A Computer Model of Paper Models of Negative Curvature

Matthew Cole and Benjamin DeMeo
Mentors: Chaim Goodman-Strauss and Diana Davis

August 9, 2013

1 Motivation

It is easy to make negatively curved polygons in space with paper, scissors, and tape. A few length calculations with the hyperbolic law of cosines allow one to add struts across the middle of such polygons, producing a fairly rigid surface. Furthermore, several polygons may be taped together, and it appears that such taping could continue indefinitely.

On the other hand, Hilbert's Embedding Theorem says the following:

Theorem 1. *A complete geometric surface of constant negative curvature cannot be immersed in* $\mathbb{R}^3$.

"Geometric surfaces" must be C^2; but it looks like our paper models are smooth. "Constant" curvature means that every point looks locally the same; but it looks like the center point of our models could fit over any other point. "Complete" surfaces have no sharp edges; but it looks like our models could be extended indefinitely. They would eventually intersect themselves, but that's not the problem- "immersions" allow self-intersections. Something else is not what it appears here.

These simple paper models look like they contradict Hilbert's Theorem. Since a paper model obviously will not overturn an established result, we set out to investigate exactly how these models fall short of the conditions of the theorem.

2 Overview

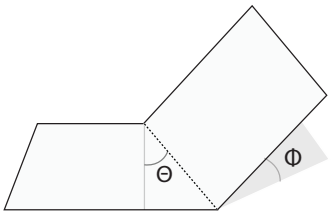
To accomplish our goals, we must implement the following:

- A flexible "strip of paper" must be represented digitally in a sensible way.
- If a strip starts in a curved position and is released, it must "relax" naturally into a flat state.
- If we hold the ends of a strip fixed, it must move to a state with the lowest possible elastic potential for the given constraints.
- Multiple strips must be able to interact with each other and relax as a unit.

2.1 Digital representation of strips

We would like to represent a strip in such a way as to allow every possible configuration of the strip, but the scope and variety of these configurations makes this a daunting task. Some approximation of a smooth strip must suffice - the finer the approximation, the larger the range of possible configurations. We used a discrete model: our strips are piecewise flat with a finite number of folds, which are evenly spaced along the length of the strip. Each fold is characterized by two variables (see figure 1): its lateral angle on the surface of the paper θ, and its exterior dihedral angle ϕ (we use the exterior rather than interior for ease of

Figure 1: One fold with its parameters θ (lateral) and ϕ (exterior dihedral)



computation). θ can take values in $(-\frac{\pi}{2}, \frac{\pi}{2})$, and ϕ ranges over $(-\pi, \pi)$. We refer throughout to the portion of a strip between folds as a 'segment.'

Our strips are oriented: they have a start and an end. At any point, there is a direction the strip is traveling (the tangent), a direction perpendicular to the surface of the strip (the normal), and a direction toward the side of the strip (the binormal). These three directions form an orthonormal basis for $\mathbb{R}^3$. We chose the matrix of column vectors (B, T, N) as our standard form. All orientation matrices have the binormal in the first column, the tangent in the second column, and the normal in the third column.

2.2 Minimizing strip energy (without constraints)

The mathematical theory of curvature helps us address the second problem. Given a point p on a two-dimensional surface S embedded in $\mathbb{R}^3$, let κ_1 and κ_2 be the maximum and minimum curvatures of any cross section of S taken normal to the surface at p. The quantity $H = \frac{\kappa_1+\kappa_2}{2}$ is called the *mean curvature* of S at p. The bending energy of a surface is well characterized by $Q = \int_S H^2$ in the sense that minimizing this quantity yields a surface with minimal energy. To simulate the bending energy of a strip, we attributed a "cost" to each fold indicating its approximate contribution to the bending potential; this is discussed further in the description of CostCircles. Let Q^* denote our approximation of Q.

We now need to make a strip"relax." In other words, we must minimize the function $Q^* : \mathbb{R}^{2k} \to \mathbb{R}$. Recall that the gradient ∇Q^* is a vector in $\mathbb{R}^{2k}$ giving the direction of greatest increase in Q. Hence, $-\nabla Q$ gives the direction of greatest decrease. To find the minimum, we scale $-\nabla Q$ down appropriately and add the result to the vector in $\mathbb{R}^{2k}$ specifying our initial configuration. In the future, we will refer to this operation as *stepping down* the cost gradient. Continued stepping leads us to a minimum. This method as a whole is known as *gradient descent*.

2.3 Minimizing strip energy (with constraints)

The third problem calls for constrained optimization. Holding the endpoints of a strip fixed imposes a constraint on the range of acceptable configurations. Since the total range of configurations is represented by $\mathbb{R}^{2k}$, the set of configurations satisfying a some constraint is a subset of $\mathbb{R}^{2k}$ called the *constraint space*. Phrased mathematically, then, the problem is to find the point within the constraint space having the lowest energy. Again, this is accomplished by gradient descent, but we must take care not to leave the constraint space. Section 4 discusses various methods in detail.

2.4 Strip interactions

Finally, we accomodate many strips interacting at once. Since each strip has its own attributes, one would like to represent them as individual objects storing their own information. While *Mathematica* is not object oriented, we can define functions within a strip initialization that allow future access to strip data. For

example, within the function InitializeStrip, we define a function called TotalLength, taking a single "Strip ID" parameter, that returns the length of the strip. These functions serve the role of accessor methods in object-oriented languages.

The Strip ID concept allows us to arrange, constrain, and optimize multiple strips individually. In principle, we could relax entire groups of strips as a unit, but as of August 7, 2013, this capability is not supported.

3 Basic Functionality

These sections contain the functions to model, compute with, and draw strips of paper in *Mathematica*.

3.1 Initialization

The 'Initialization' section of the code contains two functions. GetNextID[] returns a string suitable for being a strip's ID, and InitializeStrip ...

3.1.1 InitializeStrip

Takes: List of ten arguments, explained below
Returns: Null

InitializeStrip sets a new strip up with all the values it needs to be acted on by all our functions. It takes a list with the following ten arguments:

1. Desired Strip ID, **string**
2. Starting point, **xyz vector**
3. Target endpoint, **xyz vector**
4. Starting orientation, **3×3 matrix**, (Binormal, Tangent, Normal) as COLUMN vectors
5. Target end orientation, **3×3 matrix**, (Binormal, Tangent, Normal) as COLUMN vectors
6. Total length, **positive real number**
7. Number of flat turns, **nonnegative integer**
8. Turn angle, **real number**
9. Initial "Theta" parameter list, **list of angles between** $-\frac{\pi}{2}$ **and** $\frac{\pi}{2}$
10. Initial "Phi" parameter list, **list of angles between** $-\pi$ **and** π, same length as "Theta" list

InitializeStrip associates these values, as well as a few values immediately derived from them, with the Strip ID. Other functions can then call these values by the Strip ID and thus measure or interpret or change the strip.

3.2 Energy function

This section contains functions to calculate the bending energy of strips, modelling their integral square mean curvature.

3.2.1 CostCircles

Takes: List of two angles
Returns: Positive real number

The input angles specify a single fold on a strip of paper. The first, *Theta*, denotes the angle made by the fold with a horizontal line running breadthwise across the strip. The second, *Phi*, measures the angle between the planes containing the strip before and after the fold. CostCircles attempts to attribute a cost to this fold by estimating its contribution to the total elastic bending potential of the strip, measured as the integral of its mean curvature squared.

To represent the discrete fold as a continuous surface, we first attribute a region of the strip whose curvature is to be approximated by the fold. The boundary of this region is obtained simply by translating the fold up and down the strip by a fixed amount ϵ depending on the spacing between folds. We then represent this region as a "trough" where cross-sections by planes normal to the fold yield circular segments tangent to the strip. Computing the total mean curvature of this trough gives the approximate contribution of the fold.

The associated functions **DCostCirclesTheta** and **DCostCirclesPhi**, with the same input and output types, give the derivatives of CostCircles with respect to its parameters *Theta* and *Phi*.

3.3 Matrix Operations

This section contains functions which calculate and transform strips' orientation matrices. Many of them are used extensively by the rest of our code.

3.3.1 M

Takes: List of two angles
Returns: 3×3 matrix

M is the matrix of the coordinate transformation of a fold with angles θ and ϕ.

M is derived using Euler angles. To transform the coordinate system from one fold to the next, three rotations are necessary (see figure 2):

1. Rotate about Normal by θ;
2. Rotate about Binormal by ϕ;
3. Rotate about Normal by $-\theta$.

This is equivalent to a ZXZ Euler angle transformation with angles θ, ϕ, and $-\theta$, respectively. M is the product of these three rotation matrices.

3.3.2 dMdθ

Takes: List of two angles
Returns: 3×3 matrix

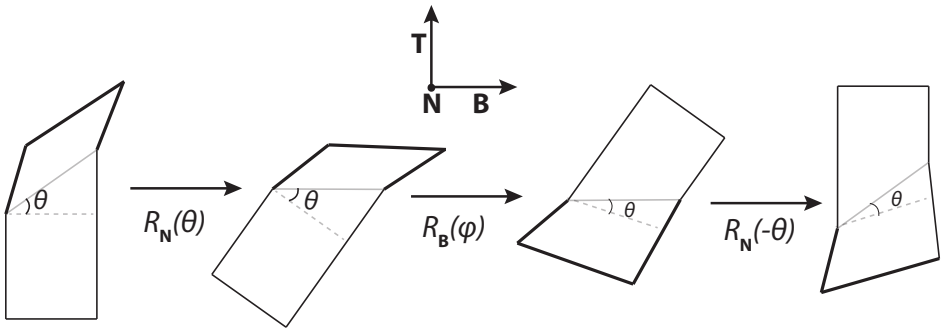
dMdθ is the componentwise partial derivative of 'M' with respect to θ.

3.3.3 dMdϕ

Takes: List of two angles
Returns: 3×3 matrix

dMdϕ is the componentwise partial derivative of 'M' with respect to ϕ.

Figure 2: The three plane rotations to transform coordinates from one segment to the next



3.3.4 PlaneRotMatrix

Takes: One angle
Returns: 3×3 matrix

PlaneRotMatrix returns the matrix corresponding to a rotation in the Binormal-Tangent plane, leaving the Normal fixed.

3.3.5 TurnMatrix

Takes: Strip ID
Returns: 3×3 matrix

A strip may have a number of flat turns in it. These must all be the same angle, and they will be as close to equally spaced along the strip as possible. TurnMatrix applies PlaneRotMatrix to obtain the matrix used for turns in the given strip.

3.3.6 TurnSpots

Takes: Strip ID
Returns: List of singleton integers (or, column vector of integers)

TurnSpots calculates the indices at which to insert the turn matrix into the list of transformation matrices. It separates the folds of the strip into one more part than there are turns and returns the indices at the edges of the separation. If the division cannot be exactly even, "Ceiling" spreads the extra segments as uniformly as possible. The indices are transposed for each of use with "Insert" in 'MList' below.

3.3.7 MList

Takes: List of a Strip ID
Returns: List of 3×3 matrices

MList calculates the transformation matrices for each segment of the given strip. It uses M on the folds and employs TurnSpots and TurnMatrix to insert the turn transformations properly.

3.3.8 HeadList

Takes: List containing a Strip ID and a list of 3×3 matrices
Returns: List of 3×3 matrices

HeadList calculates partial products of a list of matrices. Usually, it is applied to 'MList,' which results in a list of the orientation matrices at each segment of the given strip. The "IdentityMatrix" which is prepended in HeadList represents the first segment, before any folds or turns are made. Premultiplication by the strip's StartMatrix reinforces the conventional use of HeadList already mentioned; in the one other way HeadList is used (in the 'PartialTails' item in gradient calculations) this premultiplication is undone.

Given a strip's start matrix M_0 and a list $(M_1, \ldots, M_k)$, HeadList returns $(M_0,\ M_0M_1,\ M_0M_1M_2,\ \ldots,\ M_0M_1 \cdots M_k)$.

Orientation matrices are usually used as basis vectors. Interpreted as vectors, part i of HeadList gives (Binormal,Tangent,Normal) of the i^{th} segment as column vectors.

Since a strip is traveling in the direction of its tangent, its total displacement may be found by summing the tangents at each segment and scaling by the segment length, a task easily accomplished by suitably scaling the tangent column of the total of its HeadList.

3.3.9 TailList

Takes: List of a Strip ID
Returns: List of 3×3 matrices

TailList calculates partial products of a strip's transformation matrices, counting from the end. It computes the transformation matrices $(M_1, \ldots, M_k)$ with 'MList,' and then returns $(M_1 \cdots M_k,\ M_2 \cdots M_k,\ \ldots,\ M_k,\ \text{ID})$, where ID is the 3× 3 identity matrix.

3.3.10 r

Takes: Positive Integer
Returns: Positive Integer

r is designed so that MList⟦r[i]⟧= M[i$^{\text{th}}$ fold], i.e., r increments i past however many turns have happened so far. r is a kind of complement to 'TurnSpots' in that the values skipped by r are exactly those given by 'TurnSpots.' r is useful when we want to calculate partial derivatives of a product of matrices and don't want any terms corresponding to the (unchangable) turns.

r is defined inside the functions that use it.

3.4 Drawing strips

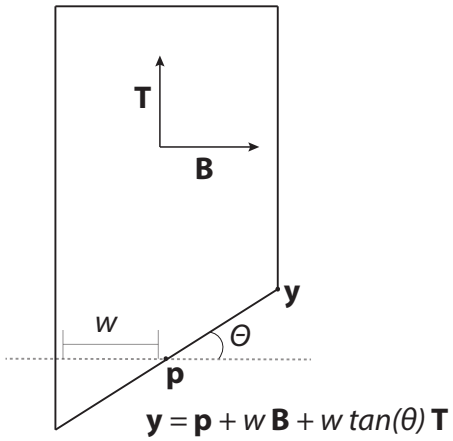
This section allows us to see in a graphics window what strips look like.

3.4.1 StripPolys

Takes: List of a Strip ID
Returns: List of lists of four xyz points. Each set of four points specifies a quadrilateral to be drawn.

StripPolys calculates the polygons to be drawn to visualize a strip. Strips extend a fixed width to the left and right (i.e., the direction of the binormal and its opposite) of their primary direction of travel (i.e., the tangent). At any fold, we can calculate the position of the left and right edge points of the strip as a function of the center point of the fold, the tangent, the binormal, and the lateral angle θ (see figure 3). This formula conveniently works for turns as well as folds, as long as we pretend that we're turning by -1/2 the turn angle (to convince yourself of this, test a couple turn angles.). We add θs of 0 at the beginning and end of the strip to ensure straight cuts there. Once we have these values, we tabulate them and organize them into quadrilaterals.

Figure 3: Calculating endpoints of a fold to draw them



3.4.2 DrawAll

Takes: List of Strip IDs
Returns: 3D Graphics window

DrawAll produces a graphics window featuring all the strips in its argument. It applies StripPolys to each strip, makes one big table with all the necessary quadrilaterals, and draws them with the "Polygon" option of "Graphics3D."

4 Optimization

4.1 Gradients

4.1.1 EnergyGrad

Takes: A StripID
Outputs: A list with twice as many elements as there are folds in the given strip.

Outputs the energy gradient of the given strip. Let Q denote the strip's total energy as measured by CostCircles, and let θ_i, and ϕ_i denote the values of *Theta* and *Phi* associated to the ith fold of the strip. Then the output of EnergyGrad is $\{\frac{\partial Q}{\partial \theta_1}, \frac{\partial Q}{\partial \phi_1}, \frac{\partial Q}{\partial \theta_2}, \frac{\partial Q}{\partial \phi_2}, ... \frac{\partial Q}{\partial \theta_k}, \frac{\partial Q}{\partial \phi_k}\}$.

4.1.2 ETNGrads

Takes: A StripID S
Outputs: A list of gradients for each of the three constraint curves (each one a list).

Let E denote the endpoint of S, and let E_f denote the desired endpoint. Similarly, let T and T_f denote the actual and desired ending unit tangent vectors, and let N and N_f denote the actual and desired ending normal vectors. The scalar values $E^* = \| E - E_f \|^2$, $T^* = \| T - T_f \|^2$, and $N^* = \| N - N_f \|^2$ measure deviations from each of the three constraints; each is zero when the corresponding constraint is satisfied and nonzero otherwise.

Thinking of E^* as a function of the values $\theta_1, \phi_1, \theta_2, \phi_2, ...\theta_k, \phi_k$ associated to the folds S, we compute the gradient $\nabla E^* = (\frac{\partial E^*}{\partial \theta_1}, \frac{\partial E^*}{\partial \phi_1}, \frac{\partial Q}{\partial \theta_2}, \frac{\partial E^*}{\partial \phi_2}, ... \frac{\partial E^*}{\partial \theta_k}, \frac{\partial E^*}{\partial \phi_k})$. Similar computations generate ∇T^* and ∇N^*. ETNGrads does these computations and outputs $\{\nabla E^*, \nabla T^*, \nabla N^*\}$.

Here's how we computed those partial derivatives. Let M_0 be the starting orientation and $\{M_i = M(\theta_i, \phi_i)\}$ the fold matrices. The endpoint constraint is

$$E \quad = \quad \text{starting pt} + (M_0 + M_0M_1 + \ldots + M_0 \cdots M_k) \cdot (0,1,0)^T,$$

the sum of the starting point and the tangent on each segment. We know that $\| \mathbf{x} \|^2 = \mathbf{x} \cdot \mathbf{x}$ for all $\mathbf{x}$, and that $(\mathbf{x} \cdot \mathbf{x})' = 2\mathbf{x} \cdot \mathbf{x}'$; therefore

$$\begin{aligned}
\frac{\partial E^*}{\partial \theta_i} &= \frac{\partial}{\partial \theta_i} \| E - E_f \|^2 \\
&= 2(E - E_f) \cdot \frac{\partial}{\partial \theta_i}(E - E_f) \\
&= 2(E - E_f) \cdot \frac{\partial}{\partial \theta_i}[(M_0 \cdots M_i + \ldots + M_0 \cdots M_k) \cdot (0,1,0)^T] \\
&= 2(E - E_f) \cdot M_0 \cdots M_{i-1} \frac{\partial M_i}{\partial \theta_i} \cdot (ID + M_{i+1} + \ldots + M_{i+1} \cdots M_k) \cdot (0,1,0)^T.
\end{aligned}$$

Partials with respect to ϕ_i are identical. Our "PartialTails" operation computes the matrix sum in the above formula, and "EndPointThetaPartials" (or Phi) computes the entire formula.

Regarding the tangent constraint, we note that

$$T \quad = \quad M_0M_1M_2M_3 \cdots M_k \cdot (0,1,0)^T$$

and that

$$\frac{\partial T}{\partial \theta_i} \quad = \quad M_0M_1M_2M_3 \cdots \frac{\partial M}{\partial \theta_i} M_{i+1} \cdots M_k.$$

Using the "HeadList" function, the "TailList" function, and the "r" function, we express this in our code by noting that the left-hand part of the above equation is the $r[i]^{\text{th}}$ term in HeadList[S,MList[S]], and that the right-hand part is $(\text{r}[i]+1)^{\text{st}}$ term in TailList[S]. In code, therefore,

$$\frac{\partial T}{\partial \theta_i} \quad = \quad \text{HeadList}[S,\text{MList}[S]][\![\text{r}[i]+1]\!] \cdot dMd\theta[\theta_i, \phi_i] \cdot \text{TailList}[S][\![\text{r}[i+1]]\!].$$

To calculate $\frac{\partial}{\partial \theta_i} \| T - T_f \|^2$, we use the same trick as above: $\frac{\partial}{\partial \theta_i} \| T - T_f \|^2 = 2(T - T_f) \cdot \frac{\partial}{\partial \theta_i} T$.

In the code, the quantity $T - T_f$ is called "EndTangentDiff," and the factor of 2 is dropped. The list named "TangentThetaPartials" computes $\frac{\partial}{\partial \theta_i} \| T - T_f \|^2$ for each valid i, and returns a table of all of them. The list named "TangentPhiPartials" does the same for ϕ, using an analogous method.

We apply the same reasoning, replacing T with N, to get the lists NormalThetaPartials and NormalPhiPartials describing the behavior of the normal.

4.2 Step Functions

4.2.1 StepDownEndPoint

Takes: A StripID S and a stepsize k
Outputs: A modified StripID and the same stepsize k

let θ_i, and ϕ_i denote the values of *Theta* and *Phi* associated to the ith fold of S. The configuration of S is characterized by $\mathbf{C} = \theta_1, \phi_1, \theta_2, \phi_2, ...\theta_k, \phi_k$. StepDownEndPoint perturbs these values by an amount proportional to k in an attempt to bring the endpoints of S closer together.

Define $E^* : \mathbb{R}^{2k} \to \mathbb{R}$ as above. Because $E^* = 0$ when the endpoint constraint is satisfied and $E^* > 0$ otherwise, the space of configurations with a valid endpoint is the level set $\{E^* = 0\}$. Generally speaking, then, decreasing E^* corresponds to moving the endpoints closer together.

By definition, $-\nabla E^*$ gives the direction of greatest decrease for E^*, so that $E^*(C - \lambda(\nabla E^*)(C)) < E^*(C)$ for small λ. The function chooses a step size (called OtherStepSize) based on how far away the endpoints are in the starting configuration C: the further apart, the larger the step size. It then scales this step size by the input value k and alters S accordingly. The configuration C of S becomes $C - \text{OtherStepSize} * k * (\nabla E^*)(C)$. k is not altered, but is returned along with the altered strip for convenience.

While *Mathematica* is not object-oriented, it is most intuitive to view StepDownEndPoint as a mutator method on the "object" S. The name of S remains unchanged, but its attributes change. The optimization method used here is commonly known as *gradient descent*.

4.2.2 MeetEndPoints

Takes: A StripID S and a stepsize k
Outputs: A modified S and the same stepsize k

Continually calls StepDownEndPoint$[\{S, k\}]$ until the endpoint of S is within a set tolerance of the desired endpoint.

4.2.3 StepDownTangent

Takes: A StripID S and a stepsize k
Outputs: A modified S and the same stepsize k

Analogous to StepDownEndPoint, this function changes the configuration of S slightly in an attempt to move its ending tangent vector closer to the desired ending tangent. The method used is exactly the same: $(\nabla T^*)(S)$ provides a viable step direction, and step size is chosen proportional to both the input k and the norm of the difference between the actual and desired tangent.

4.2.4 StepDownNormal

Takes: A StripID S and a stepsize k
Outputs: A modified S and the same stepsize k

Analogous to StepDownTangent; replace "tangent" with "normal."

4.3 Satisfying Constraints

4.3.1 FixConstraints

Takes: A StripID S and a stepsize k
Outputs: A modified S and the same stepsize k

Summarized by the following four steps:

1. Place the endpoint of S in the right place (see MeetEndPoints).
2. Call StepDownTangent$[\{S, k\}]$ until the tangent is within a specified tolerance of the desired tangent.
3. Repeat step 2 for the normal vector.
4. If all three constraints are met (to within some tolerance), return. Otherwise, go back to step 1.

Notice that three of these steps are optimizations of their own, and the fourth calls for repeating the first three. While this type of "loop within loop" structure proved unavoidable, FixETN (below) achieves the goal of fixing all constraints more elegantly. In particular, we can improve one constraint without ruining another too much.

4.3.2 FixETN

Takes: A StripID S and a stepsize k
Outputs: A modified S and the same stepsize k

A more sophisticated version of FixConstraints. Rather than stepping blindly down each constraint function, we take care not to disrupt other constraints each time we step. Henceforth, a "valid" configuration is one that satisfies all constraints.

Consider once again the vector $\mathbf{C} = (\theta_1, \phi_1, \theta_2, \phi_2, ...\theta_k, \phi_k)$ of fold angles associated to S. These entirely characterize the configuration of S in space, so that the set of all configurations satisfying a given constraint is a subset of $\mathbb{R}^{2k}$. This suggests a way to move towards one constraint space while remaining in another: *project the step vector onto the tangent space to the second constraint curve at C*. In this way, we remain close to the second constraint space but still land closer to the first.

In order to accomplish such a projection, we need to know the normal to the space corresponding to the constraint that is to be preserved. This is roughly approximated by the normalized gradient of that constraint. Using these methods, we generate the following improved step functions:

1. **StepDownTanE** moves towards the correct final tangent while preserving the correct endpoint.
2. **StepDownNormT** moves towards the correct final normal while preserving the correct tangent.
3. **StepDownEndN** moves towards the correct endpoint while preserving the final normal.

FixETN behaves like FixConstraints, but with these improved step functions.

4.4 Optimizing Under Constraints

4.4.1 Optimize

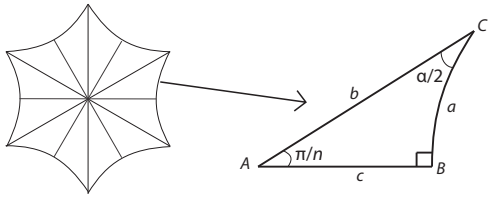
Takes: A StripID S with all constraints satisfied, and a stepsize k
Outputs: A modified S and the same stepsize k

An attempt to decrease the total energy of the given strip while still maintaining all constraints. Proceeds by the following steps:

1. Compute all gradients of the given strip (via ETNGrads and EnergyGrad).
2. Orthogonalize the three constraint gradients. This gives an orthonormal basis for the orthogonal compliment to the set of valid configurations.
3. Using this basis, project the energy gradient so that it lies in the tangent space to the set of valid configurations. This projected gradient is labeled "EnergyAlong."
4. Take a step down EnergyAlong (i.e. in the direction of -EnergyAlong) proportional to k and see if the total energy decreases. If not, undo the step, halve k, and repeat. Continue until a suitable step size is found.
5. Using FixETN, fix the constraints of the resulting strip. Compare the total energy of the current strip to the initial total energy (at step 1). If the energy has decreased overall, return. Otherwise, halve the current k again and repeat from step 1.

The method is recursive; it calls itself with successively smaller k-values until a suitable one is found. Currently, Print statements in the code output pictures after each recursion and messages indicating where the recursion occured. The recursion is guaranteed to end, because a sufficiently small step down the energy gradient always decreases the energy.

Figure 4: Triangulating a regular hyperbolic n-gon with interior angles α



5 Toward Polygon Meshes

5.1 Strip n

This section lets us retrieve information about position and orientation anywhere along a strip. This allows us to "tape" struts partway along a strip.

5.1.1 GetPositionAlong

Takes: List containing a Strip ID and a real number $\in [0, 1]$
Returns: XYZ coordinates of a point

GetPositionAlong yields the coordinates of the point any fraction of the way along the total length of a strip. This point is the sum of three things: the starting point of the strip; displacement due to a number of full segments; and displacement due to part of a final segment. The integer 'index' measures how many full segments are traversed. Segments 1-index displace by the sum of their tangents, scaled by the segment length, and the final segment's tangent must be scaled by both SegmentLength and the fraction by which it is traversed. The function is piecewise to avoid out-of-bounds indices in HeadPart.

5.1.2 GetOrientationAlong

Takes: List containing a Strip ID and a real number $\in [0, 1]$
Returns: 3×3 matrix

GetOrientationAlong yields the orientation matrix at a point any fraction of the way along the total length of a strip. This depends only on which segment the point is on. 'Index' calculates which segment this is. If the point is exactly on a fold or turn, the next segment is used, excepting only the endpoint of the strip, which uses the last segment.

5.2 Hyperbolic polygon tools

This section has functions for calculating lengths in regular hyperbolic polygons, and for automating the initialization of a regular n-gon.

5.2.1 CentertoMidpoint

Takes: Integer ≥ 3 and angle $\in (0, \pi)$
Returns: Real number

CentertoMidpoint uses the hyperbolic law of cosines to calculate the distance from the center of a regular n-gon with interior angles α to the midpoint of a side. Only works if arguments specify a shape that is actually hyperbolic. See figure 4.

5.2.2 CentertoCorner

Takes: Integer ≥ 3 and angle $\in (0, \pi)$
Returns: Real number

CentertoCorner uses the hyperbolic law of cosines to calculate the distance from the center of a regular n-gon with interior angles α to a corner. Only works if arguments specify a shape that is actually hyperbolic. See figure 4.

5.2.3 NgonHalfSidelength

Takes: Integer ≥ 3 and angle $\in (0, \pi)$
Returns: Real number

NgonHalfSidelength uses the hyperbolic law of cosines to calculate half the sidelength of a regular n-gon with interior angles α. Only works if arguments specify a shape that is actually hyperbolic. See figure 4.

5.3 Polygon Meshes

To make a polygon mesh, start by making a polygon border with MakeNgonBorder. Optimize it. Then add a single strut across the center with MakeCenterStrut. Optimize it. Then add more struts as preferred with MakeSideStrut and MakeCornerStrut, optimizing each one manually.

The recommended optimization for each strip consists of FixETN followed by repeating Optimize until the energy is not changing much with each step. These functions might take dozens of seconds. If FixETN doesn't return, you may need to fudge the length of the strut a bit- make it 2% longer and try again.

It's best to add struts one at a time, relaxing each one manually. We weren't able to automate the process of adding an entire level of struts at once.

5.3.1 MakeNgonBorder

Takes: List containing an integer ≥ 3 and an angle $\in (0, \pi)$
Returns: Strip ID

MakeNgonBorder initializes a strip with the values and size of a regular n-gon with interior angles α. Total length is only correct for hyperbolic shapes. Strip is given an arbitrary inital curvature, and no optimization is included.

5.3.2 MakeCenterStrut

Takes: Strip ID of a hyperbolic polygon
Returns: New Strip ID

MakeCenterStrut initializes a strip running across the center of the given polygon. Argument must be a regular hyperbolic polygon. No optimization is included.

5.3.3 MakeSideStrut

Takes: List containing the ID of a hyperbolic n-gon, the ID of its center strut, and an integer i from 1 to $n-1$
Returns: New Strip ID

MakeSideStrut initializes a strip running from the center point of the center strip to the midpoint of the $(i+1)^{\text{st}}$ side of the polygon. No optimization is included.

5.3.4 MakeCornerStrut

Takes: List containing the ID of a hyperbolic n-gon, the ID of its center strut, and an integer i from 1 to n
Returns: New Strip ID

MakeCornerStrut initializes a strip running from the center point of the center strip to the i^{th} vertex of the polygon. No optimization is included.

6 Future Directions

The clearest path of research is to finish the simulation of meshes by allowing an entire mesh to relax at once. This would entail not only streamlining and effectively automating the process of relaxing interior strips to fit the form of an exterior boundary, but also allowing these strips to push at the boundary. Once a global energy minimum is found for the entire mesh, the model of physical paper meshes is complete.

Generally, we would like to know where and how the paper model fails. It would be instructive to attempt to apply the central argument of Hilbert's embedding theorem to the fully relaxed meshes. This would involve tracking the *asymptotic curvature lines* on which the mean curvature is zero. According to Hilbert's theorem, these lines should become arbitrarily dense as the area represented grows, yielding a singularity that prevents completeness of the surface. Whether the error lies in the curvature, smoothness, or regularity of the meshes has yet to be discerned.