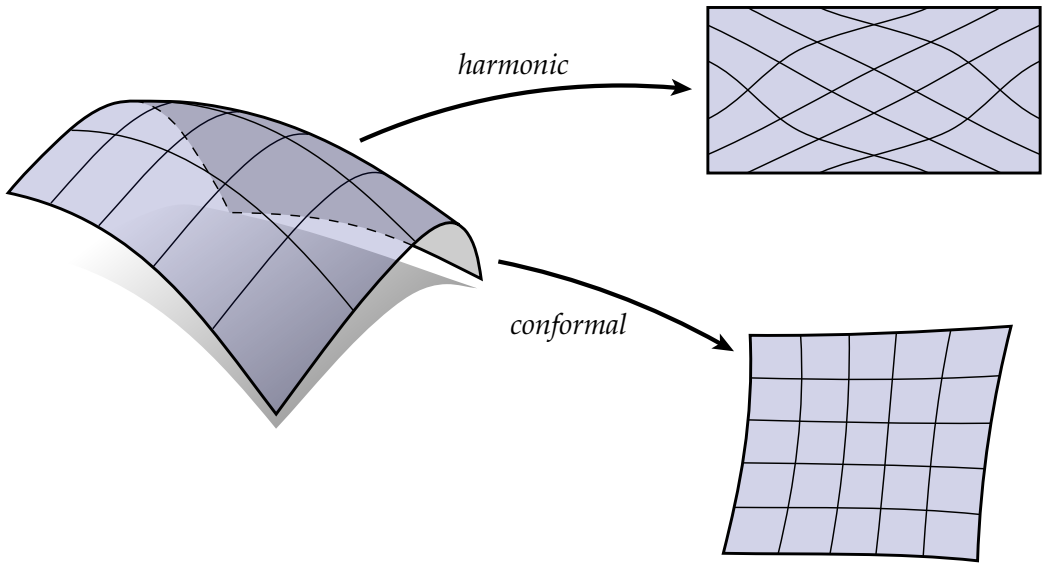
What does the minimizer look like? Well, to make things a bit easier to analyze, let's imagine that the map z is prescribed along ∂M, *i.e.*, we "nail down" *all* the points along the boundary. From our discussion of vertex normals, you may recall that the signed area is also now fixed, since it can be expressed as a boundary integral. In other words, if we pin down the boundary then $\mathcal{A}(z)$ evaluates to the same constant for all maps z, and so we need only consider the map that minimizes the Dirichlet energy $E_D(z) = \frac{1}{2}\langle\langle \Delta z, z \rangle\rangle$. In particular, since E_D is positive and quadratic, its minimum will be found wherever its gradient vanishes, *i.e.*, wherever

$$\Delta z = 0.$$

In conclusion: the minimizer of conformal energy subject to fixed boundary conditions is *harmonic*. Is it also *holomorphic*? In other words, does it preserve angles? Sadly, no: even though every conformal map is harmonic, not every harmonic map is conformal.

EXERCISE 7.11

Let M be a topological disk and let $\varphi : M \to \mathbb{C}$ be a harmonic function ($\Delta\varphi = 0$) with zero imaginary part, *i.e.*, $\mathrm{Im}(z) = 0$. Give a simple geometric reason for why φ is not a holomorphic map. (You answer should involve prose *only*—no direct calculations!)



From a practical perspective, this observation means that we can't just haphazardly nail down pieces of our rubber sheet and hope for the best—in general, a harmonic map will not preserve angles (as illustrated above, where we pin the boundary to a given rectangle). Instead, let's consider the following optimization problem:

$$\begin{aligned} \min_{z} \quad & E_C(z) \\ \text{s.t.} \quad & ||z|| = 1, \\ & \langle\langle z, \mathbb{1} \rangle\rangle = 0, \end{aligned} \tag{4}$$

where $\mathbb{1}$ denotes the constant function $z(p) \equiv 1$, i.e., the function equal to "1" at every point. What do these constraints mean geometrically? Well, suppose A is the total area of the surface $f(M)$. Then the second constraint is equivalent to

$$\frac{1}{A} \int_M z dA = 0,$$

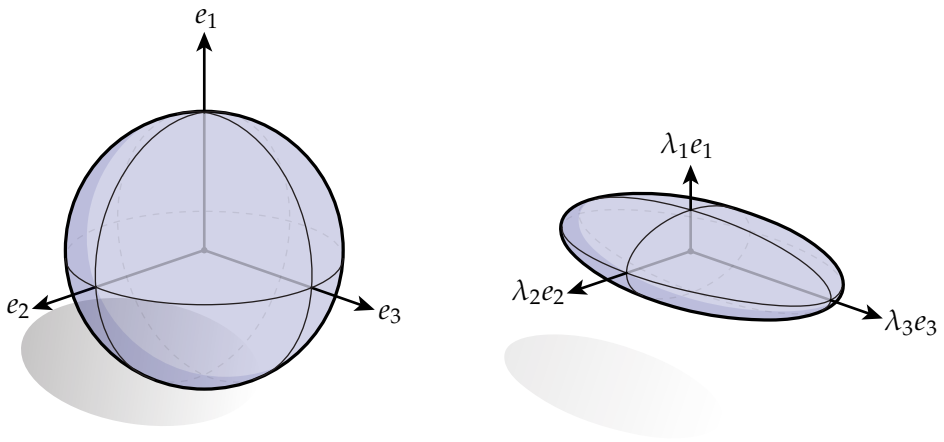
i.e., the average value of the solution is zero. Equivalently: the solution must be centered around the origin. The first constraint makes sure that the solution doesn't collapse around the origin, *i.e.*, in order for the norm to be nonzero, there must be at least one nonzero point $z(p) \neq 0$. Together, these conditions are sort of the "weakest" things we can ask for: we don't know where we want our map to go, but we sure don't want it to collapse!

7.5. Eigenvectors, Eigenvalues, and Optimization

Ok, next question: how do we actually solve an optimization problem like Equation 4? At first glance it might look rather nasty and nonlinear, but in fact problems of this form turn out to be some of the nicest ones we can hope to solve. To study the situation in more detail, let's revisit an important topic in linear algebra: *eigenvalue problems*. Often, eigenvectors and eigenvalues are introduced in the context of real, finite-dimensional matrices $A \in \mathbb{R}^{n \times n}$—in particular, we say that a unit vector $e \in \mathbb{R}^n$ is an eigenvector of A if

$$Ae = \lambda e$$

for some constant $\lambda \in \mathbb{R}$. As with all things in life, it's better if we can attach some kind of geometric meaning to this statement. One way to visualize a linear map is to apply it to a bunch of vectors in space and see how they change. For instance, suppose we apply a matrix $A \in \mathbb{R}^{3\times 3}$ to all the unit vectors in $\mathbb{R}^3$, *i.e.*, the unit sphere. What happens? For one thing, any point on the sphere corresponding to an eigenvector e_i will grow or shrink by the associated factor λ_i, but remain pointing in the same direction. What about the rest of the sphere? Well, if A is a *symmetric* matrix ($A^T = A$), then the three corresponding eigenvectors e_1, e_2, e_3 are mutually orthogonal (as you will prove in just a minute!). We can therefore visualize a symmetric linear map as a "squishing" of a sphere along these three axes, where the amount of squish is determined by the magnitude of the eigenvalues λ_i:



This picture provides a fairly decent mental image not only for real symmetric matrices, but more generally for any *self-adjoint linear operator*. A *linear operator* is just any map from one vector space to another that is, well, linear! Earlier, for instance, we looked at the Laplace operator Δ, which (roughly speaking) maps functions to the sum of their second derivatives. Functions form a vector space (you can add two functions, subtract them, multiply them by scalars, *etc.*), and the Laplacian Δ is said to be *linear* since it preserves basic vector space operations, *e.g.*,

$$\Delta(a\phi + b\psi) = a\Delta\phi + b\Delta\psi$$

for any pair of functions ϕ, ψ and scalars $a, b \in \mathbb{R}$. In the context of general linear operators, the idea of eigenvectors and eigenvalues is essentially unchanged: an *eigenfunction* of a linear operator is any function that changes only by a scalar multiple when we hit it with the operator. For instance, the constant function $\mathbb{1}$ is an eigenfunction of the Laplacian with associated eigenvalue $\lambda = 0$, since $\Delta\mathbb{1} = 0 = 0\mathbb{1}$. In the next few exercises we'll look at some important properties of linear operators and their eigenfunctions, which will help us develop algorithms for conformal maps (and other geometry processing problems).

EXERCISE 7.12

Let A be a linear operator and let $\langle\langle \cdot, \cdot \rangle\rangle$ be a Hermitian inner product. The *adjoint* of A, denoted by A^* is the unique linear operator satisfying

$$\langle\langle Ax, y \rangle\rangle = \langle\langle x, A^* y \rangle\rangle$$

for all vectors x, y. An operator is called *self-adjoint* if $A^* = A$. Show that all the eigenvalues of a self-adjoint operator are real.

EXERCISE 7.13

Let A be a self-adjoint linear operator. Show that any two eigenfunctions e_i, e_j of A with distinct eigenvalues λ_i, λ_j must be orthogonal, *i.e.*, $\langle\langle e_i, e_j \rangle\rangle = 0$.

If you've had much experience with real symmetric matrices, these facts should look familiar: eigenvalues are real, and eigenvectors are orthogonal. We've glossed over a lot of important points here—for instance, why should eigenfunctions always exist? But the main point is that if an operator is "nice enough" (*e.g.*, the Laplacian Δ on a compact surface) it will indeed behave much like a good old-fashioned matrix. This relationship is particularly valuable in geometry processing, where we would ultimately like to approximate continuous, infinite-dimensional linear operators with discrete, finite-dimensional matrices which we can actually store on our computer. For the moment, this way of thinking will help us develop an algorithm for solving our optimization problem above.

EXERCISE 7.14

Let $\mathsf{A} \in \mathbb{R}^{n \times n}$ be a real symmetric positive-definite matrix, *i.e.*, $\mathsf{x}^\mathsf{T}\mathsf{A}\mathsf{x} \geq 0$ for all x. Show that a solution to the optimization problem

$$\begin{aligned} \min_{\mathsf{x}} \quad & \mathsf{x}^\mathsf{T}\mathsf{A}\mathsf{x} \\ \text{s.t.} \quad & ||\mathsf{x}|| = 1 \end{aligned}$$

is given by any solution to the eigenvalue problem

$$\mathsf{A}\mathsf{x} = \lambda \mathsf{x},$$

where x is a unit eigenfunction and $\lambda \in \mathbb{R}$ is the smallest eigenvalue of A. Moreover, show that the minimal value itself is the eigenvalue λ. *Hint: note that the constraint* $||\mathsf{x}|| = 1$ *is equivalent to the constraint* $\langle\langle \mathsf{x}, \mathsf{x} \rangle\rangle = 1$ *and use the method of Lagrange multipliers.*

In other words, our optimization problem (Equation 4) reduces to a standard eigenvalue problem. So, how do we solve an eigenvalue problem? In general, there are many fascinating eigenvalue algorithms with fancy names and very, very complicated descriptions. Fortunately for us, when looking for just the *extreme* eigenvalues of a matrix (*i.e.*, the biggest or the smallest) we can often do just as well by using the stupidest algorithm imaginable. That algorithm is called the *power method*.

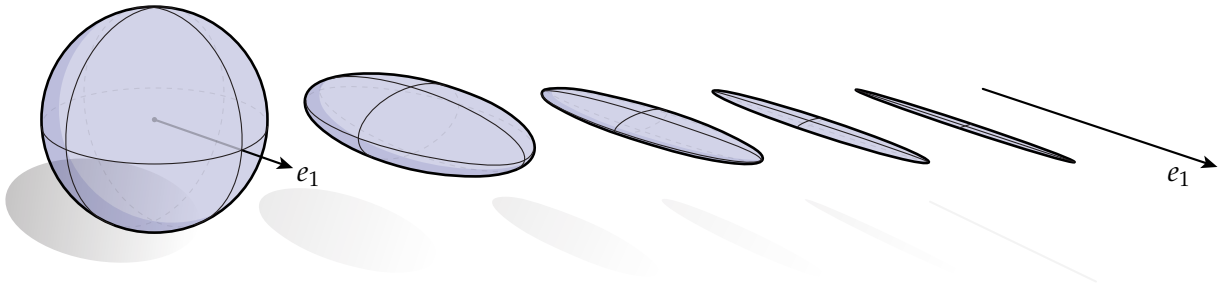
EXERCISE 7.15

The Power Method. Let $\mathsf{A} \in \mathbb{R}^{n \times n}$ be a real symmetric matrix with distinct, nonzero eigenvalues $0 < \lambda_1, \ldots, \lambda_n$, and corresponding eigenvectors $\mathsf{x}_1, \ldots, \mathsf{x}_n$. Consider the iterative procedure

$$\mathsf{y} \leftarrow \mathsf{A}\mathsf{y},$$

i.e., we just repeatedly hit the vector y with the matrix A. Show that the unit vector $\mathsf{y}/|\mathsf{y}|$ converges to the eigenvector x_n corresponding to the largest eigenvalue λ_n, as long as y is not initially orthogonal to x_n. *Hint: express* y *as a linear combination of the eigenvectors.*

To keep things simple, we made the assumption that A has distinct, nonzero eigenvalues, but in general the same principle applies: hit a vector over and over again with a matrix and you'll end up with a vector parallel to the "largest" eigenvector, *i.e.*, the eigenvector corresponding to the largest eigenvalue. (How far can you generalize your proof?) Geometrically, we can imagine that our unit sphere is gets squashed more and more until it ends up looking like a thin spindle along the direction of the largest eigenvector:



Notice that this scheme gives us the *largest* eigenvalue (and its associated eigenvector). To find the *smallest* eigenvalue we need only make a slight modification.

EXERCISE 7.16

Let e be an eigenfunction of an invertible linear operator A, with associated eigenvalue λ. Show that

$$A^{-1}e = \tfrac{1}{\lambda}e,$$

i.e., show that e is also an eigenfunction of the inverse, but has the reciprocal eigenvalue. Explain why this relationship makes sense geometrically, in light of the picture at the beginning of this section.

To get the smallest eigenvalue, then, we can simply apply the power method using the inverse matrix A^{-1} instead of the original matrix A. (Why?) In practice, however, we don't want to explicitly *compute* the inverse matrix A^{-1}, for two very important reasons:

(1) computing the inverse of a matrix is, in general, numerically unstable and,
(2) even very sparse matrices can have very dense inverses (*e.g.*, a sparse matrix that takes up $\sim$100MB of memory might have an inverse that takes up $\sim$10GB of memory!).

Instead of inverting A and iteratively applying it to some initial vector, we can just solve a linear system at each step:

ALGORITHM 1 (The Inverse Power Method).

Require: Initial guess y_0.
1: **while** Residual$(\mathsf{A}, \mathsf{y}_{i-1}) > \epsilon$ **do**
2: Solve $\mathsf{A}\mathsf{y}_i = \mathsf{y}_{i-1}$
3: $\mathsf{y}_i \leftarrow \mathsf{y}_i / |\mathsf{y}_i|$
4: **end while**

The function "Residual" measures how far our current guess y is from being an eigenvector. Recalling that $\mathsf{e}^\mathsf{T}\mathsf{A}\mathsf{e} = \lambda$ for any eigenvector e and eigenvalue λ (Exercise 14), we could write this function as

$$\text{Residual}(\mathsf{A}, \mathsf{y}) := \mathsf{A}\mathsf{y} - (\mathsf{y}^\mathsf{T}\mathsf{A}\mathsf{y})\mathsf{y},$$

being careful to note that y is a unit vector.

CODING 24. Implement the routine

```
smallestEig( const SparseMatrix<T>& A, const DenseMatrix& x ),
```

which can be found in `src/SparseMatrix.inl`. To compute the residual, you may call the subroutine `residual(A,y)`.

CODING 25. Implement the routine

```
residual( const SparseMatrix<T>& A, const DenseMatrix& x ),
```

which can be found in `src/SparseMatrix.inl`.

We now have a concrete procedure for solving eigenvalue problems. Can we use it to compute a conformal map? Well, remember that our optimization problem (Equation 4) involves two constraints: we want our solution to have unit L^2 norm $||z|| = 1$, and we want it to be orthogonal to any constant function ($\langle\langle z, \mathbb{1} \rangle\rangle = 0$). Both of these constraints demand minor modifications to our existing eigenvalue algorithm.

First, we have the constraint $||z|| = 1$, which says that we want the function $||z||$ to have unit norm. What does it mean for a discrete function to have unit norm? It *doesn't* mean that the Euclidean norm $\sqrt{|z_1|^2 + \cdots + |z_n|}$ equals one (notice, for instance, that this quantity does not take into account the fact that triangles in our mesh may have very different sizes). Instead, we should remember that the discrete collection of values z_i actually determine a continuous, piecewise linear function defined everywhere on our surface (Section 6.2). Hence, what we're really asking for here is a function with unit L^2 norm, i.e., we want to ensure that $\int_M |z|^2 \, dA$ equals one. In general we can write this kind of condition using a *mass matrix*, which expresses the L^2 norm of a function in terms of its degrees of freedom. In other words, we want to cook up a matrix $\mathsf{B} \in \mathbb{C}^{|V| \times |V|}$ such that $z^H \mathsf{B} z = \int_M |z|^2 \, dA$. Actually, in this case we've already derived the matrix B: the norm of a 0-form can be written as $||z|| = z \wedge \star z$ (Section 4.5.3), and the discrete Hodge star on primal 0-forms is just a diagonal $|V| \times |V|$ matrix $\star_0$ whose entries are equal to one-third the area of all triangles incident on the corresponding vertex (Section 4.8.4). Hence, our unit-norm constraint can be expressed as $z^H \star_0 z = 1$.

Ok, but how do we enforce this constraint in our eigensolver? We now have a problem of the form

$$\mathsf{A}\mathsf{x} = \lambda \mathsf{B}\mathsf{x},$$

called a *generalized eigenvalue problem*. In the case where A is symmetric and B is symmetric positive-definite, one can (and should!) easily show, using similar methods to those used above, that the (generalized) eigenvalues λ are still real and the (generalized) eigenvectors x are orthogonal *with respect to* B, i.e., $\mathsf{x}_i^H \mathsf{B} \mathsf{x}_j = \delta_{ij}$. Likewise, if we consider the matrix $\mathsf{C} := \mathsf{B}^{-1}\mathsf{A}$ then we get a standard eigenvalue problem $\mathsf{C}\mathsf{x} = \lambda \mathsf{x}$ where C is self-adjoint with respect to B, and all of our earlier reasoning about the power method carries through. But since we'd rather not invert B explicitly, we get a slightly modified version of the (inverse) power method:

ALGORITHM 2 (Generalized Inverse Power Method).

Require: Initial guess y_0.
1: **while** Residual$(\mathsf{A}, \mathsf{B}, \mathsf{y}_{i-1}) > \epsilon$ **do**
2: $\quad \mathsf{w} \leftarrow \mathsf{B}\mathsf{y}_{i-1}$
3: $\quad$ Solve $\mathsf{A}\mathsf{y}_i = \mathsf{w}$
4: $\quad \mathsf{y}_i \leftarrow \mathsf{y}_i / \sqrt{\mathsf{y}_i^H \mathsf{B}\mathsf{y}_i}$
5: **end while**

In other words, before each iteration we simply hit the previous iterate y_{i-1} with the mass matrix B (can you formally justify this scheme? It should involve just a simple algebraic manipulation of the original scheme!). Notice that we also divide by the norm *with respect to the mass matrix* rather than the usual Euclidean norm.

Ok, so what about the other constraint: $\langle\langle \mathsf{z}, \mathbb{1} \rangle\rangle = 0$? Once again we should recognize that the inner product $\langle\langle \cdot, \cdot \rangle\rangle$ is the L^2 inner product—not the Euclidean inner product—and adjust our algorithm accordingly. In particular, we simply need to add the line

$$\mathsf{y}_i \leftarrow \mathsf{y}_i - \overline{(\mathsf{y}_i^H \mathsf{B}\mathbb{1})}\mathbb{1}$$

to our generalized eigenvalue algorithm, right before the normalization step. In other words, we project y onto $\mathbb{1}$, and remove that component from our current iterate.

CODING 26. Modify the routine

```
smallestEig( const SparseMatrix<T>& A, const SparseMatrix<T>& B, const
DenseMatrix& x )
```

to enforce the constraints $\langle \mathsf{y}, \mathbb{1} \rangle = 0$ and $||\mathsf{y}|| = 1$. Make sure to use the appropriate mass matrix, *not* just the Euclidean norm. Also, be sure to perform the projection *before* normalization, to ensure that the final solution has unit norm upon termination of the loop.

CODING 27. Implement the routine

```
ConformalParameterization::update(),
```

which computes a conformal parameterization of a simplicial disk by computing the first non-trivial eigenvector of the conformal energy. This routine should call your previous two routines, `buildEnergy()` and `smallestEig()`. It should also copy the resulting values from the eigenvector to the texture coordinates `Vertex::texcoord` associated with each vertex. (Once this method has been implemented, you should be able to successfully flatten meshes from the Viewer.)

7.5.1. (Smallest) Eigenvalue Problems are Easy! As mentioned above, there are a lot of eigenvalue algorithms out there—many of which are far more sophisticated than the simple iterative scheme we describe above. Can't we do better by using something "more advanced?" Believe it or not, for many standard geometry processing tasks (such as conformal parameterization) the answer is, "probably not!" At least, not if we make a slight modification to our implementation of the (inverse) power method.

In particular, our current implementation of the power method solves a linear system for each iteration. However, we can save an enormous amount of work by taking advantage of *prefactorization*. Roughly speaking, prefactorization decomposes our initial matrix A into a *product* of matrices that allow us to very quickly compute the solution to a system of the form $Ax = b$. (For instance, it is always possible to write a square matrix A as the product of a *lower triangular* matrix L and an *upper triangular* matrix U, after which point we can quickly solve a system by applying forward- and back-substitution.) From a practical perspective, a prefactorization allows us to quickly solve a sequence of linear systems where the matrix A stays fixed but the right-hand side b changes ($Ax = b_1$, $Ax = b_2$, ...). Of course, "a sequence of systems with a changing right-hand side" sounds a lot like the (inverse) power method!

CODING 28. **Prefactorization.** Modify your implementation of the inverse power method to take advantage of prefactorization. In particular, you can create a `SparseFactor` object and prefactor the matrix A via the method `SparseFactor::build()`. To solve a linear system using this factorization, use the method `backsolve()` in `SparseMatrix.h`.

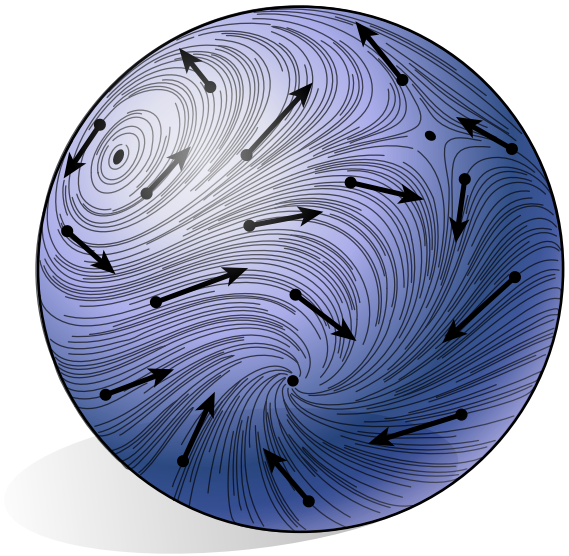
There are two important rules of thumb about matrix prefactorization in the context of geometry processing (at least when it comes to nice sparse operators like cotan-Laplace):

(1) factoring a matrix costs about as much as solving a single linear system, and
(2) the cost of backsubstitution is almost negligible compared to factorization.

The main outcome, then, is that (at least for the types of matrices we've considered in these notes) ***computing the smallest eigenvector via the power method costs about as much as solving a single linear system***. In fact, in our code framework *all* linear systems are solved by first computing a prefactorization, since (at least these days...) prefactored or *direct* linear solvers tend to be the most efficient way to solve a sparse linear system—especially if you don't have a good preconditioner. They also tend to be much more reliable than standard iterative methods like conjugate gradient, which may be very slow to converge for, *e.g.*, poor triangulations. An important exception is when working with very *large* meshes, where matrix factors may not be able to fit into memory—in this case, a simple iterative solver may be your best bet. In general, understanding the tradeoffs among different linear solvers (and other numerical tools) can make or break the effectiveness of a given geometry processing algorithm—know them well!

CHAPTER 8

Vector Field Decomposition and Design



In this chapter we look at several tools for working with *tangent vector fields* on surfaces. A tangent vector field consists of vectors lying flat along the surface—for instance, the hair on the back of your cat or the direction of the wind on the surface of the Earth. One way to describe a vector field is to simply specify each vector individually. This process is quite tedious, however, and in practice there are much more convenient ways to describe a vector field. In this assignment we're going to look at two different but closely related descriptions. First, we'll see how any tangent vector field can be expressed in terms of a *scalar potential*, *vector potential*, and a *harmonic component*, using a tool called *Helmholtz-Hodge decomposition*. Next, we'll see how a vector field can instead be expressed purely in terms of its *singularities*. Finally, we'll tie these two perspectives together and show how Helmholtz-Hodge decomposition and singularity-based editing can be combined into a highly effective tool for vector field design.

Our discussion of vector fields is closely related to the discussion of *homology* that we initiated while studying loops on surfaces. Remember that, very roughly speaking, homology is a tool that helps us detect certain "interesting features" of a space—for example, the homology generators of a surface were noncontractible loops that wrap around each handle. In the context of vector fields, we'll see a very closely related notion of *de Rham cohomology*, which helps us detect fields that also "wrap around" in a similar way. Interestingly enough, these two ideas turn out to be nearly identical in the discrete setting (the only difference is a matrix transpose!).

8.1. Hodge Decomposition

8.1.1. Breaking Up is Easy to Do. The most important tools we'll need have very little to do with geometry or topology—just good old fashioned linear algebra. In particular, we're going to show that any *chain complex* involving three vector spaces yields a decomposition of the space "in the middle" into three natural pieces. This perspective will ultimately help us make sense of objects in both the continuous and discrete setting.

First, let's recall some important ideas from linear algebra. In particular, let $A : U \to V$ be a linear map between vector spaces U and V, and let $\langle\langle , \rangle\rangle$ be an inner product on V. The *adjoint* A^* of A is the unique linear operator $A^* : V \to U$ such that

$$\langle\langle Au, v\rangle\rangle = \langle\langle u, A^*v\rangle\rangle$$

for all vectors $u \in U$ and $v \in V$. For instance, in the case where the operator A can be represented by a real matrix, its adjoint A^* is the matrix transpose. There are a few natural spaces associated with any linear operator. One is the *image*, consisting of vectors in V that can be obtained by applying A to some vector in U:

$$\text{im}(A) := \{v \in V | Au = v, u \in U\}.$$

A complementary idea is the *cokernel*, consisting of vectors in V that *cannot* be obtained as the image of some vector in U:

$$\text{coker}(A) := \text{im}(A)^\perp.$$

Here the symbol $^\perp$ denotes the *orthogonal complement*, i.e., all vectors in V orthogonal to $\text{im}(A)$. Finally, we have the *kernel*, consisting of all vectors u that get mapped to the trivial vector $0 \in V$:

$$\ker(A) := \{u \in U | Au = 0\}.$$

EXERCISE 8.1

Show that the cokernel of a linear operator is the same as the kernel of its adjoint.

In the context of surfaces, we found "interesting" loops (*i.e.*, the noncontractible ones) by looking for subsets that had an empty boundary but were not themselves the boundary of a larger set. This basic idea can be applied in lots of other settings—in particular, consider a sequence of vector spaces U, V, W connected by two maps $A : U \to V$ and $B : V \to W$. A common shorthand for such a sequence is

$$U \xrightarrow{A} V \xrightarrow{B} W.$$

We say that this sequence is a *chain complex* if

$$B \circ A = 0,$$

i.e., if any vector mapped into V by A subsequently gets "killed" when we apply B. An interesting question we can ask, then, is, "are there additional elements of V that get killed by B for some other reason?" As with loops on surfaces, answering this question helps us unearth interesting stuff about the space V.

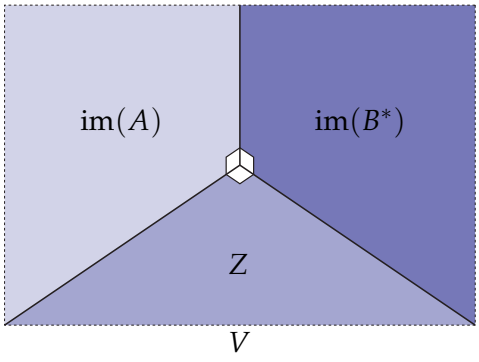
EXERCISE 8.2

The intersection $\cap$ of two linear subspaces is the collection of all vectors common to both of them. Show that

$$\mathrm{im}(A) \cap \mathrm{im}(B^*) = 0,$$

for any chain complex $U \xrightarrow{A} V \xrightarrow{B} W$. In other words, show that the maps A and B^* don't produce any common vectors except for the zero vector. *Hint: use the condition $B \circ A = 0$ and the result of the previous exercise.*

Finally, we get to the punchline. If things are starting to feel a bit abstract at this point, don't dismay! The meaning of this decomposition will become quite clear in the next section, when we apply it to tangent vector fields on surfaces.



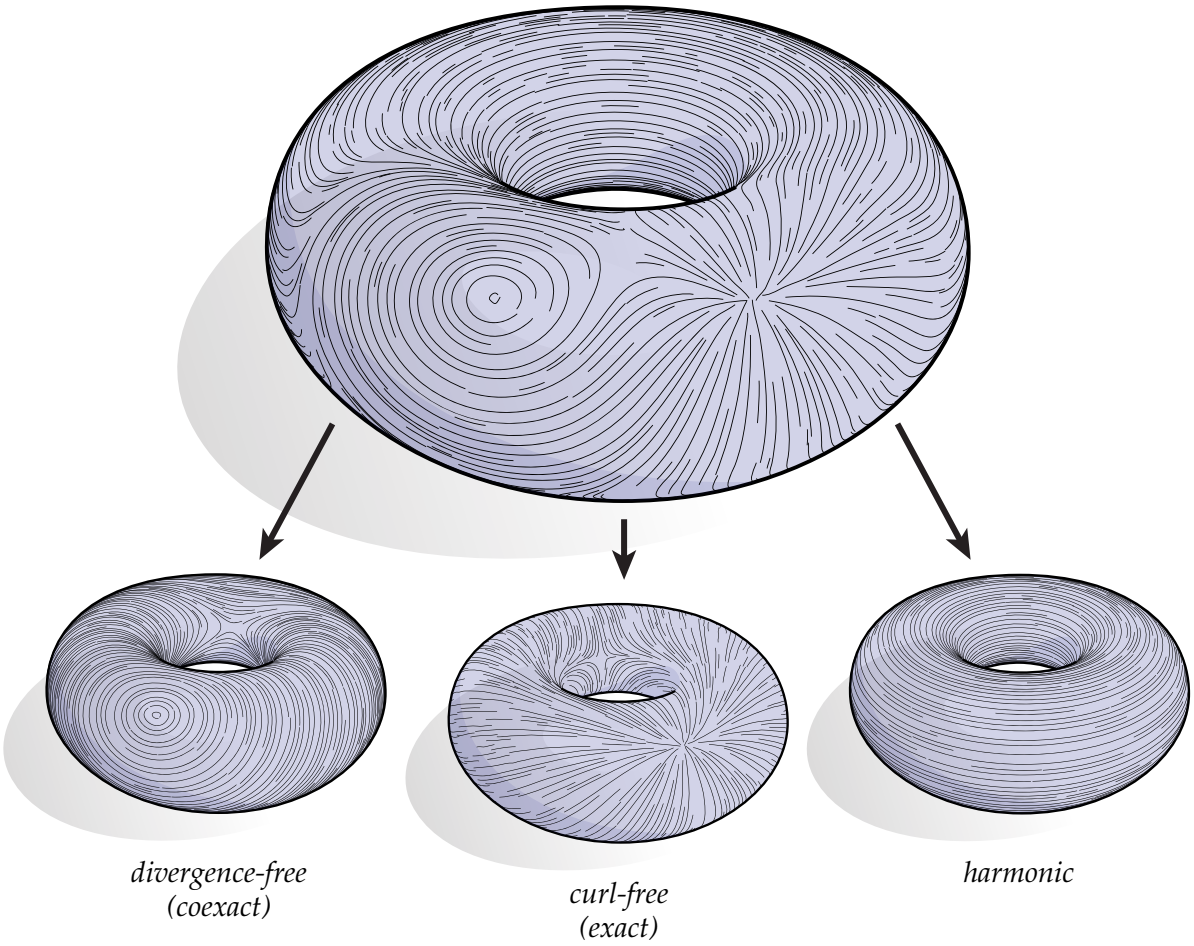
EXERCISE 8.3

Show that any vector $v \in V$ can be expressed as

$$v = Au + B^*w + z$$

for some triple of vectors $u \in U$, $w \in W$ and $z \in V$ such that $A^*z = 0$ and $Bz = 0$. Equivalently, show that the vector space U can be decomposed into the three orthogonal subspaces $\mathrm{im}(A)$, $\mathrm{im}(B^*)$, and $Z := \ker(B) \setminus \mathrm{im}(A)$, where the backslash denotes the set of all vectors in $\ker(B)$ that are not in $\mathrm{im}(A)$ (excluding the origin). *Hint: start with the two pieces of V you considered in the previous exercise, and consider what the rest looks like. It may be helpful to recall some basic facts about unions, intersection, and complements,* e.g., $(V_1 \cup V_2)^\perp = V_1^\perp \cap V_2^\perp$.

8.1.2. Helmholtz-Hodge Decomposition.



In any flow, certain features naturally catch the eye: wind swirling around the Great Red Spot on Jupiter, or water being sucked down into a mysterious abyss through the bathtub drain. Many of these features can be given a precise mathematical description using the decomposition we studied in the previous section. Consider, for instance, the vector field depicted above. Visually, we experience three features: a swirling spot, a sucking sink, and a steady flow around a lazy river, each of which is quite distinct. But can we apply the same kind of decomposition to *any* vector field? To answer this question, let's look at a chain complex called the *de Rham complex*, which shows up again and again in exterior calculus.

EXERCISE 8.4

Let d be the exterior derivative on k-forms and let $\delta := \star d \star$ be the associated *codifferential*, acting on $k+1$-forms. Assuming that the domain M has no boundary, show that d and δ are adjoint, *i.e.*, show that

$$\langle\langle d\alpha, \beta \rangle\rangle = \langle\langle \alpha, \delta\beta \rangle\rangle$$

for any k-form α and $(k+1)$-form β. *Hint: Stokes' theorem!*

EXERCISE 8.5

Helmholtz-Hodge Decomposition. Let Ω^k denote the space of real-valued k-forms on a closed compact n-dimensional manifold M, equipped with its usual L^2 inner product. A *de Rham complex* is a sequence [1]

$$\Omega^{k-1} \xrightarrow{d} \Omega^k \xrightarrow{d} \Omega^{k+1}.$$

Any such sequence is exact: if you recall, $d \circ d = 0$ was one of the defining properties of the exterior derivative d. Show that any k-form ω can be expressed as

$$\omega = d\alpha + \delta\beta + \gamma$$

for some $(k-1)$-form α, $k+1$-form β, and k-form $\gamma \in \Omega^k$ such that $d\gamma = 0$ and $\delta\gamma = 0$. *Hint: apply the results of Exercises 3 and 4.*

The three pieces $d\alpha$, $\delta\beta$, and γ show up often enough in exterior calculus that they are given special names. In particular, a k-form is

- *exact* if it can be expressed as the image of the exterior derivative (*e.g.*, $d\alpha$),
- *coexact* if it can be expressed as the image of the codifferential (*e.g.*, $\delta\beta$), and
- *harmonic* if it is in the kernel of both d and δ—in other words, a harmonic form is both *closed* ($d\gamma = 0$) and co-closed ($\delta\gamma = 0$).

We can gain some valuable intuition about these conditions by again considering the special case of surfaces. Recall that $d\phi = (\nabla\phi)^\flat$, *i.e.*, the exterior derivative of a 0-form ϕ looks like its gradient. In a similar vein, we have the following fact.

EXERCISE 8.6

Let β be a 2-form on a surface M. As with any volume form, we can think of β as the Hodge dual of some 0-form ϕ, *i.e.*, $\beta = \star\phi$. Show that

$$\delta\beta = ((\nabla\phi)^\perp)^\flat,$$

i.e., the codifferential of a 2-form looks like the gradient, rotated by 90 degrees in the counter-clockwise direction. *Hint: on a surface, what does the Hodge star on 1-forms look like? Remember our discussion of conformal structure.*

Therefore, in the special case of surfaces we can write any vector field X as the sum of a *curl-free* $\nabla\phi$, a *divergence-free* part $(\nabla u)^\perp$, and a *harmonic part* Y which is both curl- and divergence-free. In other words,

$$X = \nabla\phi + (\nabla u)^\perp + Y,$$

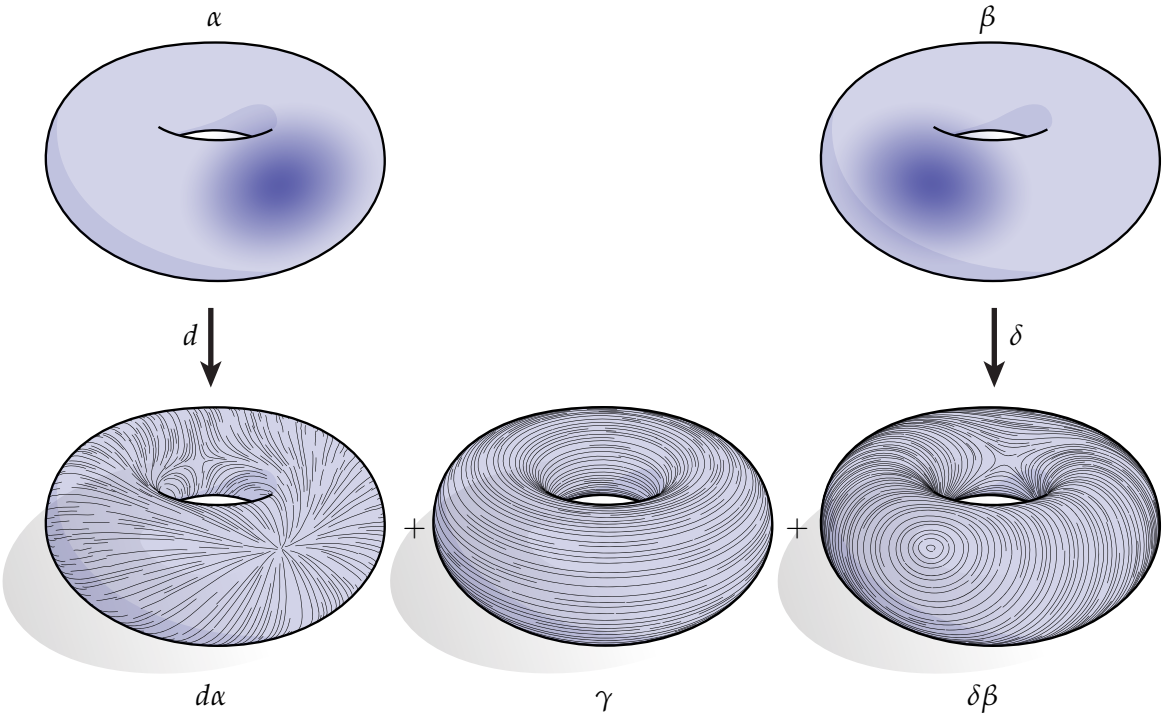
[1] Most people use the term "de Rham complex" to refer to the entire sequence going from 0-forms to n-forms, but for our purposes this little piece will suffice.

where ϕ and u are scalar functions and Y is a vector field. The corresponding de Rham complex can also be expressed in terms of vector calculus, namely

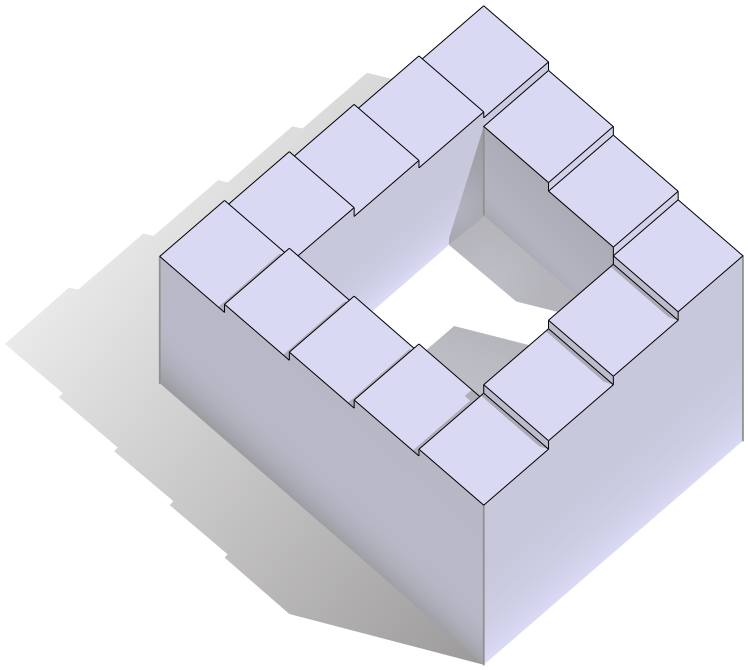
$$C^\infty \xrightarrow{\nabla} \mathfrak{X} \xrightarrow{\nabla\times} C^\infty$$

where C^∞ denotes the space of smooth, real-valued functions on M, and $\mathfrak{X}$ denotes the space of smooth vector fields.

With this interpretation in mind, we can visualize the Hodge decomposition of a 1-form on a surface via the two functions α and $\star\beta$, which are sometimes called a "scalar potential" and a "vector potential," respectively:



But wait a minute—what's the corresponding picture for the harmonic component? Actually, the *absence* of a picture is the whole point! The harmonic component is precisely the piece that *cannot* be expressed in terms of some "potential." In this example, for instance, any such potential would have to keep increasing monotonically as we walk around the torus. Of course, no periodic function can increase indefinitely, unless your name happens to be M.C. Escher:



EXERCISE 8.7

Above, we said that a k-form γ is *harmonic* if it is both closed ($d\gamma = 0$) and co-closed ($\delta\gamma = 0$). Some authors instead define harmonic k-forms as elements of the kernel of the k-form Laplacian $\Delta := d\delta + \delta d$. Show that these two definitions are equivalent, *i.e.*, show that γ is harmonic if and only if

$$\Delta\gamma = 0.$$

8.1.3. Computing a Decomposition. So far our discussion of Helmholtz-Hodge decomposition has been very ephemeral: *there exists* a decomposition of any k-form into its three constituent pieces. Unfortunately, although compilers are quite good at interpreting "if," "while," and "for each" statements, they aren't particulary adept at understanding "there exists" statements! So, how do we actually compute a decomposition of real data? Fortunately, we've already written everything down in terms of exterior calculus, which will make the translation to the discrete setting straightforward.

EXERCISE 8.8

Let A be a linear operator and let A^* be its adjoint. Show that

$$\mathrm{im}(A) \cap \ker(A^*) = 0,$$

i.e., any vector in the image of A is *not* in the kernel of A^*.

EXERCISE 8.9

Let ω be a real-valued k-form on a closed compact domain. From Exercise 5 we know that ω can be expressed as $\omega = d\alpha + \delta\beta + \gamma$ for some $k-1$-form α, $k+1$-form β, and harmonic k-form γ. Show that α and β are solutions to the linear equations

$$\delta d\alpha = \delta\omega$$

and

$$d\delta\beta = d\omega,$$

respectively. *Hint: as always, if you get stuck, think about what you did in the previous exercise!*

A practical procedure for decomposing a vector field is therefore:

ALGORITHM 3 (Helmholtz-Hodge Decomposition).

1: Solve $\delta d\alpha = \delta\omega$
2: Solve $d\delta\beta = d\omega$
3: $\gamma \leftarrow \omega - d\alpha - \delta\beta$

In other words, we compute the two potentials, then see what remains. (By the way, can you apply this procedure to any chain complex, or do we need some special structure from the de Rham complex?) The nice thing about this algorithm is that we can implement it using the discrete differential operators we've already looked at—no additional discretization required!

To be a bit more explicit, let $d_0 \in \mathbb{R}^{|E|\times|V|}$ and $d_1 \in \mathbb{R}^{|F|\times|E|}$ be the discrete exterior derivatives on 0- and 1-forms, respectively, where V, E, and F, denote the set of vertices, edges, and faces in a triangulated surface. Also let $\star_0 \in \mathbb{R}^{|V|\times|V|}$, $\star_1 \in \mathbb{R}^{|E|\times|E|}$ and $\star_2 \in \mathbb{R}^{|F|\times|F|}$ denote the diagonal Hodge star matrices computed using the circumcentric dual. (At this point you may find it useful to review the earlier material on discrete exterior calculus.) The two systems above can then be discretized as

$$\star_0^{-1} d_0^T \star_1 d_0\alpha = \star_0^{-1} d_0^T \star_1 \omega$$

and

$$d_1 \star_1^{-1} d_1^T \star_2 \beta = d_1\omega,$$

where $\alpha \in \mathbb{R}^{|V|}$ and $\beta \in \mathbb{R}^{|F|}$ are encoded as a single value per vertex and per face, respectively. Computationally, it is often more efficient if we can solve an equivalent symmetric system. In the case of the first system we can simply remove the factor $\star_0^{-1}$ from both sides, yielding

$$d_0^T \star_1 d_0\alpha = d_0^T \star_1 \omega.$$

This system is also positive-semidefinite—independent of the quality of the triangulation—since the operator $d_0^T \star_1 d_0$ on the left-hand side is simply the restriction of the positive-definite Laplace-Beltrami operator Δ to the space of piecewise linear functions (see our discussion of the Poisson equation). Therefore, we can always solve for the potential α using a highly efficient numerical method like sparse Cholesky factorization—this method is implemented in our code framework via the routine `solvePositiveDefinite`. As for the second equation, consider the change of variables $\tilde{\beta} := \star_2\beta$. We can then solve the symmetric system

$$d_1 \star_1^{-1} d_1^T \tilde{\beta} = d_1\omega$$

and recover the final solution by computing $\beta = \star_2^{-1}\tilde{\beta}$. (This final step comes at virtually no additional cost since the matrix $\star_2$ is diagonal—the inverse simply divides each entry of $\tilde{\beta}$ by a

known constant.) Unlike the first equation, this system is not always positive-semidefinite. A sufficient (though not actually necessary) condition for positive-definiteness is that the entries of the 1-form Hodge star $\star_1$ are positive. In general, however, this condition will not be met, but we can still efficiently compute a solution using LU factorization, which is implemented in our code framework via the method `solveSquare`. (An extremely rough rule of thumb is that LU factorization is about twice as costly as Cholesky, since it involves two distinct factors.)

CODING 29. Implement the methods

```
void HodgeStar0Form<T> ::  build()
void HodgeStar1Form<T> ::  build()
void HodgeStar2Form<T> ::  build()
```

in `DiscreteExteriorCalculus.inl`, which build the diagonal Hodge star matrices $\star_0$, $\star_1$, and $\star_2$, respectively. Notice that these methods are templated on a type T, which can be `Real`, `Complex`, or `Quaternion`. In all cases, however, the matrices should have real entries (*i.e.*, the imaginary parts should be zero). This setup will allow you to solve a variety of different geometry processing problems using the same collection of routines.

CODING 30. Implement the methods

```
void ExteriorDerivative0Form<T> ::  build()
void ExteriorDerivative1Form<T> ::  build()
```

in `DiscreteExteriorCalculus.inl`, which build the discrete exterior derivatives d_0 and d_1 on 0-forms and 1-forms, respectively. Check that these two matrices have been properly built by computing their product and verifying that the resulting matrix is zero.

CODING 31. Implement the methods

```
HodgeDecomposition ::  computeZeroFormPotential()
HodgeDecomposition ::  computeTwoFormPotential()
HodgeDecomposition ::  extractHarmonicPart()
```

using the matrices you built in the last two exercises. You should now be able to visualize the three components of the provided tangent vector fields from the Viewer. Compare the speed of solving the two linear systems with the generic routine `solve()` versus the more specialized routines `solvePositiveDefinite()` and `solveSquare()`, respectively.

One final question: why should this procedure work in the discrete setting? In other words, how do we know that any *discrete* vector field can be decomposed as in the smooth case? Well, as you should have verified in Coding 30, the sequence of vector spaces corresponding to the discrete exterior derivatives d_0 and d_1 is exact. Therefore, all of our previous results about Hodge decomposition still apply! In other words, it makes no difference (at least not in this case) whether we work with infinite-dimensional function spaces or finite-dimensional vector spaces. Ideally, this kind of behavior is exactly the kind of thing we want to capture in the discrete setting: our discretization should preserve the most essential structural properties of a smooth theory, so that we can directly apply all the same theorems and results without doing any additional work.

8.2. Homology Generators and Harmonic Bases

When working with surfaces of nontrivial topology, we often need to be able to compute two things: *generators* for the first homology group, and *bases* for the space of harmonic 1-forms. Loosely speaking, homology generators represent all the "basic types" of nontrivial loops on a surface. For instance, on a torus we can find two homology generators: a loop going around the inner radius, and a loop going around the outer radius—more generally, a closed surface of genus g will have

$2g$ generators. Likewise, we will have $2g$ distinct bases for the harmonic 1-forms, each circulating around a different handle in a different direction (see for example the harmonic component we extracted above via Helmholtz-Hodge decomposition). People have come up with lots of funky schemes for computing these objects on triangle meshes; here we describe the two methods that are perhaps simplest and most efficient. In fact, these methods are closely related: we first compute a collection of homology generators, then use these generators to construct a basis for harmonic 1-forms.

8.2.1. Homology Generators. The algorithm for finding generators, called *tree-cotree decomposition* **[Epp03, EW05]**, depends on nothing more than the most elementary graph algorithms. If you don't remember anything about graphs, here's a quick and dirty reminder. A *graph* is a set of vertices V and a collection of edges E which connect these vertices, *i.e.*, each edge $e_{ij} \in E$ is an unordered pair $\{v_i, v_j\}$ of distinct vertices $v_i, v_j \in V$. For example, the vertices and edges of a simplicial surface describe a graph. The faces and *dual* edges also describe a graph. A *subgraph* is a subset of a graph that is also a graph. A graph is *connected* if every vertex can be reached from every other vertex along some sequence of edges. A *tree* is a connected graph containing no cycles. If we wanted to be a bit more geometric (we are studying geometry, after all), we could say that a tree is a simplicial 1-complex that is both connected and simply-connected. If most of this stuff sounds unfamiliar to you, *go read about graphs!* They're important. And fun.

To find a set of generators, we'll need to compute a couple *spanning trees*. A spanning tree is just a tree touching all the vertices of a given graph. How do we compute a spanning tree? If you've studied graphs before, you might vaguely recall someone mumbling something like, *"Prim's algorithm... Kruskal's algorithm... $O(n \log n)$..."* These two algorithms compute *minimal* spanning trees, *i.e.*, spanning trees with minimum total edges weight. We don't care about edge weight, ergo, we don't care about Prim or Kruskal (sorry guys). Actually, we can use a much simpler *linear* time algorithm to get a spanning tree:

ALGORITHM 4 (X-First Search).

Put any vertex in a bag, marking it as visited. Until this bag is empty, pull out a vertex and put all unvisited neighbors in the bag, marking them as visited. Every time you put a neighbor in the bag, add the corresponding edge to the tree.

Pretty easy. A "bag" here could be a stack, a queue, or…whatever. In other words, you could do depth-first search. Or breadth-first search. Or *anything*-first search. The point is, we just need to visit all the vertices[2]. Oh yeah, and we'll call the initial vertex the *root*. Once we know how to compute a spanning tree, finding a collection of generators on a simplicial surface is also easy:

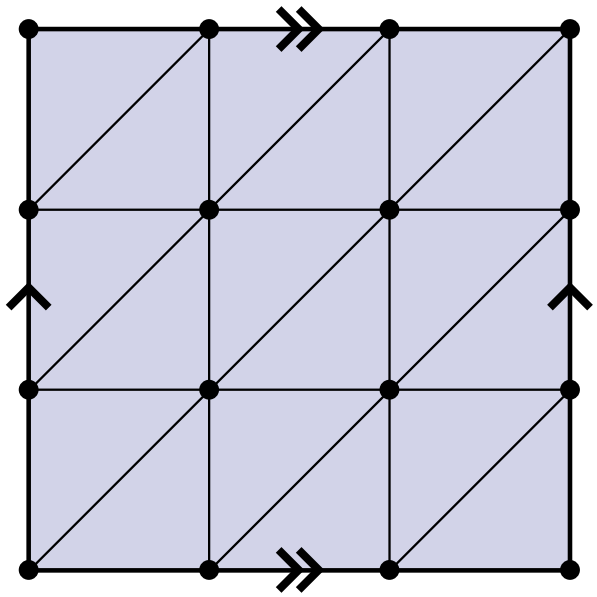
ALGORITHM 5 (Tree-Cotree).

1: Build a spanning tree $T^\star$ of dual edges.
2: Build a spanning tree T of primal edges that do not cross edges in $T^\star$.
3: For each dual edge $e^\star_{ij}$ that is neither contained in $T^\star$ nor crossed by T, follow both of its endpoints back to the root of $T^\star$. The resulting loop is a generator.

Overall, the algorithm will produce $2g$ generating loops. Instead of writing a proof, let's just get a sense for how this algorithm works in practice:

EXERCISE 8.10

Recall that the *fundamental polygon* of the torus is just a square with opposite sides glued together. Consider the following graph on the torus:



[2]This discussion inspired by Jeff.

Run the tree-cotree algorithm by hand, *i.e.*, draw a primal and dual spanning tree on this graph. How many generators did you get? *Hint: be careful about corners and edges!*

CODING 32. Implement the methods
`TreeCotree::buildPrimalSpanningTree()`,
`TreeCotree::buildDualSpanningCoTree()`, and
`TreeCotree::buildCycles()`,
which can be found in `TreeCotree.h`.

Finally, though we will not show it here, **TREE-COTREE** will work for surfaces with boundary as long as T^* contains a dual edge crossing a primal edge incident on each boundary loop.

8.2.2. Harmonic Bases. Once we have the homology generators, the harmonic 1-form bases can also be found using a rather simple procedure **[TACSD06]**. In fact, we can take advantage of our newly-acquired knowledge of Hodge decomposition. Suppose we start out with a 1-form ω that is closed but not exact. From Exercise 5, we know that ω must then have the form

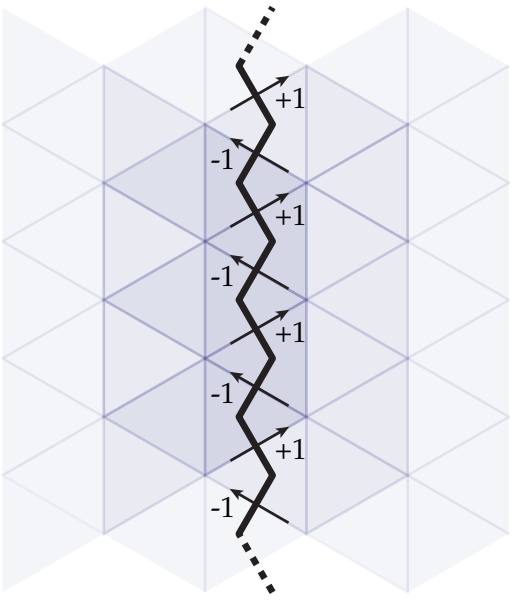
$$\omega = d\alpha + \gamma$$

for some 0-form α and harmonic 1-form γ. Using our procedure for Helmholtz-Hodge decomposition (Algorithm 8.1.3) we can easily extract just the harmonic part. In fact, since ω has no coexact component we need only solve the first equation $\delta d\alpha = \delta\omega$, or equivalently

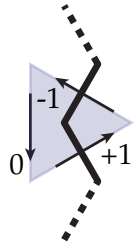
$$\Delta\alpha = \delta\omega.$$

In other words, just a standard scalar Poisson equation. We can then extract the harmonic component via $\gamma = \omega - d\alpha$ as before.

Sounds pretty good, but where did ω come from in the first place? In other words, how do we construct a 1-form that is closed but not exact? Well, once we have our generators, it's quite easy. For every edge crossing from the "left" of the generator to the "right," set ω to $+1$; for every edge crossing from the "right" to the "left," set ω to -1:



For all remaining edges, set ω to zero. The resulting 1-form is closed. Why? Well, remember that the discrete exterior derviative on 1-forms is just the (oriented) sum of edge values around each triangle. Therefore, in each triangle crossed by our generator, we get $1 - 1 + 0 = 0$:



(In all other triangles we get $0 + 0 + 0 = 0$.) Ok, so this particular choice of ω is closed. But is it also exact?

EXERCISE 8.11

Show that the 1-form ω described above is not exact, *i.e.*, it has a nonzero harmonic component. *Hint: Stokes' theorem!*

CODING 33. Implement the method `HarmonicBases::buildClosedPrimalOneForm()`, which constructs a closed discrete primal 1-form corresponding to a given homology generator. Be careful about orientation!

If you successfully completed Exercise 11, you probably noticed that ω integrates to a nonzero value along the corresponding generator ℓ. Likewise, it's not hard to verify that ω vanishes when integrated along any other generator. As a result, we can use this procedure to construct a basis for the harmonic 1-forms on a triangulated surface.

EXERCISE 8.12

Let $\ell_1, \ldots, \ell_{2g}$ be a collection of homology generators, constructed as described in Section 8.2.1. Let $\omega_1, \ldots, \omega_1$ be the corresponding closed 1-forms, and let $\gamma_1, \ldots, \gamma_n$ be the corresponding harmonic components. Show that the γ_i are linearly independent.

ALGORITHM 6 (Harmonic-Basis).

1: Compute homology generators $\ell_1, \ldots, \ell_n$ using **TREE-COTREE**.
2: **for** $i = 1, \ldots, n$ **do**
3: Construct a closed 1-form ω_i corresponding to ℓ_1.
4: Solve $\Delta\alpha_i = \delta\omega_i$.
5: $\gamma_i \leftarrow \omega_i - d\alpha_i$
6: **end for**

As discussed in the chapter on parameterization, we can greatly accelerate the process by *prefactoring* the Laplace operator. Since the cost of prefactorization is typically far greater than the

cost of backsubstitution (and since **TREE-COTREE** amounts to a simple linear traversal), the overall cost of this algorithm is roughly the same as the cost of solving a single Poisson equation.

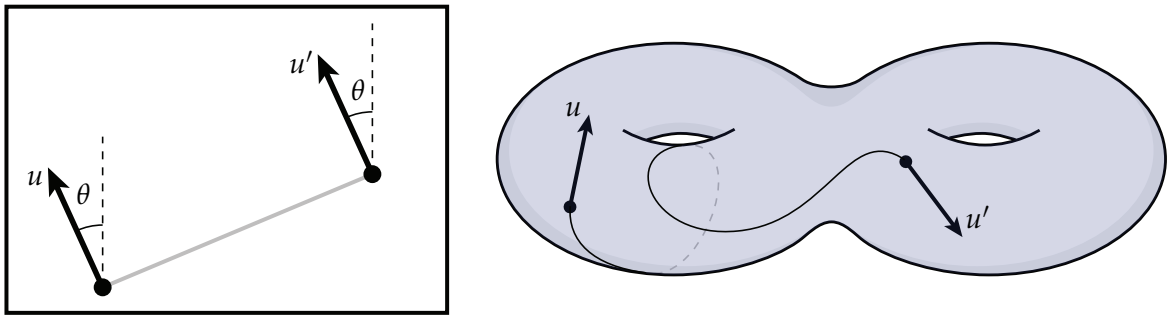
CODING 34. Implement the method `HarmonicBases::build()` which implements the algorithm **HARMONIC-BASIS** described above. You should prefactor the Laplacian using a `SparseFactor` object, and solve the Poisson equations via `backsolvePositiveDefinite()`.

8.3. Connections and Parallel Transport

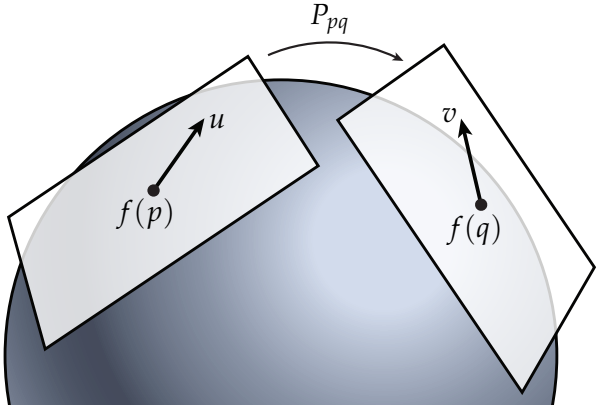
In discrete differential geometry, there are often many ways to discretize a particular smooth object. As discussed earlier, however, the hope is that we can discretize all the objects in a given *theory* so that relationships from the smooth theory still hold in the discrete picture. For instance, when we looked at Hodge decomposition we discretized the exterior derivative and the Hodge star in such a way that the Hodge decomposition has an identical expression in both the smooth and discrete world: $\omega = d\alpha + \delta\beta + \gamma$. In the case of surfaces, we represented a vector field as a *discrete 1-form*, *i.e.*, a number associated with each oriented edge giving the circulation along (or flux through) that edge.

In this section we're going to adopt a different perspective based on the theory of *connections* and *parallel transport*. This time around, we're going to represent a vector field as an *angle-valued dual 0-form*. More plainly, we're going to store an angle on each face that gives the direction of the field. Note that this representation ignores *magnitude*, so what we're really working with is a *direction field*. Before going too much further with the discrete theory, though, let's first talk about the smooth objects we want to discretize!

8.3.1. Parallel Transport.



Suppose we have a tangent vector $u = df(X)$ sitting on an immersed surface $f(M)$. How do we move from one point of the surface to another while preserving the direction of u? If $f(M)$ is completely flat (like the plane itself) then the most natural thing is to slide u from one point to the other along a straight path—keeping the angle with some reference direction fixed—to obtain a new vector u'. This process is called *parallel transport*, and the tangent vectors u and u' are, as usual, said to be *parallel*. Parallel transport on a *curved* surface is a bit trickier. If we keep u pointing in the same direction, then it ceases to be tangent and now sticks out the side. On the other hand, if we instead keep u flat against the surface then we no longer have a consistent, global reference direction. Overall, the notion of "same direction" is not very well-defined!



Still, having some notion of "same direction" could be very convenient in a lot of situations. So, rather than looking for some "natural" definition, let's define for ourselves what it means to be parallel! Ok, but how do we do that? One idea is to explicitly specify a *parallel transport map* $P_{pq} : T_pM \to T_qM$ that immediately "teleports" vectors from the tangent plane T_pM to the tangent plane T_qM. We could then say that—*by definition*—two vectors $X \in T_pM$ and $Y \in T_qM$ are parallel if $P_{pq}(X) = Y$. (Or equivalently, if the embedded vector $u := df(X)$ is the same as $v := df(Y)$). Unfortunately we'd have to specify this map for *every pair* of points p, q on our surface. Sounds like a lot of work! But we're on the right track.

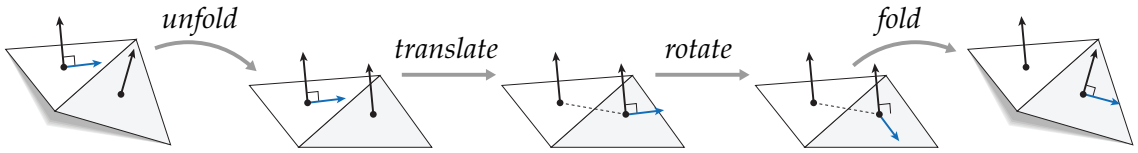
An alternative is to describe what it means for vectors to be parallel *locally*. In other words, how must a vector *change* as we move along the surface in order to remain parallel? One way to encode this information is via a *connection 1-form*, which we can express as a linear map $\omega : TM \to TM$, *i.e.*, given a direction of motion Z, the quantity $\omega(Z)$ tells us how much a vector X must change in order to remain parallel. (The reason ω is called a "connection" is because it tells us how to *connect* nearby tangent spaces, *i.e.*, how to identify tangent vectors in one space with vectors in a "nearby" space.) To get any more formal than this takes a bit of work—for now let's just make sure we have a solid geometric intuition, which should serve us well in the discrete setting:

EXERCISE 8.13

Take a stiff piece of cardboard and draw an arrow on it. Now roll it around on the surface of a basketball for a while. In effect, you're defining a map between the tangent plane where you first set down the cardboard and the tangent plane at the current location. The rolling and twisting motion you apply at any given moment effectively defines a connection (at least along a particular path). Try the following experiment. Starting at some clearly marked initial point, be very careful to note which direction your arrow points. Now roll the cardboard around for a while, eventually bringing it back to the initial point. Does the arrow point in the same direction as it did initially? What happens if you take a different path?

The phenomenon you've (hopefully) just observed is something called the *holonomy* of the connection, *i.e.*, the failure of the connection to preserve the direction of a vector as we go around a closed loop. We'll say a bit more about holonomy in just a minute.

8.3.2. Discrete Connections.

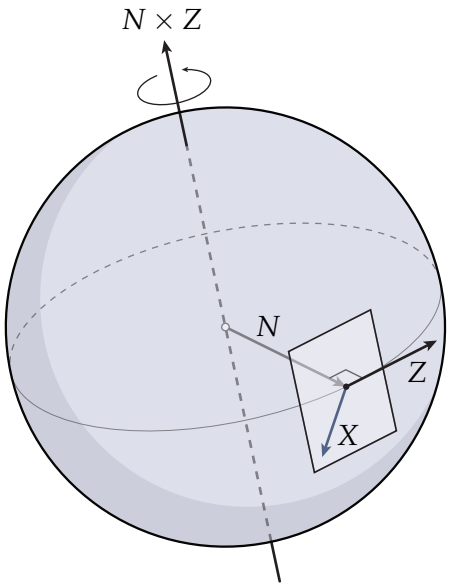


How should we specify a connection in the discrete setting? Well, for a given a pair of triangles (i, j), we can imagine rigidly unfolding them the plane, translating a vector from one to the other, applying a rotation by some small angle θ_{ij}, and then rigidly "refolding" these triangles into their initial configuration, as illustrated above. In other words, we can describe a connection on a triangle mesh via a single angle $\varphi_{ij} \in \mathbb{R}$ for each oriented *dual* edge in our mesh. We should also make sure that $\varphi_{ji} = -\varphi_{ij}$. In other words, the motion we make going from face j to face i should be the opposite of the motion from i to j. Enforcing symmetry ensures that our notion of "parallel" is consistent no matter which direction we travel. The whole collection of angles $\varphi \in \mathbb{R}^{|E|}$ is called a *discrete connection*.

By the way, does this object sound familiar? It should! In particular, we have a single number per oriented dual edge, which changes sign when we change orientation. In other words, φ is a real-valued, dual, discrete 1-form.

8.3.3. The Levi-Civita Connection. In terms of the picture above, we said that an angle $\varphi_{ij} = 0$ means "just translate; don't rotate." If we set *all* of our angles to zero, we get a very special connection called the *Levi-Civita* connection[3]. The Levi-Civita connection effectively tries to "twist" a tangent vector as little as possible as it moves it from one point to the next. There are many ways to describe the Levi-Civita connection in the smooth setting, but a particularly nice geometric description is given by Kobayashi:

THEOREM 1. ***(Kobayashi)*** *The Levi-Civita connection on a smooth surface is the pullback under the Gauss map of the Levi-Civita connection on the sphere.*



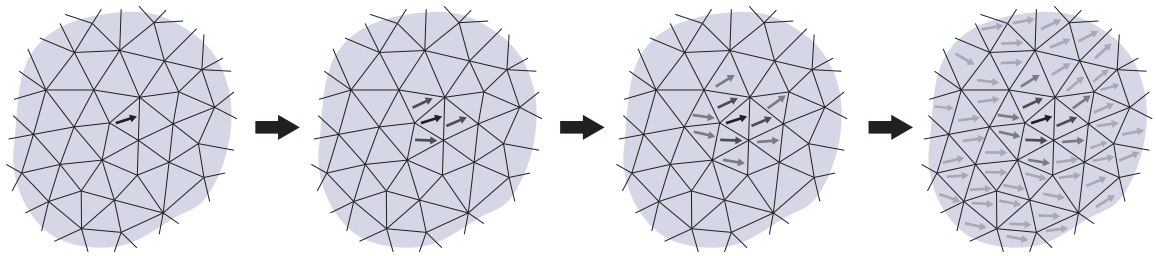
[3]Those with some geometry background should note that a discrete connection really encodes the *deviation* from Levi-Civita; it should not be thought of as the connection itself.

What does this statement mean? First, recall that the Gauss map $N : M \to S^2$ takes a point on the surface to its corresponding unit normal—this normal can also be thought of as a point on the unit sphere. And what's the Levi-Civita connection on the sphere? Well, we said that Levi-Civita tries to "twist" vectors as little as possible. On a sphere, it's not hard to see that the motion of least twist looks like a rotation of the tangent plane along a great arc in the direction Z of parallel transport. More explicitly, we want a rotation around the axis $N \times Z$, where N is the normal of our initial tangent plane. Altogether, then, Kobayashi's theorem says the following. If we start out with a tangent vector $\tilde{X}$ on our surface and want to transport it in the direction $\tilde{Z}$, we should first find the tangent plane with normal N on the sphere, and the two corresponding tangent vectors X and Z. (Extrinsically, of course, these are just the same two vectors!) We can then apply an (infinitesimal) rotation along the great arc in the direction Z, dragging X along with us.

EXERCISE 8.14

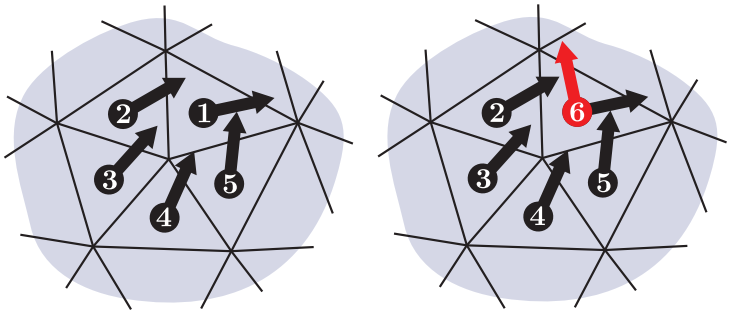
Use Kobayashi's theorem to justify the "unfold, translate, refold" procedure that is used to define the discrete Levi-Civita connection. *Hint: think about unfolding as a rotation.*

8.3.4. Holonomy.



At this point you may be wondering what all this stuff has to do with vector field design. Well, once we define a connection on our mesh, there's an easy way to construct a vector field: start out with an initial vector, parallel transport it to its neighbors using the connection, and repeat until you've covered the surface (as depicted above). One thing to notice is that the vector field we end up with is completely determined by our choice of connection. In effect, we can design vector fields by instead designing connections.

However, something can go wrong here: depending on which connection we use, the procedure above may not provide a consistent description of any vector field. For instance, consider the planar mesh below, and a connection that applies a rotation of 18° as we cross each edge in counterclockwise order. By the time we get back to the beginning, we've rotated our initial vector ❶ by only $5 \times 18^\circ = 90^\circ$. In other words, our connection would have us believe that ❶ and ❻ are parallel vectors!



This phenomenon is referred to as the *holonomy* of the connection. More generally, holonomy is the difference in angle between an initial and final vector that has been transported around a closed loop. (This definition applies in both the discrete and smooth setting.)

8.3.5. Trivial Connections. To construct a consistently-defined vector field, we must ensure that our connection has *zero* holonomy around every loop. Such a connection is called a *trivial connection*. In fact, the following exercise shows that this condition is sufficient to guarantee consistency everywhere:

EXERCISE 8.15

Show that parallel transport by a trivial connection is path-independent. *Hint: consider two different paths from point a to point b.*

As a result we can forget about the particular paths along which vectors are transported, and can again imagine that we simply "teleport" them directly from one point to another. If we then reconstruct a vector field via a trivial connection, we get a *parallel vector field*, *i.e.*, a field where (at least according to the connection) every vector is parallel to every other vector. In a sense, parallel vector fields on surfaces are a generalization of *constant* vector fields in the plane. But actually, the following exercise shows that *any* vector field can be considered parallel—as long as we choose the right connection:

EXERCISE 8.16

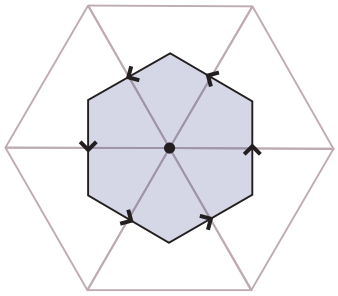
Show that every discrete vector field (*i.e.*, a vector per face) is parallel with respect to some trivial discrete connection. *Hint: think about the difference between vectors in adjacent triangles.*

8.3.6. Curvature of a Connection. We can use a trivial connection to define a vector field, but how do we find a trivial connection? The first thing you might try is the Levi-Civita connection—after all, it has a simple, straightforward definition. Sadly, the Levi-Civita connection is not in general trivial:

EXERCISE 8.17

Show that the holonomy of the discrete Levi-Civita connection around the boundary of any dual

cell equals the angle defect of the enclosed vertex.

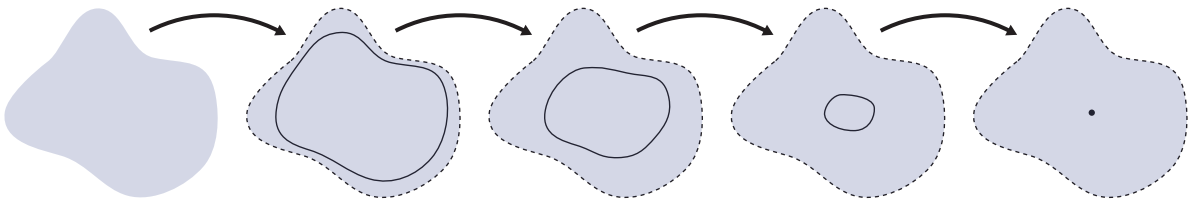


Therefore, unless our mesh is completely flat, Levi-Civita will exhibit some non-zero amount of holonomy. Actually, you may recall that angle defect is used to define a discrete notion of *Gaussian curvature*. We can also use a connection to determine curvature—in particular, the *curvature of a connection* (smooth or discrete) over a topological disk $D \subset M$ is given by the holonomy around the region boundary ∂D.

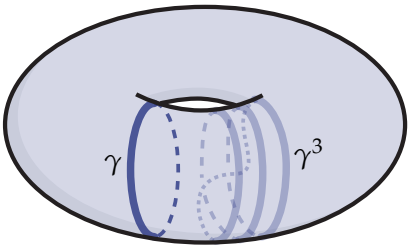
EXERCISE 8.18

Show that these two notions of curvature are the same, *i.e.*, show that the curvature of the discrete Levi-Civita connection over any disk D equals the total discrete Gaussian curvature over that region. *Hint: use induction on faces.*

Curvature gives us one tool to test whether a connection is trivial. In particular, a trivial connection *must* have zero curvature everywhere. For this reason it's reasonable to say that every trivial connection is "flat." But is every flat connection also trivial? Well, remember that the curvature of a connection is defined in terms of the holonomy around region boundaries. Any such boundary is called a *contractible loop* because it can be continuously deformed to a point without "catching" on anything:



In general, there may also be *noncontractible* loops on a surface that *cannot* be described as the boundary of any disk. For instance, consider the loop γ pictured on the torus to the left:



In general a surface of genus g will have $2g$ "basic types" of noncontractible loops called *generators*. More precisely, two loops are said to be *homotopic* if we can get from one to the other by simply sliding it along the surface without ever breaking it. No two distinct generators are homotopic to each other, and what's more, we can connect multiple copies of the generators to "*generate*" any noncontractible loop on the surface. For instance, consider the loop γ^3, which consists of three copies of γ joined end-to-end. (Formally, the space of loops together with the operation of concatenation describe the *first homology group* on the surface.)

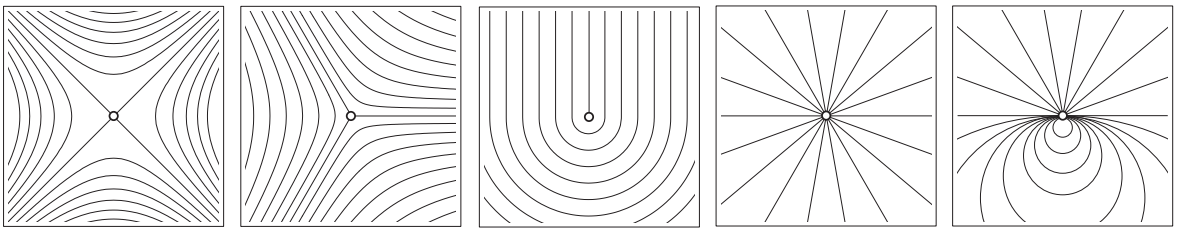
If we want to check if a connection is trivial, we need to know that it has nontrivial holonomy around both contractible *and* noncontractible loops. Equivalently: it must have zero *curvature* and nontrivial holonomy around noncontractible loops. As you're about to demonstrate, though, we don't need to check all the loops—just a small collection of *basis* loops.

EXERCISE 8.19

Show that the holonomy around any discrete loop is determined by the curvature at each vertex and the holonomy around a collection of $2g$ generators.

8.3.7. Singularities. There's one more issue we run into when trying to find a trivial connection. You may remember the *Gauss-Bonnet* theorem, which says that $\sum_{v \in V} d(v) = 2\pi\chi$, *i.e.*, the total Gaussian curvature over a surface equals 2π times the Euler characteristic χ. In fact, this theorem holds if we replace the Gaussian curvature with the curvature of *any* connection (not just Levi-Civita). But something's odd here: didn't we say that a trivial connection should have zero holonomy—hence zero curvature? So unless $\chi = 0$ (*i.e.*, M is a torus) we have a problem!

Fortunately the solution is simple: we can permit our connection to exhibit nonzero holonomy (hence nonzero curvature) around some loops, as long as this holonomy is an integer multiple of 2π. Geometrically, a vector parallel transported around any closed loop will still end up back where it started, even if it "spins around" some whole number of times k along the way. Any vertex where $k \neq 0$ is called a *singularity* (see below for some examples). As we'll see in the moment, singularities actually make it *easier* to design vector fields with the desired appearance, since one can control the global appearance of the field using only a small number of degrees of freedom.



8.4. Vector Field Design

Now on to the fun part: designing vector fields. At this point, you've already written most of the code you'll need! But let's take a look at the details. To keep things simple we're going to assume that M is a topological sphere, so you can forget about non-contractible loops for now.

Our goal is to find a connection 1-form φ such that the holonomy around every loop is zero. If we let $\varphi_{ij} = 0$ for every dual edge $e^\star_{ij}$, then the holonomy around any dual cell will be equal to the integrated Gaussian curvature over that cell (Exercise 17), which we'll denote by $K \in \mathbb{R}^{|V|}$. Therefore, we need to find angles φ_{ij} such that

$$d_0^T \varphi = -K, \tag{5}$$

i.e., the holonomy around the boundary of every dual cell should exactly *cancel* the Gaussian curvature. (Notice that d_0^T is the discrete exterior derivative on *dual* 1-forms.) We also need to incorporate singularities. That's easy enough: we can just ask that the angles φ_{ij} cancel the existing Gaussian curvature, but *add* curvature corresponding to singularities:

$$d_0^T \varphi = -K + 2\pi k. \tag{6}$$

Here $k \in \mathbb{Z}^{|V|}$ is a vector of integers encoding the type of singularity we want at each vertex. To "design" a vector field, then, we can just set k to a nonzero value wherever we want a singularity. Of course, we need to make sure that $\sum_i k_i = \chi$ so that we do not violate Gauss-Bonnet.

CODING 35. Implement the method `Vertex::totalGaussCurvature()` which computes the total Gauss curvature of the dual vertex associated with a given vertex, *i.e.*, 2π minus the sum of incident angles.

We now have a nice linear system whose solution gives us a trivial connection with a prescribed set of singularities. One last question, though: is the solution unique? Well, our connection is determined by one angle per edge, and we have one equation to satisfy per dual cell—or equivalently, one per vertex. But since we have roughly three times as many edges as vertices (which you showed earlier on!), this system is *underdetermined*. In other words, there are *many* different trivial connections on our surface. Which one gives us the "nicest" vector field? While there's no completely objectively answer to this question, the most reasonable thing may be to look for the trivial connection *closest* to Levi-Civita. Why? Well, remember that Levi-Civita "twists" vectors as little as possible, so we're effectively asking for the *smoothest* vector field. Computationally, then, we need to find the solution to Equation 6 with minimum norm (since the angles φ_{ij} already encode the deviation from Levi-Civita). As a result, we get the optimization problem

$$\begin{aligned} \min_{\varphi \in \mathbb{R}^{|E|}} \quad & ||\varphi||^2 \\ \text{s.t.} \quad & d_0^T \varphi = -K + 2\pi k. \end{aligned} \tag{7}$$

One way to solve this problem would be to use some kind of steepest descent method, like we did for mean curvature flow. However, we can be a bit more clever here by recognizing that Equation 7 is equivalent to looking for the solution to Equation 6 that has no component in the null space of d_0^T—any other solution will have larger norm.

EXERCISE 8.20

Show that the null space of d_0^T is spanned by the columns of d_1^T. *Hint: what happens when you apply d twice?*

Hence, to get the smoothest trivial connection with the prescribed curvature we could (1) compute *any* solution $\tilde{\varphi}$ to Equation 6, then (2) project out the null space component by computing $\varphi = \tilde{\varphi} - d_1^T(d_1 d_1^T)^{-1} d_1 \tilde{\varphi}$. Overall, then, we get a trivial connection by solving two nice, sparse linear systems. Sounds pretty good, and in fact that's how the algorithm was originally proposed way back in 2010. But it turns out there's an even nicer, more elegant way to compute trivial connections, using Helmholtz-Hodge decomposition. (This formulation will also make it a little easier to work with surfaces of nontrivial topology.)

8.4.1. Trivial Connections++. So far, we've been working with a 1-form φ, which describes the deviation of our connection from Levi-Civita. Just for fun, let's rewrite this problem in the smooth setting, on a closed surface M of genus g. Our measure of smoothness is still the total deviation from Levi-Civita, which we can again write as $||\varphi||^2$. We still have the same constraint on simply-connected cycles, namely $d\varphi = u$ where $u = -K + 2\pi k$ (in the smooth setting, we can think of k as a sum of Dirac deltas). This time around, we'll also consider the holonomy around the $2g$ nontrivial generators ℓ_i. Again, we'll use the 1-form φ to "cancel" the holonomy we experience in the smooth setting, *i.e.*, we'll enforce the constraint

$$\int_{\ell_i} \varphi = v_i,$$

where $-v_i$ is the holonomy we get by walking around the loop ℓ_i. (In the discrete setting we can measure this quantity as before: transport a vector around each loop by unfolding, sliding, and refolding *without* any extra in-plane rotation.) Overall, then, we get the optimization problem

$$\begin{aligned} \min_{\varphi} \quad & ||\varphi||^2 \\ \text{s.t.} \quad & d\varphi = u, \\ & \textstyle\int_{\ell_i} \varphi = v_i,\ i = 1, \ldots, 2g. \end{aligned} \tag{8}$$

Like any other 1-form, φ has a Hodge decomposition, *i.e.*, we can write it as

$$\varphi = d\alpha + \delta\beta + \gamma$$

for some 0-form α, 2-form β, and harmonic 1-form γ. This expression can be used to simplify our optimization problem, as you are about to show!

EXERCISE 8.21

Show that Equation 8 can be rewritten as

$$\begin{aligned} \min_{\varphi} \quad & ||\delta\beta||^2 + ||\gamma||^2 \\ \text{s.t.} \quad & d\delta\beta = u, \\ & \textstyle\int_{\ell_i} \delta\beta + \gamma = v_i,\ i = 1, \ldots, 2g. \end{aligned}$$

Hint: use Stokes' theorem and the result of Exercise 3.

There are a couple nice things about this reformulation. First, we can find the coexact part by simply solving the linear equation $d\delta\beta = u$, or equivalently $d \star d \star \beta = u$. As with Hodge decomposition, we can make this system even nicer by making the substitution $\tilde{\beta} := \star\beta$, in which case we end up with a standard scalar Poisson problem

$$\Delta\tilde{\beta} = u. \tag{9}$$

We can then recover β itself via $\beta = \star\tilde{\beta}$, as before. Note that the solution $\tilde{\beta}$ is unique up to a constant, since on a compact domain the only harmonic 0-forms are the constant functions (as you showed when we studied the Poisson equation). Of course, since constants are in the kernel of δ, we still get a uniquely-determined coexact part $\delta\beta$.

The second nice thing about this formulation is that we can directly solve for the harmonic part γ by solving the system of linear equations

$$\int_{\ell_i} \gamma = v_i - \int_{\ell_i} \delta\beta,$$

i.e., since $\delta\beta$ is uniquely determined by Equation 9, we can just move it to the right-hand side.

A slightly nicer way to write this latter system is using the *period matrix* of our surface. Let $\ell_1, \ldots, \ell_{2g}$ be a collection of homology generators, and let $\xi_1, \ldots, \xi_{2g}$ be a basis for the harmonic 1-forms. The period matrix $\mathsf{P} \in \mathbb{R}^{2g \times 2g}$ is then given by

$$\mathsf{P}_{ij} = \int_{\ell_i} \xi_i,$$

i.e., it measures how much each harmonic basis "lines up" with each generating cycle. Period matrices have an intimate relationship with the conformal structure of a surface, which we discussed when looking at parameterization. But we don't have time to talk about that now—we have to compute vector fields! In the discrete setting, we can compute the entries of the period matrix by simply summing up 1-form values over generating cycles. In other words, if ℓ_i is a collection of dual edges forming a loop, and ξ_i is a dual discrete 1-form (*i.e.*, a value per dual edge), then we have

$$\mathsf{P}_{ij} = \sum_{e_k^\star \in \ell_i} (\xi_j)_k.$$

CODING 36. Implement the method `Connection::buildPeriodMatrix()`, which simply sums up the values of the harmonic 1-form bases over each homology generator. (You should compute these quantities using your existing implementation of **TREE-COTREE** and **HARMONIC-BASIS**.)

Once we have the period matrix, we can express our constraint on generator holonomy as follows. Let $\mathsf{z} \in \mathbb{R}^{2g}$ be the coefficients of the harmonic component γ with respect to our basis of

harmonic 1-forms, *i.e.*,

$$\gamma = \sum_{i=1}^{2g} z_i \xi_i.$$

Also, let $\tilde{v} \in \mathbb{R}^{2g}$ be the right-hand side of our constraint equation, encoding both the desired generator holonomy as well as the integrals of the coexact part along each generator:

$$\tilde{v}_i = v_i - \int_{\ell_i} \delta\beta.$$

Then the harmonic component can be found by solving the $2g \times 2g$ linear system

$$Pz = \tilde{v},$$

where the period matrix P is constant (*i.e.*, it depends only on the mesh geometry and not the configuration of singularities or generator holonomy).

Overall, then, we have the following algorithm for computing the smoothest vector field on a simplicial surface with a prescribed collection of singularities:

ALGORITHM 7 (Trivial-Connection++).

Require: Vector $k \in \mathbb{Z}^{|V|}$ of singularity indices adding up to $2\pi\chi$

1: Solve $\Delta\tilde{\beta} = u$
2: Solve $Pz = \tilde{v}$
3: $\varphi \leftarrow \delta\beta + \gamma$

The resulting 1-form can be used to produce a unit vector field via the procedure described in Section 8.3.4. Note that the most expensive part of the entire algorithm is prefactoring the cotan-Laplace matrix, which is subsequently used to compute both the harmonic 1-form bases and to update the potential β. In comparison, all other steps (finding generating loops, *etc.*) have a negligible cost, and moreover can be computed just once upon initialization (*e.g.*, the period matrix P). In short, *finding the smoothest vector field with prescribed singularities costs about as much as solving a single scalar Poisson problem!* If you've been paying attention, you'll notice that this statement is kind of a theme in these notes: if treated correctly, many of the fundamental geometry processing tasks we're interested in basically boil down to solving a Poisson equation. (This outcome is particularly nice, since in principle we can use the same prefactorization for many different applications!)

CODING 37. Write the method `Connection::compute()`, which implements the algorithm **TRIVIAL-CONNNECTION++**. You should now be able to edit vector fields through the Viewer by shift-clicking on singularities.

CHAPTER 9

Conclusion

Given the framework you've already built, a bunch of other geometry processing algorithms can be implemented almost immediately. For instance, one can compute shortest paths or *geodesic distance* by solving a Poisson equation and integrating a heat flow, just as in Chapter 6 [**CWW13**]. One can also improve the quality of the mesh itself, again by solving simple Poisson equations [**MMdGD11**]. More broadly, one can use these tools to simulate mechanical phenomena such as elastic bodies [**ACOL00**]. These topics (and more!) will be covered in a future revision of these notes.

Bibliography

[ACOL00] Marc Alexa, Daniel Cohen-Or, and David Levin. As-Rigid-as-Possible Shape Interpolation. In *Proc. ACM SIGGRAPH*, pages 157–164, 2000.

[CDS10] Keenan Crane, Mathieu Desbrun, and Peter Schröder. Trivial Connections on Discrete Surfaces. *Comp. Graph. Forum*, 29(5):1525–1533, 2010.

[CWW13] Keenan Crane, Clarisse Weischedel, and Max Wardetzky. Geodesics in Heat: A New Approach to Computing Distance Based on Heat Flow. *ACM Trans. Graph.*, 2013.

[dGC10] Fernando de Goes and Keenan Crane. Trivial Connections Revisited: A Simplified Algorithm for Simply-Connected Surfaces, 2010.

[DHLM05] Mathieu Desbrun, Anil Hirani, Melvin Leok, and Jerrold Marsden. Discrete Exterior Calculus. *ArXiv e-prints*, 2005.

[DKT08] Mathieu Desbrun, Eva Kanso, and Yiying Tong. Discrete Differential Forms for Computational Modeling. In Alexander I. Bobenko, Peter Schröder, John M. Sullivan, and Günther M. Ziegler, editors, *Discrete Differential Geometry*, volume 38 of *Oberwolfach Seminars*, pages 287–324. Birkhäuser Verlag, 2008.

[DMSB99] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan Barr. Implicit Fairing of Irregular Meshes using Diffusion and Curvature Flow. In *Proc. ACM SIGGRAPH*, pages 317–324, 1999.

[Epp03] David Eppstein. Dynamic Generators of Topologically Embedded Graphs. In *Proc. ACM-SIAM Symp. Disc. Alg. (SODA)*, 2003.

[EW05] Jeff Erickson and Kim Whittlesey. Greedy Optimal Homotopy and Homology Generators. In *Proc. ACM-SIAM Symp. Disc. Alg. (SODA)*, 2005.

[Hat02] A. Hatcher. *Algebraic Topology*. Algebraic Topology. Cambridge University Press, 2002.

[Hir03] Anil Hirani. *Discrete Exterior Calculus*. PhD thesis, Pasadena, CA, USA, 2003.

[KCPS13] Felix Knöppel, Keenan Crane, Ulrich Pinkall, and Peter Schröder. Globally Optimal Direction Fields. In *Proc. ACM SIGGRAPH*, 2013.

[Mac49] Richard MacNeal. *The Solution of Partial Differential Equations by means of Electrical Networks*. PhD thesis, Caltech, 1949.

[MMdGD11] Patrick Mullen, Pooran Memari, Fernando de Goes, and Mathieu Desbrun. HOT: Hodge-optimized triangulations. In *Proc. ACM SIGGRAPH*, 2011.

[MTAD08] Patrick Mullen, Yiying Tong, Pierre Alliez, and Mathieu Desbrun. Spectral Conformal Parameterization. *Comp. Graph. Forum*, 27(5):1487–1494, 2008.

[SC18] Nick Sharp and Keenan Crane. Variational surface cutting. *ACM Trans. Graph.*, 37(4), 2018.

[TACSD06] Y. Tong, P. Alliez, D. Cohen-Steiner, and M. Desbrun. Designing Quadrangulations with Discrete Harmonic Forms. In *Proc. Symp. Geom. Proc.*, 2006.

Appendices

APPENDIX A

Derivatives of Geometric Quantities

Here we consider derivatives of some basic geometric quantities associated with triangulated surfaces. Such derivatives are needed for, *e.g.*, computing the gradient of a discrete energy, which is often needed for energy minimization. Though the final expressions are given below, you are encouraged to try and derive them yourself. In general, there are a variety of ways to obtain the gradient for a discrete energy:

- **By hand, in components** — The usual strategy for taking a gradient (which you may have learned in an introductory calculus class) is to simply write out a list of partial derivatives, expressed in components. For instance, for an energy $\phi(x_1, \ldots, x_n)$ you might write out the derivative $\partial\phi/\partial x_1$, then $\partial\phi/\partial x_2$, and so on. The upside to this approach is that it will always produce an answer: as long as you are willing to carefully grind through enough applications of the chain rule, you will eventually come up with the correct expression. For this reason, it is also easy to automate this process algorithmically (see *symbolic differentiation*, below). On the other hand, writing everything out in components can sometimes take a great deal of work (especially for a large number of variables), and without careful simplification may lead to long complicated expressions that are difficult to interpret and implement (see examples below). One way to simplify such calculations is to work out derivatives in terms of matrices or tensors (instead of individual components), though even here there can sometimes be a disconnect between expressions and their geometric meaning. In general, it is always good to look for simplifications you can make based on geometry—for instance, if your calculation has a term $e_{ij} + e_{jk} + e_{ki}$ where the vectors e are the edges of a triangle, you know this expression can be simplified to zero!
- **Symbolic differentiation** — Symbolic differentiation is very much like taking derivatives by hand, except that the process is automated algorithmically. If you've ever used a package like *Mathematica* or *Maple* to compute a derivative, you have probably (unknowingly) used symbolic differentiation. The basic idea is that, given an input expression, the algorithm builds a corresponding expression tree; one can then obtain derivatives by applying deterministic transformations to this expression tree. Further transformations will simplify the expression, though in general the problem of expression simplification is not easy: in practice, even good implementations of symbolic differentiation can produce derivative expressions that are *far* longer and more complicated than needed (see examples below). On the other hand, once an expression is available, it can be efficiently evaluated and re-evaluated to obtain derivates at different points in the domain of the function.
- **Numerical differentiation** — A very different approach to obtaining derivatives is to forget entirely about finding a closed-form expression, and simply obtain a numerical approximation. For instance, if $\phi(x)$ is a function of a single variable x, its derivative at a point x_0 can be approximated by the finite difference $\frac{1}{\varepsilon}(\phi(x_0 + \varepsilon) - \phi(x_0))$ for some small value $\varepsilon > 0$ (essentially just a first-order Taylor series approximation). For a function of many variables, this process is repeated for each variable, *i.e.*, just take a difference between the value at the point of interest and the value at a point where one of the coordinates

has been slightly perturbed. The numerical approach makes it trivial to differentiate very complicated functions ϕ, since one need only evaluate the function ϕ at a given point, and does not need to know anything about what this function looks like. For instance, one can differentiate arbitrary pieces of code provided as a "black box." On the other hand, numerical differentiation can be very expensive computationally since it requires many evaluations of ϕ (especially for higher-order derivatives); moreover, it can produce inaccurate and unpredictable results, especially since it is not obvious how to choose the parameter ε.

- **Automatic differentiation** — An algorithmic approach that is a sort of middle ground between numerical and symbolic differentiation is *automatic differentiation*. The basic idea here is that rather than manipulating ordinary values, the algorithm works with *tuples* that encode the value and its derivative. Operations defined on these tuples encode a transformation of both the value and its derivative (essentially by the chain rule). For instance, if we have two tuples (x, x') and (y, y') then their product might be defined as $(x, x') * (y, y') := (x * y, x * y' + x' * y)$, where the second term reflects the product rule $(xy)' = xy' + x'y$. Once these operations have been defined, one can evaluate an expression and its derivative simultaneously by simply constructing the expression from tuples rather than ordinary values. This approach is that it yields derivatives that are typically more accurate than basic numerical differentiation; it is also fairly efficient and can be applied to a rather general class of functions. On the other hand, it is often less efficient than symbolic differentiation (which can take advantage of simplification), and can often require far more computation than simple expressions derived using geometric insight.
- **By hand, using geometric arguments** — The approach we will take here, described in greater detail below, is to take derivatives by hand, but use geometric arguments rather than grinding out partial derivatives in coordinates. This approach cannot always be applied, but is highly effective for differentiating fundamental quantities that appear in geometry (lengths, angles, areas, and so forth). It can also provide some geometric insight into the *meaning* of derivatives, which are often connected to other objects in interesting and surprising ways. For instance, in Chapter 5 we saw that the derivative of discrete surface area yields a mean curvature normal expressed in terms of the cotangent Laplacian; a fact that would be missed entirely when using numerical or automatic differentiation. Finally, expressions derived this way are often short and sweet, meaning that they are easy to implement and less prone to common numerical artifacts (such as cancellation error) that might arise in automatically-derived expressions. See below for further discussion.

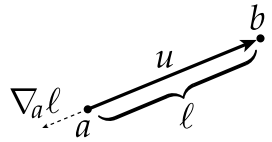
For a given quantity, the basic recipe for obtaining the gradient via geometric arguments follows a standard pattern:

- First, find the *unit vector* such that moving in that direction increases the quantity the fastest. This direction gives the direction of the gradient.
- Next, find the rate of change when moving in this direction. This quantity gives the magnitude of the gradient.

To make this idea clear, consider the following example:

Example. Let ℓ be the length of the vector $u := b - a$, where a and b are points in $\mathbb{R}^2$. Let $\hat{u} := u/\ell$ be the unit vector in the same direction as u. Show that the gradient of ℓ with respect to the location of the point a is given by

$$\nabla_a \ell = -\hat{u}.$$



Solution. The fastest way to increase ℓ is to move a along the direction of $-u$ (since a small motion in the orthogonal direction Ju looks like a rotation, which does not change the length). Since moving a by one unit increases the length ℓ by one unit, the magnitude of the gradient is 1. Thus, $\nabla_a \ell = -\hat{u}$.

(If you feel uneasy about this approach you may wish to compute the same expression in coordinates, just to be sure that you get the same thing!)

Two examples illustrate the benefit of the geometric approach. Recall that in Chapter 5 you were asked to derive an expression for the gradient of the area of a triangle with respect to one of its vertices. In particular, if the triangle has vertices $a, b, c \in \mathbb{R}^3$, then the gradient of its area A with respect to the vertex a can be expressed as

$$\nabla_a A = \tfrac{1}{2} N \times (b - c).$$

This formula can be obtained via a simple geometric argument, has a clear geometric meaning, and generally leads to a an efficient and error-free implementation. In contrast, here's the expression produced by taking partial derivatives via `Mathematica` (even after calling `FullSimplify[]`):

```
In[25]:= a = {a1, a2, a3};
         b = {b1, b2, b3};
         c = {c1, c2, c3};
         w = Cross[b - a, c - a] / 2;
         A = √w.w ;
         FullSimplify[{∂a1 A, ∂a2 A, ∂a3 A}]

Out[30]= { ((b2 - c2) (-b2 c1 + a2 (-b1 + c1) + a1 (b2 - c2) + b1 c2) +
             (b3 - c3) (-b3 c1 + a3 (-b1 + c1) + a1 (b3 - c3) + b1 c3)) /
           (2 √((a2 b1 - a1 b2 - a2 c1 + b2 c1 + a1 c2 - b1 c2)^2 +
                (a3 b1 - a1 b3 - a3 c1 + b3 c1 + a1 c3 - b1 c3)^2 +
                (a3 b2 - a2 b3 - a3 c2 + b3 c2 + a2 c3 - b2 c3)^2)),
         ((b1 - c1) (a2 (b1 - c1) + b2 c1 - b1 c2 + a1 (-b2 + c2)) +
             (b3 - c3) (-b3 c2 + a3 (-b2 + c2) + a2 (b3 - c3) + b2 c3)) /
           (2 √((a2 b1 - a1 b2 - a2 c1 + b2 c1 + a1 c2 - b1 c2)^2 +
                (a3 b1 - a1 b3 - a3 c1 + b3 c1 + a1 c3 - b1 c3)^2 +
                (a3 b2 - a2 b3 - a3 c2 + b3 c2 + a2 c3 - b2 c3)^2)),
         ((b1 - c1) (a3 (b1 - c1) + b3 c1 - b1 c3 + a1 (-b3 + c3)) +
             (b2 - c2) (a3 (b2 - c2) + b3 c2 - b2 c3 + a2 (-b3 + c3))) /
           (2 √((a2 b1 - a1 b2 - a2 c1 + b2 c1 + a1 c2 - b1 c2)^2 +
                (a3 b1 - a1 b3 - a3 c1 + b3 c1 + a1 c3 - b1 c3)^2 +
                (a3 b2 - a2 b3 - a3 c2 + b3 c2 + a2 c3 - b2 c3)^2))}
```

Longer expressions like these of course produce the correct values. But without further simplification (by hand) they will often be less efficient, and can potentially exhibit poorer numerical

behavior due to the use of a longer sequence of floating-point operations. Moreover, they are far less easy to understand/interpret, especially if this calculation is just one small piece of a much larger equation (as it often is). In general, taking gradients the "geometric way" often provides greater simplicity and deeper insight than just grinding everything out in components. Another example is the expression for the gradient of angle via partial derivatives, as computed by *Mathematica* (see below for a *much* simpler expression derived via geometric arguments):

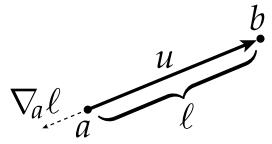
```
In[58]:= a = {a1, a2, a3};
         b = {b1, b2, b3};
         c = {c1, c2, c3};
         θ = ArcCos[ (a - b).(c - b) / (√((a - b).(a - b)) √((c - b).(c - b))) ];
         FullSimplify[{∂a1 θ, ∂a2 θ, ∂a3 θ}]
```

$$
\begin{aligned}
\text{Out[62]=}\ \Bigg\{ & \Big(a1\,b2^2 + a1\,b3^2 - a2\,b2\,(a1 + b1 - 2\,c1) - a3\,b3\,(a1 + b1 - 2\,c1) + a2^2\,(b1 - c1) + a3^2\,(b1 - c1) - b2^2\,c1 - \\
& \quad b3^2\,c1 + a2\,(a1 - b1)\,c2 - a1\,b2\,c2 + b1\,b2\,c2 + a3\,(a1 - b1)\,c3 - a1\,b3\,c3 + b1\,b3\,c3 \Big) \Big/ \\
& \Bigg(\left((a1 - b1)^2 + (a2 - b2)^2 + (a3 - b3)^2 \right)^{3/2} \sqrt{(b1 - c1)^2 + (b2 - c2)^2 + (b3 - c3)^2} \\
& \quad \sqrt{1 - \frac{((a1 - b1)\,(-b1 + c1) + (a2 - b2)\,(-b2 + c2) + (a3 - b3)\,(-b3 + c3))^2}{\left((a1 - b1)^2 + (a2 - b2)^2 + (a3 - b3)^2\right)\left((b1 - c1)^2 + (b2 - c2)^2 + (b3 - c3)^2\right)}} \Bigg), \\
& \Big(a3^2\,b2 - a3\,b2\,b3 + b1\,b2\,c1 + a1^2\,(b2 - c2) - a3^2\,c2 - b1^2\,c2 + 2\,a3\,b3\,c2 - b3^2\,c2 - \\
& \quad a1\,(a2\,(b1 - c1) + b2\,(b1 + c1) - 2\,b1\,c2) + a2\,(b1\,(b1 - c1) - (a3 - b3)\,(b3 - c3)) - a3\,b2\,c3 + \\
& \quad b2\,b3\,c3 \Big) \Big/ \Bigg(\left((a1 - b1)^2 + (a2 - b2)^2 + (a3 - b3)^2 \right)^{3/2} \sqrt{(b1 - c1)^2 + (b2 - c2)^2 + (b3 - c3)^2} \\
& \quad \sqrt{1 - \frac{((a1 - b1)\,(-b1 + c1) + (a2 - b2)\,(-b2 + c2) + (a3 - b3)\,(-b3 + c3))^2}{\left((a1 - b1)^2 + (a2 - b2)^2 + (a3 - b3)^2\right)\left((b1 - c1)^2 + (b2 - c2)^2 + (b3 - c3)^2\right)}} \Bigg), \\
& \Big(b3\,(b1\,c1 + (a2 - b2)\,(a2 - c2)) + a3\,(b1\,(b1 - c1) - (a2 - b2)\,(b2 - c2)) + a1^2\,(b3 - c3) - \\
& \quad \left(b1^2 + (a2 - b2)^2\right) c3 - a1\,(a3\,(b1 - c1) + b3\,(b1 + c1) - 2\,b1\,c3) \Big) \Big/ \\
& \Bigg(\left((a1 - b1)^2 + (a2 - b2)^2 + (a3 - b3)^2 \right)^{3/2} \sqrt{(b1 - c1)^2 + (b2 - c2)^2 + (b3 - c3)^2} \\
& \quad \sqrt{1 - \frac{((a1 - b1)\,(-b1 + c1) + (a2 - b2)\,(-b2 + c2) + (a3 - b3)\,(-b3 + c3))^2}{\left((a1 - b1)^2 + (a2 - b2)^2 + (a3 - b3)^2\right)\left((b1 - c1)^2 + (b2 - c2)^2 + (b3 - c3)^2\right)}} \Bigg) \Bigg\}
\end{aligned}
$$

A.1. List of Derivatives

We here give expressions for the derivatives of a variety of basic quantities often associated with triangle and tetrahedral meshes in $\mathbb{R}^3$. Unless otherwise noted, we assume quantities are associated with geometry in $\mathbb{R}^3$ (though many of the expressions easily generalize).

A.1.1. Edge Length.



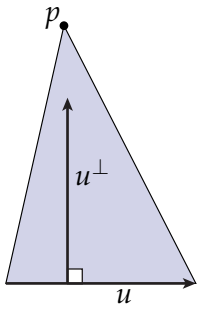
Let ℓ be the length of the vector $u := b - a$, where a and b are points in $\mathbb{R}^3$. Let $\hat{u} := u/\ell$ be the unit vector in the same direction as u. Then the gradient of ℓ with respect to the location of the point a is given by

$$\nabla_a \ell = -\hat{u}.$$

Similarly,

$$\nabla_b \ell = \hat{u}.$$

A.1.2. Triangle Area.

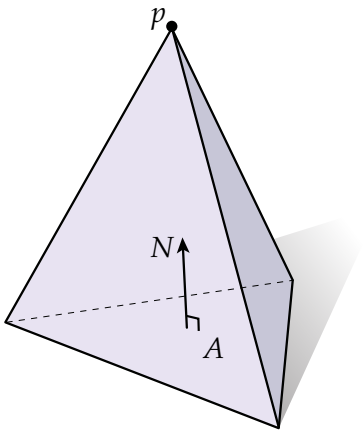


Consider any triangle in $\mathbb{R}^3$, and let u be the vector along the edge opposite a vertex p. Then the gradient of the triangle area A with respect to the location of p is

$$\nabla_p A = \tfrac{1}{2} N \times u,$$

where N is the unit normal of the triangle (oriented so that $N \times u$ points from u toward p, as in the figure above).

A.1.3. Tetrahedron Volume.

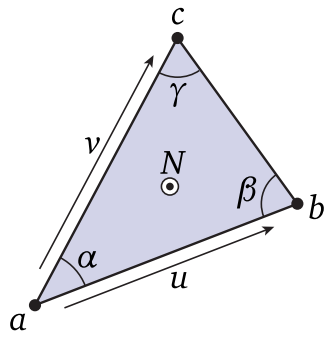


Consider any tetrahedron in $\mathbb{R}^3$, and let N be the unit normal of a triangle pointing toward the opposite vertex p (as in the figure above). The gradient of volume of the tetrahedron with respect to the location of p is

$$\nabla_p V = \frac{1}{3} AN,$$

where A is the area of the triangle with normal N.

A.1.4. Interior Angle.



Consider a triangle with vertices $a, b, c \in \mathbb{R}^3$, and let α be the signed angle between vectors $u := b - a$ and $v := c - a$. Then

$$\begin{aligned} \nabla_a \alpha &= -(\nabla_b \alpha + \nabla_c \alpha), \\ \nabla_b \alpha &= -(N \times u)/|u|^2, \\ \nabla_c \alpha &= (N \times v)/|v|^2. \end{aligned}$$

A.1.4.1. *Cosine.* Let θ be the angle between two vectors $u, v \in \mathbb{R}^3$. Then

$$\begin{aligned} \nabla_u \cos\theta &= (v - \langle v, \hat{u} \rangle \hat{u})/|u||v|, \\ \nabla_v \cos\theta &= (u - \langle u, \hat{v} \rangle \hat{v})/|u||v|, \end{aligned}$$

where $\hat{u} := u/|u|$ and $\hat{v} := v/|v|$. If u and v are edge vectors of a triangle with vertices $a, b, c \in \mathbb{R}^3$, namely $u := b - a$ and $v := c - a$, then

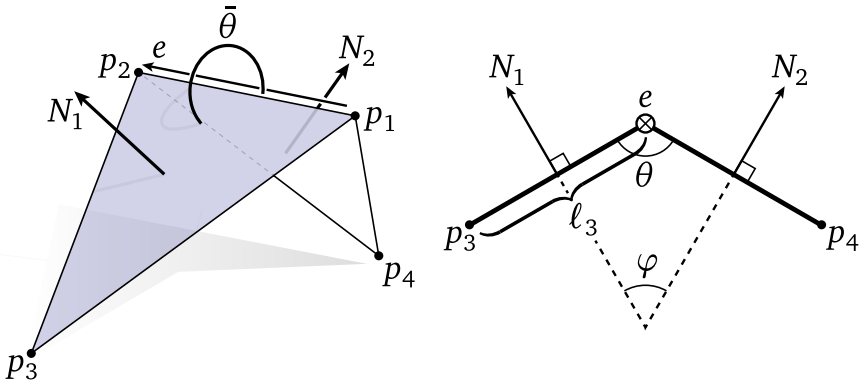
$$\nabla_a \cos\theta = -(\nabla_u \cos\theta + \nabla_v \cos\theta).$$

A.1.4.2. *Cotangent.* For any angle θ that depends on a vertex position p, we have

$$\nabla_p \cot\theta = -\frac{1}{\sin^2\theta}\nabla_p\theta.$$

The expression for the gradient of θ with respect to p can then be computed as above.

A.1.5. Dihedral Angle.



Consider a pair of triangles sharing an edge e, with vertices and normals labeled as in the figure above; let θ be the interior dihedral angle θ, complementary to the angle φ between normals. Explicitly, we can write θ as

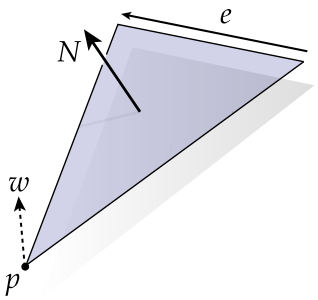
$$\theta = \text{atan2}(e \cdot (N_1 \times N_2), N_1 \cdot N_2),$$

where the use of the two-argument arc tangent function ensures we obtain the proper sign. Then

$$\begin{aligned} \nabla_{p_3}\theta &= |e|N_1/(2A_1), \\ \nabla_{p_4}\theta &= |e|N_2/(2A_2), \end{aligned}$$

where A_1, A_2 are the areas of the triangles with normals N_1, N_2, respectively. Gradients with respect to p_1 and p_2 can be found in the appendix to Wardetzky et al, *"Discrete Quadratic Curvature Energies"* (CAGD 2007).

A.1.6. Triangle Normal.



Consider a triangle in $\mathbb{R}^3$, and let e be the vector along an edge opposite a vertex p. If we move p in the direction w, the resulting change in the unit normal N (with orientation as depicted above)

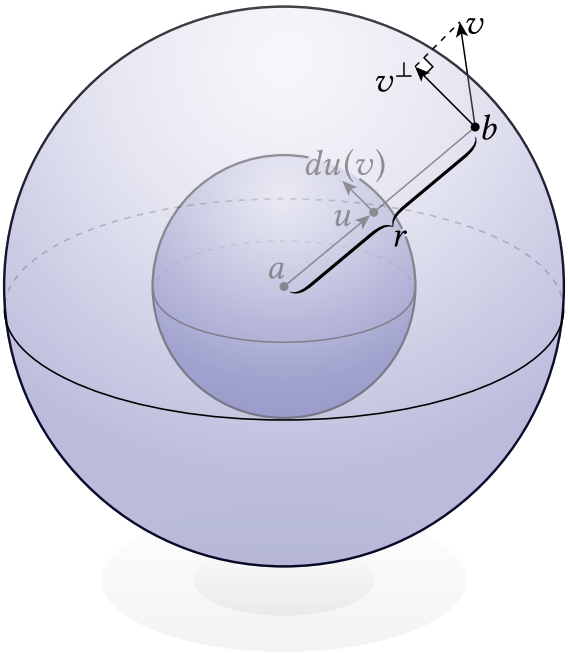
can be expressed as

$$dN(w) = \frac{\langle N, w \rangle}{2A} e \times N,$$

where A is the triangle area. The corresponding Jacobian matrix is given by

$$\frac{1}{2A}(e \times N)N^T.$$

A.1.7. Unit Vector.



Consider the unit vector $u := (b - a)/|b - a|$ associated with two points $a, b \in \mathbb{R}^3$. The change in this vector with respect to a motion of the endpoint b in the direction v can be expressed as

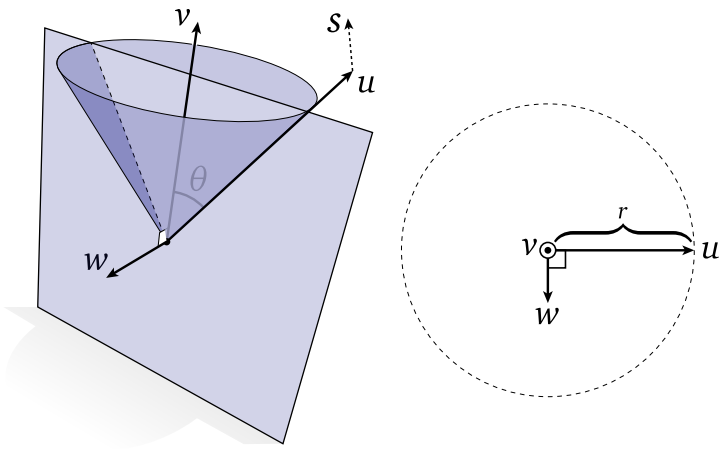
$$du(v) = \frac{v - \langle v, u \rangle u}{r},$$

where $r := |b - a|$ is the distance between a and b. The corresponding Jacobian matrix is

$$\frac{1}{r}(I - uu^T),$$

where I denotes the 3×3 identity matrix.

A.1.8. Cross Product.



For any two vectors $u, v \in \mathbb{R}^3$, consider the vector

$$w := \frac{u \times v}{|u \times v|}.$$

If we move u in the direction s, then the resulting change to w is given by

$$dw(s) = \frac{\langle w, s \rangle}{|u \times v|} w \times v.$$

The corresponding Jacobian matrix is

$$\frac{1}{|u \times v|}(w \times v)w^T.$$