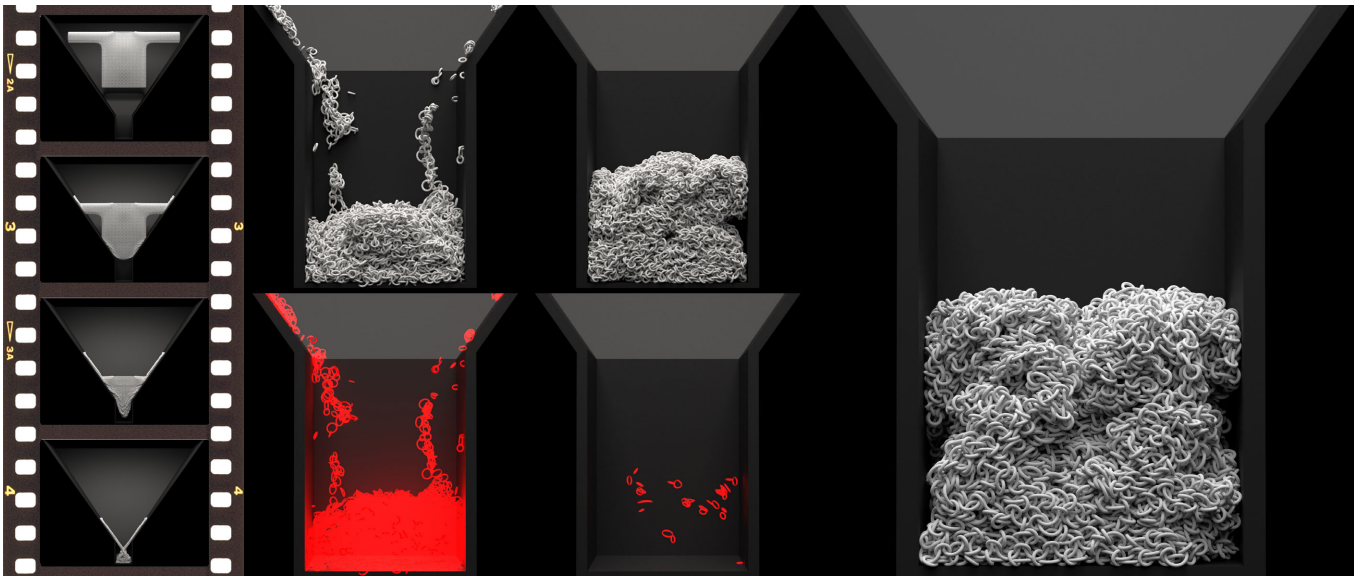
Fast Linking Numbers for Topology Verification of Loopy Structures

ANTE QU and DOUG L. JAMES, Stanford University, USA



(a) Animation (b) $1/\Delta t = 480\,Hz$ (c) $1/\Delta t = 1200\,Hz$ (d) $1/\Delta t = 2400\,Hz$

Fig. 1. **Topologically verified rigid-body simulation of fitted Kusari chainmail** (a) is used to pack the garment of 14112 curved rings with 18752 links into a box for 3D printing. Our fast linking number methods can detect pull-through between previously linked curved loops that result from large timesteps: (b) widespread failure when timestepped at $f = 480\,Hz$ with 4048 destroyed links shown in red; (c) timestepping at $f = 1200\,Hz$ is much better and looks visually fine, but still has 16 failed links hiding inside the pile; whereas (d) using $f = 2400\,Hz$ we can verify that there are no violated links (or spurious new ones) in the final result. Rapid topology verification also allows corrupted simulations to be detected and aborted early.

It is increasingly common to model, simulate, and process complex materials based on loopy structures, such as in yarn-level cloth garments, which possess topological constraints between inter-looping curves. While the input model may satisfy specific topological linkages between pairs of closed loops, subsequent processing may violate those topological conditions. In this paper, we explore a family of methods for efficiently computing and verifying linking numbers between closed curves, and apply these to applications in geometry processing, animation, and simulation, so as to verify that topological invariants are preserved during and after processing of the input models. Our method has three stages: (1) we identify potentially interacting loop-loop pairs, then (2) carefully discretize each loop's spline curves into line segments so as to enable (3) efficient linking number evaluation using accelerated kernels based on either counting projected segment-segment crossings, or by evaluating the Gauss linking integral using direct or fast summation methods (Barnes-Hut or fast multipole methods). We evaluate CPU and GPU implementations of these methods on a suite of test problems, including yarn-level cloth and chainmail, that involve significant processing: physics-based relaxation and animation, user-modeled deformations, curve compression and reparameterization. We show that topology errors can be efficiently identified to enable more robust processing of loopy structures.

CCS Concepts: • **Computing methodologies** → **Animation**; *Shape modeling*; • **Theory of computation** → Computational geometry.

Additional Key Words and Phrases: Topology, checksum, linking number, Gauss linking integral, Barnes-Hut, fast multipole method, yarn-level cloth

ACM Reference Format:
Ante Qu and Doug L. James. 2021. Fast Linking Numbers for Topology Verification of Loopy Structures. *ACM Trans. Graph.* 40, 4, Article 106 (August 2021), 19 pages. https://doi.org/10.1145/3450626.3459778

Authors' address: Ante Qu, antequ@cs.stanford.edu; Doug L. James, djames@cs.stanford.edu, Computer Science, Stanford University, 353 Jane Stanford Way, Stanford, CA, 94305, USA.

0730-0301/2021/8-ART106 $15.00
https://doi.org/10.1145/3450626.3459778

1 INTRODUCTION

It is increasingly common to model, simulate, and process complex materials based on loopy structures, such as in knitted yarn-level cloth garments, which possess topological constraints between inter-looping curves. In constrast to a solid or continuum model, the integrity of a loopy material depends on the preservation of its loop-loop topology. A failure to preserve these links, or an illegal creation of new links between unlinked loops, can result in an incorrect representation of the loopy material.

Unfortunately, common processing operations can easily destroy the topological structure of loopy materials. For example, time-stepping dynamics with large time-steps or contact solver errors that cause pull-through events, deformation processing that allows loops to separate or interpenetrate, or even applying compression and reparameterization on curves, can all cause topology errors. Furthermore, once the topology of a loopy material has been ruined, the results can be disastrous (an unravelling garment) or misleading (incorrect yarn-level pattern design).

In this paper, we explore a family of methods for efficiently computing and verifying *linking numbers* between closed curves. Intuitively if two loops are unlinked (i.e., they can be pulled apart), the linking number is zero; otherwise, the linking number is a nonzero signed integer corresponding to how many times they loop through one another, with a sign to disambiguate the looping direction (see Figure 2). In mathematical terms, the linking number is a homotopical invariant that describes the "linkage" of two oriented closed curves in 3D space. While a closed curve can be either a "knot" or a simple "loop", we will use the term "loop" to refer to all closed curves. Given two disjoint, oriented loops γ_1 and γ_2 where each maps $S^1 \to \mathbb{R}^3$, the linking number is an integer-valued function, $\lambda(\gamma_1, \gamma_2)$, that counts the number of times each curve winds around the other [Rolfsen 1976]. Curve deformations that result in one part of a curve crossing the other loop at a single point must change the linking number by ± 1. By evaluating (or knowing) the linking numbers before a computation, then evaluating them afterwards, we can detect certain topological changes. A linking number checksum can therefore be useful as a sanity check for topology preservation—while we cannot rule out intermediate topology changes when two states have the same linking number, different linking numbers imply a topology change. We efficiently and systematically explore this idea and apply it to several applications of loopy materials in computer graphics where it is useful to verify the preservation of topological invariants during and after processing (see Figure 1 for a preview of our results).

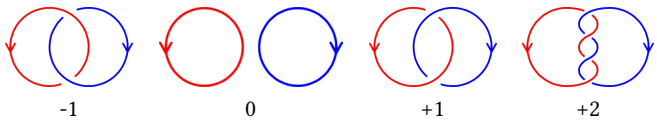


Fig. 2. **Linking numbers** count the oriented linkages between two closed curves (examples from https://en.wikipedia.org/wiki/Linking_number).

Our method has three stages: (1) to avoid computing linking numbers between obviously separated loops, we first perform a pair search to identify potentially linked loops; (2) we discretize each loop's spline curves into line segments for efficient processing, being careful to ensure that the discretized collection of loops remain topologically equivalent to the input; then (3) we evaluate all loop-loop linking numbers (and record the results in a sparse matrix) using one of several accelerated linking-number kernels: either (i) by counting projected segment-segment crossings, or (ii) by evaluating the Gauss linking integral using direct or fast summation methods (both Barnes-Hut and fast multipole methods). An overview of our method is shown in Figure 4. We evaluate CPU and GPU implementations of these methods on a suite of test problems, including yarn-level cloth and chainmail examples, that involve significant processing: physics-based relaxation and animation, user-modeled deformations, curve compression and reparameterization. From our evaluation we conclude that counting crossings and the Barnes-Hut method are both efficient for computing the linking matrix. In addition to loopy structures, we also show an example where our method is part of an automatic procedure to topologically verify braids, which are structures with open curves that all attach to two rigid ends; in particular, we verify the relaxation of stitch patterns by attaching the ends of stitch rows to rigid blocks. We show that many topology errors can be efficiently identified to enable more robust processing of loopy structures.

2 BACKGROUND

2.1 Computing the Linking Number

There are several equivalent ways to calculate (and define) the linking number, all of which will be explored in this paper. The linking number, $\lambda(\gamma_1, \gamma_2)$, is a numerical invariant of two oriented curves, γ_1, γ_2. It intuitively counts the number of times one curve winds around the other curve (see Figure 2). The linking number is invariant to link homotopy, a deformation in which curves can pass through themselves, but not through other curves [Milnor 1954].

Counting Crossings. One way to compute the linking number is to count crossings in a "link diagram". Suppose our curves are discretized into polylines. The link diagram is a 2D "regular projection" of this set of 3D curves that also stores the above-belowness at every intersection; that is, if the projection plane is the XY plane, then the Z coordinate of each curve indicates which curve is above or below. A projection is deemed regular if no set of 3 or more points, or 2 or more vertices, coincide at the same projected point [Rolfsen 1976].

To compute the linking number, start with $\lambda(\gamma_1, \gamma_2) = 0$, and at every detected intersection, compare the orientation of the curves with Figure 3, and increment the linking number by $+1/2$ if it is positive, or $-1/2$ if it is negative.

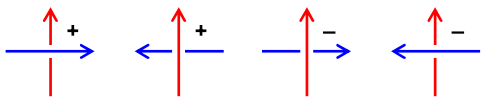


Fig. 3. **Crossing orientation** determines whether to increment or decrement the linking number. (From https://en.wikipedia.org/wiki/Linking_number).

Gauss's Integral Form. Another mathematically equivalent method to compute the linking number, which doesn't require a projection plane or geometric overlap tests, is to compute the linking integral λ, first introduced by Gauss [Rolfsen 1976]:

$$\lambda(\gamma_1, \gamma_2) = \frac{1}{4\pi} \int_{\gamma_1} \int_{\gamma_2} \frac{\mathbf{r}_1 - \mathbf{r}_2}{|\mathbf{r}_1 - \mathbf{r}_2|^3} \cdot (d\mathbf{r}_1 \times d\mathbf{r}_2). \tag{1}$$

This integral form can be interpreted as an application of Ampere's law from magnetostatics for a current loop and a test loop[1].

Direct Summation Methods. If the loops γ_1, γ_2 are discretized into line segments indexed by j and i, and we denote the midpoints of γ_1 by $\mathbf{r}_j$ and γ_2 by $\mathbf{r}_i$, and the line segment length vectors by $\mathbf{s}_j$ and $\mathbf{s}_i$ respectively, then taking a midpoint quadrature, we have

$$\lambda(\gamma_1, \gamma_2) \approx \frac{1}{4\pi} \sum_{j,i} \frac{\mathbf{r}_j - \mathbf{r}_i}{|\mathbf{r}_j - \mathbf{r}_i|^3} \cdot (\mathbf{s}_j \times \mathbf{s}_i). \tag{2}$$

This midpoint approximation is only accurate when the segments are short compared to the distance between each pair. Fortunately, an exact expression for polyline loops is known of the form

$$\lambda(\gamma_1, \gamma_2) = \sum_{j,i} \lambda_{ji} \tag{3}$$

where λ_{ji} is the contribution from a pair of line segments [Arai 2013; Berger 2009]. If we denote the vertices (endpoints of the segments) of each loop by $\mathbf{l}_j, \mathbf{k}_i$, the contribution from the segment pair (j, i) is given in [Arai 2013] (which uses the signed solid angle formula [Van Oosterom and Strackee 1983]):

$$\lambda_{ji} = \frac{1}{2\pi} \left(\operatorname{atan}\left(\frac{\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})}{|\mathbf{a}||\mathbf{b}||\mathbf{c}| + (\mathbf{a} \cdot \mathbf{b})|\mathbf{c}| + (\mathbf{c} \cdot \mathbf{a})|\mathbf{b}| + (\mathbf{b} \cdot \mathbf{c})|\mathbf{a}|} \right) + \operatorname{atan}\left(\frac{\mathbf{c} \cdot (\mathbf{d} \times \mathbf{a})}{|\mathbf{c}||\mathbf{d}||\mathbf{a}| + (\mathbf{c} \cdot \mathbf{d})|\mathbf{a}| + (\mathbf{a} \cdot \mathbf{c})|\mathbf{d}| + (\mathbf{d} \cdot \mathbf{a})|\mathbf{c}|} \right) \right), \tag{4}$$

where

$$\mathbf{a} = \mathbf{l}_j - \mathbf{k}_i, \quad \mathbf{b} = \mathbf{l}_j - \mathbf{k}_{i+1}, \quad \mathbf{c} = \mathbf{l}_{j+1} - \mathbf{k}_{i+1},$$
$$\mathbf{d} = \mathbf{l}_{j+1} - \mathbf{k}_i, \quad \operatorname{atan}(y/x) = \operatorname{atan2}(y, x).$$

This computation requires two trigonometric operations per segment pair. More recently, Bertolazzi et al. [2019] used the arctan addition formula to eliminate all trigonometric operations.

Fast Summation Methods. If we discretize the loops into N samples, Gauss's double integral can be approximated using fast summation methods, such as the Barnes-Hut algorithm which uses a tree to compute N-body Gaussian summations in $O(N \log N)$ time. Alternately one can use the Fast Multipole Method (FMM); some FMM libraries provide off-the-shelf implementations of the Biot-Savart integral, which is the inner integral of the linking integral.

2.2 Related Work

Many works in the past have tackled the topic of robust collision processing. Epsilon geometry and robust predicates [Salesin et al. 1989; Shewchuk 1997] have been used to precisely answer "inside-outside" questions, such as where a point lies with respect to a flat face ("vertex-face"), or a straight line with respect to another ("edge-edge"). These determinant-based geometric tests (such as [Feito and Torres 1997]) are also useful for detecting inversions in finite elements and preventing them from collapsing [Irving et al. 2004; Selle et al. 2008]. Some papers [Harmon et al. 2011; Kim et al. 2019; Selle et al. 2008; Volino and Magnenat-Thalmann 2006; Zhang et al. 2007] propose methods to repair interpenetrations between objects by minimizing a computable quantity, such as an intersection contour, a space-time intersection volume, an altitude spring, or another form of energy. For models with a large number of elements, a few works use hierarchical bounding volumes to compute energy-based collision verification certificates [Barbič and James 2010; Zheng and James 2012], or to validate geometry in hair [Kaufman et al. 2014], holey shapes [Bernstein and Wojtan 2013], or cloth [Baraff et al. 2003].

Many of the above methods either perform a check or repair a violation by answering discrete collision-detection questions about a point vs. a flat face or a straight edge vs. another straight edge. To generalize these questions to arbitrary surfaces (and soups and clouds), a few papers [Barill et al. 2018; Jacobson et al. 2013] compute generalized winding numbers to answer the inside-outside question. The generalized winding number can be expressed as a Gaussian integral, and Barill et al. [2018] uses Barnes-Hut style algorithms to speed up the computation.

Continuous collision detection (CCD) methods [Basch 1999; Brochu et al. 2012; Harmon et al. 2011; Wang and Cao 2021] can verify that a deformation trajectory is collision free. Such methods could be used to detect curve-curve crossings and thus topology changes. In contrast, our approach of computing a topology verification checksum is a discrete before/after test: we verify the topology at a fixed state rather than the geometry along the entire deformation path. While our method cannot guarantee a deformation is collision free, it works even if the deformation path is ambiguous (e.g., in verifying model reparameterizations, which we will show in §4.3.2), or, when the path is known, it can be performed as sparsely or frequently as the user requires.

For simulations and deformations that are more suitable for 1D rod elements rather than volumetric elements, it can be difficult to apply discrete inside-outside checks. For example, in yarn-level simulations of cloth, yarns are often modeled as cylindrical spline or rod segments [Kaldor et al. 2008, 2010; Leaf et al. 2018; Wu et al. 2020]. While the inside-outside question applied to the cylindrical volumes can determine interpenetration when detected, oftentimes an illegal or large simulation step can cause a "pull-through", where the end state has no interpenetration but rather a topological change in the curve configuration. Pull-throughs can cause changes to the appearance and texture of the resulting relaxed pattern, yet remain undetected unless spotted by a human eye. Many of these simulations [Leaf et al. 2018; Yuksel et al. 2012] prevent pull-throughs by using large penalty contact forces and limiting step sizes using a known guaranteed displacement limit. Some stitch meshing papers [Guo et al. 2020; Narayanan et al. 2018, 2019; Wu et al. 2018, 2019] avoid closed loops as a result of making the model knittable or machine knittable. In contrast, many of the models from [Yuksel et al. 2012] consist of yarns that form closed loops. We propose a method to automatically detect pull-throughs in closed-loop yarn models after they occur, so that the simulation can take more aggressive steps and backtrack or exit when the violation is detected. Other

[1] *Magnetostatics Interpretation:* If we imagine one loop has current flowing through it, the Gauss linking integral essentially applies Ampere's law on the other loop, using the magnetic field generated by the current loop (given to us by the Biot-Savart law), to compute the total current enclosed. Similar to the winding number computation in [Barill et al. 2018], we note that the first term of the integrand is the gradient of the Laplace Green's function $G(\mathbf{r}_1, \mathbf{r}_2) = -1/(4\pi|\mathbf{r}_1 - \mathbf{r}_2|)$; in magnetostatics, the vector potential $\mathbf{A}$ of a monopole current source at $\mathbf{r}_1$ with current I and length vector $\mathbf{s}_1$ is $\mu_0 I \mathbf{s}_1 G(\mathbf{r}_1, \mathbf{r}_2)$, and the magnetic field at $\mathbf{r}_2$ due to $\mathbf{r}_1$ is $\mathbf{B} = \nabla_2 \times \mathbf{A} = -\mu_0 I \mathbf{s}_1 \times \nabla_2 G(\mathbf{r}_1, \mathbf{r}_2)$. Applying Ampere's law gives us the linking integral. This interpretation will later allow us to apply computational tools for multipole expansions.

modeling applications in graphics that are more suitable for 1D elements include chains and chainmail [Bergou et al. 2008; Tasora et al. 2016], threads [Bergou et al. 2010], knots [Bergou et al. 2008; Harmon et al. 2011; Spillmann and Teschner 2008; Yu et al. 2020], hair [Bridson et al. 2002; Chang et al. 2002; Kaufman et al. 2014; Selle et al. 2008], and necklaces [Guibas et al. 2002].

To detect pull-through violations of edge-edge contacts, one can locally verify the sign of the volume of the tetrahedron (computed as a determinant) spanned by the two edges. To generalize this local notion to arbitrary closed curves, one can compute the linking number between the two curves. Interestingly, Gauss's integral form of the linking number uses, in the integrand, this exact determinant, applied on differential vectors, but scaled so that it is the degree, or signed area of the image, of the Gauss map of the link.

In graphics, some works [Dey et al. 2013, 2009] compute the linking number, by projecting and computing crossings, to determine if a loop on an object's surface is a handle versus a tunnel, which is useful for topological repair and surface reparameterization. Another work [Edelsbrunner and Zomorodian 2001] computes the linking numbers of simplicial complexes within DNA filtrations, by using surface crossings (which are generated from the filtration process), to detect nontrivial tangling in biomolecules. In the computational DNA topology community, many papers [Clauvelin et al. 2012; Fuller 1978; Klenin and Langowski 2000; Krajina et al. 2018; Sierzega et al. 2020] use topological invariants such as the linking number, twist, and writhe to understand DNA ribbons; in particular, Berger [2009] summarizes how to compute these quantities, and [Arai 2013; Bertolazzi et al. 2019] present more robust exact expressions for line segment curves using the signed solid angle formula [Van Oosterom and Strackee 1983]. Moore [2006] also discusses how to subdivide spline curves for computational topology. A few works in animation and robotics [Ho and Komura 2009; Ho et al. 2010; Pokorny et al. 2013; Zarubin et al. 2012] also use the linking integral as a topological cost in policy optimization, e.g., for grasping. However, none of these works propose using acceleration structures, such as Barnes-Hut trees or the Fast Multipole Method [Greengard and Rokhlin 1987], to compute the linking number between curves, mainly because the input size is sufficiently small or the problem has a special structure (e.g., simplicial complexes that come with surface crossings, or ribbons where the twist is easy to compute) that allows for a cheaper method. We propose using acceleration structures to speed up linking-number computation for general closed curves.

The linking integrand uses the Green's function for the Laplace problem in a Biot-Savart integral, and there are numerous Fast Multipole libraries for the Laplace problem, including FMM3D [Cheng et al. 1999] and FMMTL [Cecka and Layton 2015]. FMMTL provides an implementation of the Biot-Savart integral and has recently been used to simulate ferrofluids in graphics [Huang et al. 2019]. Another library [Burtscher and Pingali 2011] implements a very efficient, parallel Barnes-Hut algorithm for the N-Body Laplace problem on the GPU.

3 FAST LINKING NUMBER METHODS

In this section, we describe several ways to compute linking numbers (based on Algorithm 1) which all take an input model and generate the sparse upper-triangular linking matrix as a topology-invariant certificate for verification.

ALGORITHM 1: Method ComputeLinkingNumbers

Input : $\Lambda = \{\gamma_i\}$, a list of looped curves.
Output: M, a sparse triangular linking matrix of integers.

Function `ComputeLinkingNumbers(`Λ`)`**:**
  $M \leftarrow 0$;
  $P \leftarrow$ `PotentialLinkSearch(`Λ`)`;
  $\{\Gamma_i\} \leftarrow$ `Discretize(`Λ, P`)`;
  foreach $(i, j) \in P$ **do**
    $M_{ij} \leftarrow$`ComputeLink(`Γ_i, Γ_j`)`;
  end
  return M;

3.1 Method Overview

The core method takes as input a model of closed loops defined either by line segment endpoints or polynomial spline control points, and outputs a certificate consisting of a sparse triangular matrix of pairwise linking numbers between loops. Our goal is to robustly handle a wide range of inputs, such as those with a few very large intertwined loops, e.g., DNA, or many small loops, e.g., chainmail.

The main method "ComputeLinkingNumbers" is split into three stages (which are summarized in Figure 4 and described in pseudocode in Algorithm 1):

(1) Potential Link Search (PLS) (§3.2): This stage exploits the fact that any pair of loops with disjoint bounding volumes have no linkage. It takes a sequence of loops and produces a list of potentially linked loop pairs by using a bounding volume hierarchy.
(2) Discretization (§3.3): This stage discretizes the input curves, if they are not in line segment form, into a sequence of line segments for each loop, taking care to ensure that the process is link homotopic (i.e., if we continuously deform the original curves into the final line segments, no curves cross each other) and thus preserves linking numbers.
(3) Linking Number Computation (§3.4): This stage computes the linking matrix for the potentially linked loops, and can be performed with many different linking-number methods.

Fast Verification. Verification of the linking matrix can be done exhaustively by computing and comparing the full linking matrix computed using Algorithm 1. However, in some applications it is sufficient to know that any linkage failed, e.g., so that a simulation can be restarted with higher accuracy. In such cases, an "early exit" strategy can be used to quickly report topological failure by exiting when any linking number differs from the input.

Bounding Volumes. All bounding volumes in our method are convex. The convexity provides guarantees in §3.2 and §3.3.

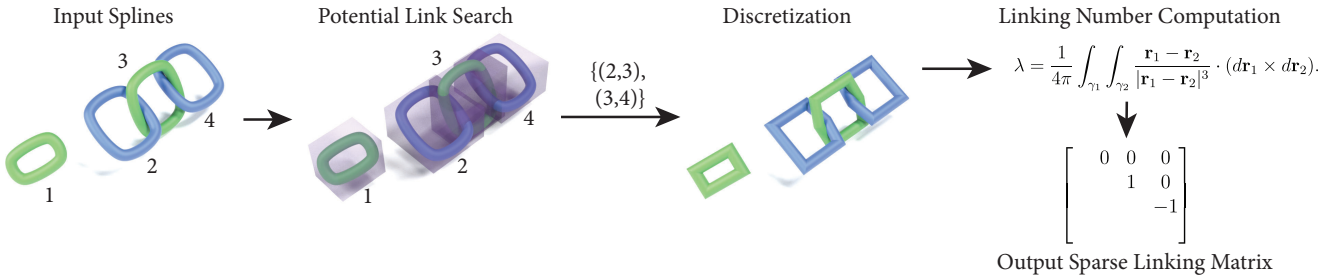


Fig. 4. **Method Overview:** Our method inputs a set of spline curves and outputs a sparse triangular linking number matrix as a topology invariant. First, (§3.2) we perform a potential link search to get a list of potentially linked loop pairs. Second, (§3.3) we discretize the input into a set of homotopically equivalent polylines. Third, (§3.4) we compute the linking number for each potentially linked pair, and output the sparse linking matrix.

3.2 Potential Link Search (PLS)

When the number of loops, L, is large, computing the linking integral between all loop pairs can be expensive, as there are $\binom{L}{2} = L(L-1)/2$ pairs. For this stage of the method, we input a list of loops $\{\gamma_i\}$, and output a list of potentially linked loop pairs, $\{(i, j)\}$ (where $i < j$ are loop indices), which gets passed to the next stages (§3.3 and §3.4). All other loop pairs must have 0 linkage.

We exploit the fact that if two loops have a nonzero linkage, then their bounding volumes, which are convex, must overlap. Therefore, we precompute an axis-aligned bounding box (AABB) for each loop, and build an AABB-tree of loops. Edelsbrunner and Zomorodian [2001] also use bounding boxes to cull the list of potentially linked loops. For any two loops with overlapping bounding volumes, we add the pair to a list of potentially linked loops. See Algorithm 2 for pseudocode, and Figure 4 for an illustration.

ALGORITHM 2: Potential Link Search

Input : $\Lambda = \{\gamma_i\}$, a list of looped curves.
Output: $P = \{(i_k, j_k)\}$, a list of pairs of loop indices that potentially link.

```
// This function returns a list of pairs of loops that have overlapping
   bounding boxes.
Function PotentialLinkSearch(Λ):
    Boxes ← {};
    for i ← 0 to |Λ| − 1 do
        Boxes[i] ← ComputeBoundingBoxOfLoop(γ_i);
    end
    Tree ← BuildBVH(Boxes);
    I ← GetIntersectingBoxes(Tree, Tree);
    P ← {(i, j) : (i, j) ∈ I ∧ (i < j)};
    return P ;
```

3.3 Discretization

Given a set of input loops discretized into polynomial splines or line segments and a set of overlapping loop pairs, our discretization step computes a set of corresponding loops discretized into homotopically equivalent line segments, so that we can use the method in the next stage (Section 3.4) to compute the exact linkage. See Figure 5 for an illustration. While there are many ways to refine

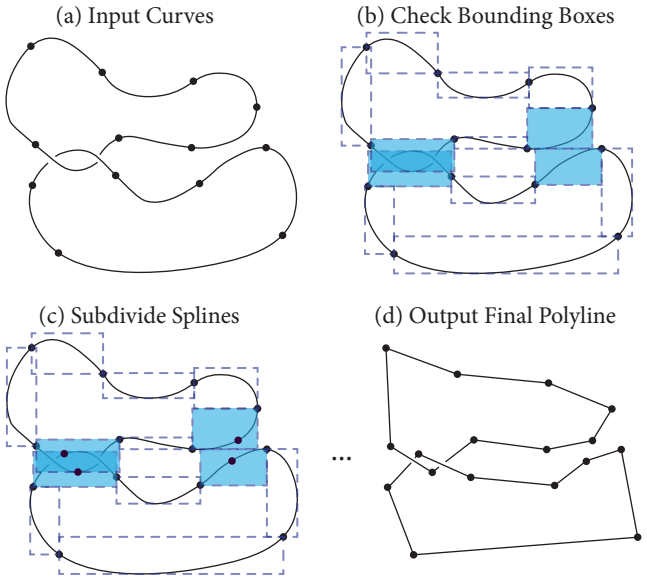


Fig. 5. **Discretization Illustration:** (a) Input curves with spline knots illustrated. (b) Given this input, we compute bounding boxes (in 3D) and find splines with overlapping boxes. (c) We refine each spline with an overlapping box by subdividing it into two splines, and we repeat steps (b) and (c) until no boxes overlap. (d) The result of discretization is a set of polylines homotopically equivalent to the input.

spline segments into line segments [De Casteljau 1959; Micchelli and Prautzsch 1989], many applications simply refine them to a fixed tolerance. Our method is spatially adaptive and refines the spline further only when curves from different loops pass closely by each other. The discretization routine takes advantage of the fact that if two convex volumes don't overlap, then two curves bounded by these volumes can be deformed into two line segments within these volumes without crossing each other. Example pseudocode is provided in Algorithm 3, and the routine is as follows.

Suppose every spline segment is parameterized in t as $\mathbf{p}(t)$. We take multiple passes. At the beginning, all spline segments are unprocessed. In each pass, we start with a list of unprocessed spline

ALGORITHM 3: Discretization

Input : $\Lambda = \{\gamma_i\}$, a list of looped curves, each of which can be a cubic spline or a polyline;
$P = \{(i_k, j_k)\}$, a list of pairs of loops that potentially link.
Output : $\{\Gamma_i\}$, a list of polylines corresponding to the curve list.

```
// This function homotopically refines input curves into polylines.
Function Discretize(Λ, P):
    L ← |Λ|;
    UnprocessedSegments ← {{}_i}; // L sets of segments.
    ξ ← the average coordinate magnitude;
    ϵ ← machine epsilon;
    // Ensure all segments have finite, nonzero length.
    parallel for i ← 0 to L − 1 do
        Γ_i ← {}; UnprocessedSegments[i] ← {};
        foreach Curve segment s ∈ γ_i do
            s.ComputeBoundingBox();
            if (DiameterOfBBox(s) < ϵξ) then
                return Error("Input has zero-length segments.");
            end
            UnprocessedSegments[i].InsertSegment(s);
        end
    end
    Trees ← {}; // One tree for each curve.
    while (At least one UnprocessedSegments[i] is not empty) do
        parallel for i ← 0 to L − 1 do
            // (I) Build an AABB-Tree for each curve.
            Trees[i]←BuildBVH(UnprocessedSegments[i]);
        end
        parallel for i ← 0 to L − 1 do
            CurrentSegments ← UnprocessedSegments[i];
            UnprocessedSegments[i] ← {};
            foreach (j : ((i, j) ∈ P or (j, i) ∈ P)) do
                // (II) Traverse BVHs, and (III-V) subdivide
                   "under-refined" segments.
                foreach (s : (s ∈ CurrentSegments and s ∈
                  GetIntersectingBoxes(Trees[i], Trees[j]))) do
                    r, t ← SubdivideSegmentIntoHalves(s);
                    r.ComputeBoundingBox();
                    t.ComputeBoundingBox();
                    if (DiameterOfBBox(r or t) < ϵξ) then
                        return Error("Curves i and j intersect.");
                    end
                    UnprocessedSegments[i].InsertSegments(r, t);
                    CurrentSegments.Exclude(s);
                end
            end
            // (VI-VIII) Collect remaining segments into results.
            Γ_i ← Γ_i ∪ CurrentSegments;
        end
    end
    return {Γ_i};
```

segments in each loop. For each loop, we (I) build a BVH of its unprocessed spline segments. If any segment has almost zero length, we exit with an error, as this indicates that either the input is corrupt or the curves are nearly intersecting. In our implementation we set the length threshold ϵ to the appropriate machine epsilon. For each loop pair, we (II) traverse the loops' BVHs, and (III) any two segments from the two different loops with overlapping bounding volumes are marked as "under-refined". After all BVH pairs are traversed, we (IV) split every "under-refined" segment into two spline segments, by keeping the same coefficients but splitting the t parameter's domain in halves, and (V) add them into a new list of unprocessed segments for the next pass. Every segment not marked "under-refined" is (VI) converted into a line segment with the same endpoints, and (VII) added to a results list for the loop. Afterwards, we repeat this process until no segments are unprocessed. At the end, we (VIII) sort the line segment list for each loop. In our implementation (Algorithm 3), all steps are parallelized across loops, and so steps (II–III) are actually ran twice for each loop pair: once from each loop in the pair.

To ensure the algorithm terminates if two loops nearly intersect, we exit with an error if any segment length gets shorter than floating-point epsilon times the average coordinate. In our examples, this limit is reached in about 45 passes for double precision, and under 20 passes for single precision, with most of the later passes only processing a small number of primitives. Exiting at this limit ensures the next stage (§3.4) does not compute linking numbers on nearly intersecting curves.

3.3.1 Proof that discretization is homotopically equivalent. For non-intersecting input curves with polynomial spline segments, we show that Algorithm 3 terminates when using tight AABBs. Also, we show that if this algorithm terminates, it produces a homotopically equivalent segmentation. These proofs can be found in Appendix A.

3.4 Linking Number Computation

Given a set of loops discretized into line segments, and a set of loop pairs to check, we wish to compute their respective linkages and output a linking matrix. We present several classes of methods here based on different prior work, and our results and analysis section (§4) will discuss the relative advantages of each method.

When running this stage on the CPU, we parallelize across the list of potentially linked loop pairs and have each thread compute its own linking number individually. We only use the parallelized version of these methods if we have very few or only one loop pair, which can arise when there are very few loops. For the GPU-accelerated methods, we just iterate through the list of loop pairs and launch the GPU kernels sequentially.

3.4.1 Method 1: Count Crossings (CC_{ann} and CC). The first approach is, given a pair of loops, find a regular projection and use it to count crossings. We modify the implementation provided with [Dey et al. 2013] for computing the linking number, and evaluate two optimized methods (CC_{ann} and CC). Their projection code takes three steps to heuristically find a regular projection: first, it uses approximate nearest neighbors (ANN) to estimate a projection direction away from all line-segment directions among the two loops,

defining an initial frame whose z axis will be the direction of projection. Next, it corrects the frame by rotating about the z axis into the direction such that all the projected segments would be oriented, at minimum, as far away as possible from vertical (y) or horizontal (x) directions. Thirdly, it rotates the frame about the y axis so that all the projected segments would be, at minimum, the farthest from being degenerate (aligned with the z axis). Each of these steps take $O(N \log N)$ time to sort the projected segment angles and perform the maximin. Afterwards, they project the segments onto the XY plane, and then perform an $O(N_k N_l)$ (where N_k, N_l are the numbers of segments in each input loop) brute-force intersection check between all segment pairs. The intersection check uses 2D exact line segment queries from CGAL [Brönnimann et al. 2021] and exits with an error when degeneracies are detected.

While this procedure is sufficient for the loops generated from the surface mesh analysis in [Dey et al. 2013], our linking number computation must support larger inputs. We accelerated the intersection search by building a BVH for one of the two loops when their average loop size exceeds 750 segments, and evaluating segments from the other loop against this tree in parallel. We refer to this modified method as CC_{ann}. The ANN call also introduces a large overhead, so, after experimentation, we simply replace the initial frame with a randomized frame. This greatly speeds up performance for many examples, and we report the performance for this method (which we call CC) as well as robustness tests in the results §4.1.2. See Algorithm 4 for pseudocode.

ALGORITHM 4: ComputeLink: Counting Crossings

Input : $\{\mathbf{l}_j\}$, $\{\mathbf{k}_i\}$, the two polyline loops, each represented as a list of vertices. Each vertex is a column vector.
Output: λ, the linking number

// This function computes the linking number between two closed polylines by counting crossings and uses a BVH.
Function `ComputeLink`($\{\mathbf{l}_j\}$, $\{\mathbf{k}_i\}$)**:**
  $\lambda \leftarrow 0$;
  $N_k \leftarrow |\{\mathbf{k}_i\}|$;
  // Find a regular projection frame.
  $F \leftarrow (\hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{z}}) \leftarrow$ `GenerateRegularProjectionFrame`($\{\mathbf{l}_j\}$, $\{\mathbf{k}_i\}$);
  // Rotate into the frame.
  $\{\mathbf{l}_j\} \leftarrow \{F^T \mathbf{l}_j\}$; $\{\mathbf{k}_i\} \leftarrow \{F^T \mathbf{k}_i\}$;
  // Build a tree from the projected segments of $\{\mathbf{l}_j\}$.
  tree $\leftarrow$ `BuildBVH`($\{((l_{jx}, l_{jy}), (l_{(j+1)x}, l_{(j+1)y}))\}$);
  // Sum the orientations of all crossings
  parallel for $i \leftarrow 0$ **to** N_k **do**
    ProjectedSegment $\leftarrow ((k_{ix}, k_{iy}), (k_{(i+1)x}, k_{(i+1)y}))$;
    foreach $j \in$ tree.Query(ProjectedSegment) **do**
      SegmentI, SegmentJ $\leftarrow (\mathbf{k}_i, \mathbf{k}_{i+1}), (\mathbf{l}_j, \mathbf{l}_{j+1})$;
      $\lambda \leftarrow \lambda +$ 0.5 Orientation(SegmentI, SegmentJ);
    end
  end
  return λ ;

3.4.2 Method 2: Direct Summation (DS). This approach simply uses the exact double-summation formula for Gauss's integral from [Arai 2013], produced above as (4). That is, for loops γ_1, γ_2, consisting of N_l, N_k line segments respectively, with j, i enumerating the segments respectively, compute

$$\lambda(\gamma_1, \gamma_2) = \sum_i^{N_k} \sum_j^{N_l} \lambda_{ji}. \tag{5}$$

See Algorithm 5 for a simple implementation. Unfortunately this approach computes $2N_l N_k$ arctangents for loops of sizes N_l and N_k, which is expensive for large loops.

When this is a single-threaded computation, the approach from [Bertolazzi et al. 2019] removes all arctangents by using angle summations (counting negative x-axis crossings) instead. We use a modified version of their approach, and for robustness, we add a single arctan at the end to compute the remainder after the angle summation. See Appendix B.1 for details on how we multithread this on the CPU and the GPU.

ALGORITHM 5: ComputeLink: Direct Summation.

Input : $\{\mathbf{l}_j\}$, $\{\mathbf{k}_i\}$, the two polyline loops, each represented as a list of vertices. Each vertex is a column vector.
Output: λ, the linking number

// This function computes the linking number between two closed polylines using the [Arai 2013] expression.
Function `ComputeLink`($\{\mathbf{l}_j\}$, $\{\mathbf{k}_i\}$)**:**
  $\lambda \leftarrow 0$;
  $N_l \leftarrow |\{\mathbf{l}_j\}|$; $N_k \leftarrow |\{\mathbf{k}_i\}|$;
  foreach int $(i, j) \in [0, N_k - 1] \times [0, N_l - 1]$ **do**
    $\mathbf{a} \leftarrow \mathbf{l}_j - \mathbf{k}_i$;
    $\mathbf{b} \leftarrow \mathbf{l}_j - \mathbf{k}_{i+1}$;
    $\mathbf{c} \leftarrow \mathbf{l}_{j+1} - \mathbf{k}_{i+1}$;
    $\mathbf{d} \leftarrow \mathbf{l}_{j+1} - \mathbf{k}_i$;
    $p \leftarrow \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$;
    $d_1 \leftarrow |\mathbf{a}||\mathbf{b}||\mathbf{c}| + \mathbf{a} \cdot \mathbf{b}|\mathbf{c}| + \mathbf{b} \cdot \mathbf{c}|\mathbf{a}| + \mathbf{c} \cdot \mathbf{a}|\mathbf{b}|$;
    $d_2 \leftarrow |\mathbf{a}||\mathbf{d}||\mathbf{c}| + \mathbf{a} \cdot \mathbf{d}|\mathbf{c}| + \mathbf{d} \cdot \mathbf{c}|\mathbf{a}| + \mathbf{c} \cdot \mathbf{a}|\mathbf{d}|$;
    $\lambda \leftarrow \lambda + (\text{atan2}(p, d_1) + \text{atan2}(p, d_2))/(2\pi)$;
  end
  return λ;

3.4.3 Method 3: Fast Multipole Method (FMM). We can also use external FMM libraries (provided by FMMTL [Cecka and Layton 2015] or FMM3D [Cheng et al. 1999]) to compute the linking integral using fast summation approximations. In particular, if we represent γ_2 as a source current, the Fast Multipole Method on the Biot-Savart integral will approximately evaluate, at each target point $\mathbf{r}$, the field

$$\mathbf{f}(\mathbf{r}) = \frac{1}{4\pi} \int_{\gamma_2} \frac{d\mathbf{r}_2 \times (\mathbf{r} - \mathbf{r}_2)}{|\mathbf{r} - \mathbf{r}_2|^3}. \tag{6}$$

The final linking number is then $\lambda(\gamma_1, \gamma_2) = \int_{\gamma_1} \mathbf{f}(\mathbf{r}_1) \cdot d\mathbf{r}_1$. In this notation, loop γ_2 is the "source", and γ_1 is the "target".

Because many of these libraries do not directly support finite line-segment inputs, we use segment midpoints for the source and target points. After evaluation by the library, we compute a set of finite-segment corrections for close segment pairs poorly approximated by midpoint samples, and add the correction. While we already know

the exact evaluation expressions for line-segment pairs and could have modified libraries to use them, our goal is to make no modifications to existing FMM libraries that operate on point samples, so that our results can be easily reproduced and improvements to FMM libraries on newer hardware can be easily incorporated. This is why we perform post-evaluation correction rather than directly modify the FMM kernels and tree building structures.

Given the aforementioned loop discretizations of γ_1, γ_2 in the direct evaluation section, denote the midpoint of each line segment as $\mathbf{r}_j$ or $\mathbf{r}_i$, respectively, and the displacement vector (difference of endpoints) as $\mathbf{s}_j$, $\mathbf{s}_i$, respectively. FMMTL directly provides a Biot-Savart implementation, so we simply have to pass in the line segment displacements $\mathbf{s}_i$ as the sources, with positions $\mathbf{r}_i$, and FMMTL computes the fields $\mathbf{f}_j = \mathbf{f}(\mathbf{r}_j)$ at positions $\mathbf{r}_j$.

The linking number can then be computed from the field $\mathbf{f}_j$ using

$$\lambda(\gamma_1, \gamma_2) = \sum_j \mathbf{s}_j \cdot \mathbf{f}_j. \tag{7}$$

To reduce the duplicate work of building the FMM trees, we batch evaluation points for each loop. That is, given source loop γ_n, we concatenate the target evaluation points r_j for all target loops to compute F_j for all target loops, and then compute each linking-number sum individually.

When a library does not provide a native Biot-Savart implementation, we can alternately pass in a list of three-vector sources $\{\mathbf{s}_i\}$ into a Laplace FMM. The gradient output gives us (6), except with an outer instead of a cross product. The output is a 3×3 tensor $C_j = \sum_i \mathbf{s}_i \otimes \nabla G(\mathbf{r}_j, \mathbf{r}_i)$ at each evaluation point $\mathbf{r}_j$, and we can simply grab the field f_j from the definition of cross product:

$$\mathbf{f}_j = (C_{j23} - C_{j32}, C_{j31} - C_{j13}, C_{j12} - C_{j21}). \tag{8}$$

After the FMM library computation, we add a finite-segment correction, because we passed in our line segments as point sources and targets. We pick a distance ratio β and apply this correction to all segment pairs that are less than $\beta(l_1 + l_2)/2$ apart, where l_1, l_2 are their segment lengths. Specifically, before the FMM calls we (I) build a bounding volume for each segment, and (II) dilate each bounding volume by $\beta l/2$, where l is the segment length. We then (III) build a BVH of segment bounding volumes for each loop. For any loop pair, (IV) traverse their BVH trees, and (V) accumulate the difference between expression (4) and its approximation (2) for every pair of overlapping segment bounding volumes. We then (VI) add this correction to the original FMM result. In our implementation, we parallelize steps (I–III) by loop and steps (IV–V) by loop pair, with a reduction at the end. We used a finite-segment correction because the correction is faster to compute in practice than using denser sample points along each segment for the FMM input; in our results in §4, the correction takes a small fraction (<1/8) of the runtime.

3.4.4 Method 4: Barnes-Hut (BH). Similar to [Barill et al. 2018], simple Barnes-Hut trees [Barnes and Hut 1986] can be sufficient for approximately evaluating the integral using fast summation without the overhead of the full FMM. For each loop pair, the FMM approach computes a field at every "target" point; however, we only need a single scalar. Since each loop can participate in many loop pairs, we can instead precompute a tree for each loop. Then for each loop pair, we traverse both trees from the top down using Barnes-Hut; this can save effort especially when the loops do not hug each other closely and multiple loop pairs use the same target loop.

The Barnes-Hut algorithm takes advantage of a far-field multipole expansion. Given two curves γ_1, γ_2, parameterize γ_1 as $\mathbf{r}_1 = \mathbf{r}_1(s)$ and γ_2 as $\mathbf{r}_2 = \mathbf{r}_2(t)$, and denote $d\mathbf{r}_1/ds$ and $d\mathbf{r}_2/dt$ by $\mathbf{r}_1'$ and $\mathbf{r}_2'$. Suppose also that we are only integrating the portions of γ_1 near $\tilde{\mathbf{r}}_1$, and the portions of γ_2 near $\tilde{\mathbf{r}}_2$. Also, let $\boldsymbol{\nabla}$ denote differentiation with respect to $\mathbf{r}_2$ (note that $\boldsymbol{\nabla}_{\mathbf{r}_1} G$ is just $-\boldsymbol{\nabla}_{\mathbf{r}_2} G$), $\otimes$ denote a tensor (outer) product, and $\cdot$ denote an inner product over all dimensions. Taylor expanding (1) with respect to $\mathbf{r}_1$ and $\mathbf{r}_2$, we have

$$\lambda = -\int \mathrm{d}s\, \mathrm{d}t\ (\mathbf{r}_1' \times \mathbf{r}_2') \cdot \boldsymbol{\nabla} G(\mathbf{r}_1, \mathbf{r}_2). \tag{9}$$

$$\begin{aligned}\lambda = -\int \mathrm{d}s\, \mathrm{d}t\ \Big[&(\mathbf{r}_1' \times \mathbf{r}_2') \cdot \boldsymbol{\nabla} G(\tilde{\mathbf{r}}_1, \tilde{\mathbf{r}}_2) \\ &+ ((\mathbf{r}_1' \times \mathbf{r}_2') \otimes (-(\mathbf{r}_1 - \tilde{\mathbf{r}}_1) + (\mathbf{r}_2 - \tilde{\mathbf{r}}_2))) \cdot \boldsymbol{\nabla}^2 G(\tilde{\mathbf{r}}_1, \tilde{\mathbf{r}}_2) \\ &+ \frac{1}{2}((\mathbf{r}_1' \times \mathbf{r}_2') \otimes (-(\mathbf{r}_1 - \tilde{\mathbf{r}}_1) + (\mathbf{r}_2 - \tilde{\mathbf{r}}_2)) \otimes (-(\mathbf{r}_1 - \tilde{\mathbf{r}}_1) + (\mathbf{r}_2 - \tilde{\mathbf{r}}_2))) \\ &\quad \cdot \boldsymbol{\nabla}^3 G(\tilde{\mathbf{r}}_1, \tilde{\mathbf{r}}_2) + O(|\tilde{\mathbf{r}}_1 - \tilde{\mathbf{r}}_2|^{-5})\Big].\end{aligned} \tag{10}$$

Expanding (10) and applying some algebra, every term can be split into a product of two separate integrals, one for the $\mathbf{r}_1$ factors and one for the $\mathbf{r}_2$ factors. These integrals, known as moments, can be precomputed when constructing the tree. For each node on both curves, we precompute the following moments:

$$\mathbf{c}_M = \int \mathbf{r}'\, \mathrm{d}s, \tag{11}$$

$$C_D = \int \mathbf{r}'(\mathbf{r} - \tilde{\mathbf{r}})^T\, \mathrm{d}s, \tag{12}$$

$$C_Q = \int \mathbf{r}' \otimes (\mathbf{r} - \tilde{\mathbf{r}}) \otimes (\mathbf{r} - \tilde{\mathbf{r}})\, \mathrm{d}s. \tag{13}$$

Having just the $\mathbf{c}_M$ moment at each node is sufficient to give us the first term in (10). Adding the C_D and C_Q moments gives us the second and third terms respectively. The derivatives of $G(\tilde{\mathbf{r}}_1, \tilde{\mathbf{r}}_2)$, how we compute these moments, and how we parallelize the Barnes-Hut on the CPU and GPU (using the Barnes-Hut NBody implementation from [Burtscher and Pingali 2011]) are in Appendix B.2.

We set a β tolerance parameter, similar to most Barnes-Hut algorithms, to distinguish near-field and far-field: two nodes at locations $\mathbf{p}$ and $\mathbf{q}$ are in the far field of each other if $|\mathbf{p} - \mathbf{q}|$ is greater than β times the sum of the bounding-box radii of the two nodes. Starting from both tree roots, we compare their bounding boxes against each other. If the nodes are in the far field, we use the precomputed moments and evaluate the far-field expansion; otherwise, we traverse down the tree of the larger node and check its children. If both nodes are leaves, we use the direct arctangent expression (4). See Algorithm 6 for a summary.

In our implementation, we first run Barnes-Hut with the second-order expansion in (10) (using all three terms) with $\beta_{\text{init}} = 2$. Unlike other problems that use Barnes-Hut, the linking number certificate requires the same absolute error tolerance, regardless of the magnitude of the result. In large examples, some regions can induce a strong magnetic field from one loop on many dense curves from the other loop, resulting in a very large integrand. Our implementation

ALGORITHM 6: ComputeLink: Barnes-Hut

Input : tree1, tree2, the root nodes of the two trees, with $\mathbf{r}$ the node's center, R the bounding box radius, and $\mathbf{c}_M, C_D, C_Q$ the moments, precomputed at each node.
Output : λ, the linking number

```
// This function computes the linking number between two closed
   polylines using the Barnes-Hut algorithm with a quadrupole
   far-field expansion.
Function ComputeLink(tree1, tree2):
    if (|tree1.r−tree2.r| > β(tree1.R+tree2.R)) then
        return FarFieldEvaluation(tree1, tree2); // Use (10)
    else if (tree1 and tree2 have no children) then
        // Use (4)
        return DirectEvaluation(tree1.Segment, tree2.Segment);
    else if (tree1.R > tree2.R and tree1 has children) then
        return ∑_c ComputeLink (tree1.child[c], tree2);
    else
        return ∑_c ComputeLink (tree1, tree2.child[c]);
    end
```

therefore uses an error estimate based on the next-order expansion term to set a new β for a second evaluation if necessary. The estimate is accumulated at each far-field node pair and reported after the evaluation. The new β_t is given by

$$E_{\text{estimate}} = k \sum_{(i,j)\in\text{far-field node pairs}} |\mathbf{r}|^{-5}(R_i|\mathbf{c}_{jM}|\|C_{iQ}\| + R_j|\mathbf{c}_{iM}|\|C_{jQ}\| + 3(\|C_{iD}\|\|C_{jQ}\| + \|C_{iQ}\|\|C_{jD}\|)), \tag{14}$$

$$\beta_t = \left(\frac{E_{\text{estimate}}}{E_{\text{target}}}\right)^{1/4} \beta_{\text{init}}. \tag{15}$$

Here $\mathbf{r} = \tilde{\mathbf{r}}_j - \tilde{\mathbf{r}}_i$, R is the bounding-box radius at that node, and k is a constant. If $\beta_t > \beta_{\text{init}}$, then it reruns Barnes-Hut using the second-order expansion and $\beta = \min(\beta_t, \beta_{\max})$.

4 RESULTS AND ANALYSIS

We now compare the many aforementioned methods on several benchmarks and applications. Please see the supplemental video for all animation results.

4.1 Experiments and Verification

We perform all our experiments on a single 18-core Intel® Xeon® CPU E5-2697 v4 @ 2.30GHz with 64GB of RAM, and the GPU versions are evaluated on an NVIDIA GeForce GTX 1080. In Table 1, we evaluate the methods on inputs with varying numbers of loops, loop sizes, and loop shapes, and report runtimes and accuracies of the linkage values. For CPU Barnes-Hut, we used $\beta_{\text{init}} = 2$, $\beta_{\max} = 10$, and $E_{\text{target}} = 0.2$. We chose β_{init} so that the absolute errors were well below 10^{-2} for most examples. For Barnes-Hut on the GPU, since we only evaluated up to dipole moments with no error estimation, we used $\beta = 4$ if $N_1 N_2 < 10^8$ and $\beta = 16$ otherwise. We also included the runtime and accuracies achieved using direct summation [Bertolazzi et al. 2019], the FMMTL library [Cecka and Layton 2015; Cheng et al. 1999], and the modified count-crossings method from [Dey et al. 2013] accelerated with parallelism and a BVH. The number of discretized line segments for each input is listed in Table 2. In most cases, Discretization did not significantly increase the segment count (The exception is the chevron 3×3, which uses a special procedure defined in §4.3.5 where each curve participates in two virtual loops).

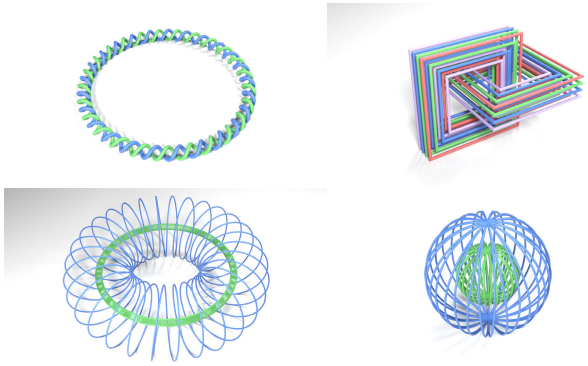


Fig. 6. **Large-input Stress Tests:** Synthethic examples were generated for (Top Left) DNA-like double helix ribbons, (Top Right) thick square links, (Bottom Left) tori, and (Bottom Right) woundballs, of variable sizes. The "thick square links" example has $L^2/4$ links with $\lambda = 1$, and the torus has a single λ equal to the product of the number of toroidal periods in the green loop and poloidal periods in the blue loop. We also generated a $\lambda = 0$ torus where both loops simply alternate directions every period, and it also has 2 more larger copies of the blue loop. The woundball has two similar concentric spherical curves, one winding a ball half the size of the other, and a parameter ν indicating the number of times each curve crosses the North pole.

In general, the Counting Crossings (CC) and the two Barnes-Hut (BH) implementations give the fastest runtimes, with the CC, BH, and BHG all taking under 2 seconds for all examples shown in the video (first 11 rows of Table 1). As a result, we believe either a CC or a BH method is performant enough for most applications.

The GPU implementations perform best when they have very few, or one, pairs of large loops to evaluate, and this can be seen in the ribbon, $\lambda = 10^6$ torus, and low-frequency woundball. In particular, the smaller double helix ribbons (N_I=200000) examples illustrate that the GPU Direct Summation (DS) is a 370× speedup over the CPU DS, and both DS implementations have very little error (mostly under 10^{-5} for the single-precision GPU implementation). The Barnes-Hut CPU implementation benefits from traversing two trees, a quadrupole expansion, as well as error estimation, allowing it to perform almost as well as the BH GPU on a wide range of examples that involve more than one loop pair. However, both Barnes-Hut implementations suffer when the segment lengths are relatively long (compared to the distance between curves), which is true in the higher-ν woundball (a higher ν has longer segments in order to traverse the ball more times), because their limited far-field expansion requires a relatively high β. In contrast, the FMM excels on the high-ν woundball, giving the fastest performance, because it uses arbitrary-order expansion terms as needed. Both BH and FMM suffer from hard-to-predict absolute errors, especially in examples with high λ. In general, CC appears to perform the best in the most

Table 1. **Linkage Computation for Various Inputs:** In this table, N_I is the total number of input curve segments, L is the number of input loops, and P is the computed number of potentially linked loop pairs. The first 11 models correspond to animated results shown in the supplemental video, and the rest are stress tests depicted in Fig. 6. The sweater and glove models originate from [Yuksel et al. 2012], with "Sweater" referring to "sweater flame ribbing". "PLS Time" is the Potential Link Search Time and "Dscr Time" is the discretization time; DS is direct summation; CC_{ann} and CC for counting crossings with ANN and randomized direction initialization, respectively; BH for Barnes-Hut, and FMM for Fast Multipole Method (using FMMTL with batched target points; FMM3D is slower in almost all cases). DSG and BHG run the DS and BH, respectively, on the GPU. A "N/A" indicates the run took longer than a half hour or did not fit in memory. λ indicates the linkage between curves in a model, and ν is a woundball parameter explained in Fig. 6.
* For this example, FMMTL failed, and this is the FMM3D runtime instead.
† These used $\beta = 40$, instead of $\beta = 16$ normally used for large input. (The GPU BH only evaluates up to dipole moments, requiring high β for large input.)

Linkage Computation for Various Input, Using Various Methods (All Times in [s])

Model	N_I	L	P	PLS Time	Dscr. Time	Linking Number Matrix Compute Times							Abs. Error		
						DS	CC_{ann}	CC	BH	FMM	DSG	BHG	BH	BHG	FMM
Alien Sweater (Initial)	139506	146	1335	0.013	0.059	11.1	12.6	0.34	0.32	1.09	0.63	0.27	3E-3	2E-3	1E-3
Alien Sweater (Final)	139506	146	2426	0.017	0.050	13.9	22.4	0.38	1.18	1.61	1.01	0.63	3E-3	8E-4	1E-3
Sheep Sweater	729628	549	4951	0.024	0.152	169	95.5	0.85	1.31	5.85	4.34	0.96	2E-3	5E-3	3E-3
Sweater	271902	253	2985	0.015	0.120	29.3	33.0	0.29	0.74	2.33	1.43	0.56	3E-3	2E-3	9E-4
Glove	58537	70	1022	0.012	0.046	2.71	8.49	0.10	0.14	0.56	0.35	0.16	2E-3	2E-3	4E-4
Knit Tube (Initial)	18228	39	233	0.012	0.032	0.21	1.01	0.07	0.05	0.09	0.02	0.08	4E-3	9E-4	1E-3
Knit Tube (Final)	18228	39	180	0.012	0.082	0.30	1.03	0.06	0.08	0.11	0.30	0.12	6E-3	3E-4	3E-3
Chainmail (Initial)	211680	14112	18781	0.028	0.070	0.04	1.85	0.03	0.09	0.10	1.20	N/A	1E-3	N/A	5E-4
Chainmail (Final)	211680	14112	61417	0.056	0.086	0.11	6.30	0.07	0.15	0.17	3.91	N/A	3E-4	N/A	7E-4
Chevron 3×3	11741	32	58	0.013	0.049	0.19	0.44	0.02	0.07	0.06	0.02	0.03	5E-3	2E-4	1E-3
Rubber Bands	51200	1024	20520	0.018	0.035	0.23	7.99	0.05	0.12	0.16	1.11	1.87	7E-4	2E-5	5E-4
Double Helix Ribbon $\lambda = 10$	200000	2	1	0.018	0.076	177	5.60	0.06	0.15	0.34	0.48	0.02	7E-3	1E-3	1E-4
Double Helix Ribbon $\lambda = 10^3$	200000	2	1	0.016	0.112	177	1.82	0.07	1.62	0.55	0.48	0.03	0.22	7E-2	0.37
Double Helix Ribbon $\lambda = 10$	2E7	2	1	0.465	6.816	N/A	131	8.62	11.8	N/A	N/A	2.63†	3E-2	5E-4	N/A
Double Helix Ribbon $\lambda = 10^3$	2E7	2	1	0.441	6.816	N/A	118	9.51	31.3	N/A	N/A	5.30†	0.17	0.15	N/A
Thick Square Link	500000	1250	4.1E5	0.054	0.146	161	N/A	31.1	6.83	37.6	26.4	23.2	1E-3	6E-4	1E-4
Thick Square Link	4E6	5000	6.4E6	0.531	1.874	N/A	N/A	105	145	830	1844	N/A	2E-3	N/A	2E-4
Torus $\lambda = 10^6$	2E7	2	1	0.456	7.027	N/A	100.3	9.30	13.9	22.2	N/A	4.94	5E-3	6E+2	9E-4
Torus $\lambda = 0$	4E7	4	6	0.519	7.698	N/A	N/A	55.1	18.3	N/A	N/A	N/A	3E-3	N/A	N/A
Woundball $\nu = 10^3$	1E6	2	1	0.048	0.387	N/A	6.93	0.55	0.68	1.32	12.0	0.14	6E-2	1E-2	2E-5
Woundball $\nu = 10^4$	2E6	2	1	0.056	0.756	N/A	43.9	7.96	15.3	5.19*	47.7	90.0	1E-2	5E-4	7E-6

Table 2. **Discretization Results:** This table shows, for the first 10 models in Table 1, how many line segments each model was discretized into. All other models were input as line segments separated enough that discretization did not change the segment count. Most of these 10 models ended with only a little more than one segment per control point, with the exception of the twisted knit tube and the chevron pattern.

Number of Discretized Segments for Various Inputs

Model	N_I	N_D
Alien Sweater (Initial)	139506	158708
Alien Sweater (Final)	139506	143228
Sheep Sweater	729628	801447
Sweater	271902	296813
Glove	58537	65938
Knit Tube (Initial)	18228	18941
Knit Tube (Final)	18228	27625
Chainmail (Initial)	211680	211680
Chainmail (Final)	211680	212552
Chevron 3×3	11741	32804

scenarios, and because it tabulates an integer, it has 0 error in the runs we observed.

4.1.1 Barnes-Hut β behavior: We illustrate how varying β can impact accuracy in Figure 7. The error appears to behave as β^{-3} for the first-order (dipole) expansion, whereas for the second-order (quadrupole) expansion the error appears to behave as β^{-4}.

4.1.2 Numerical conditioning and robustness: For direct summation (and lowest-level evaluations of the tree algorithms), we use the expression from Arai [2013] (reproduced in §2.1 as (4)), because it is well defined over the widest range of input. In particular, it is only singular when the two line segments intersect or collinearly overlap, or if either has zero length. Our method already detects and flags these conditions in the discretization step. Furthermore, the signed solid angle formula is widely used due to its simplicity and stability [Van Oosterom and Strackee 1983].

All methods compute using double precision when implemented on the CPU, while the GPU implementations use single precision.

To learn more about the robustness of the projection in the count-crossings (CC) method, we tested it, using random initial frames

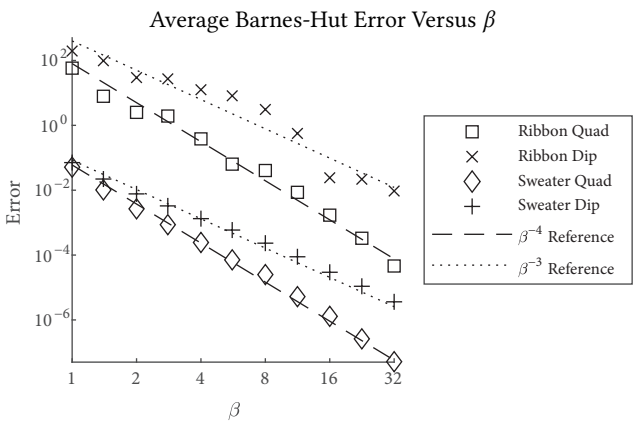


Fig. 7. **Barnes-Hut Accuracy Versus β:** We plot the (L1) average error versus the Barnes-Hut β parameter for the double-helix DNA ribbon and the alien sweater examples. We also show this with only dipole ("Dip", first-order) terms versus having the quadrupole ("Quad", second-order) terms as well. The error appears to drop off as β^{-3} using the dipole expansion and as β^{-4} for the quadrupole expansion.

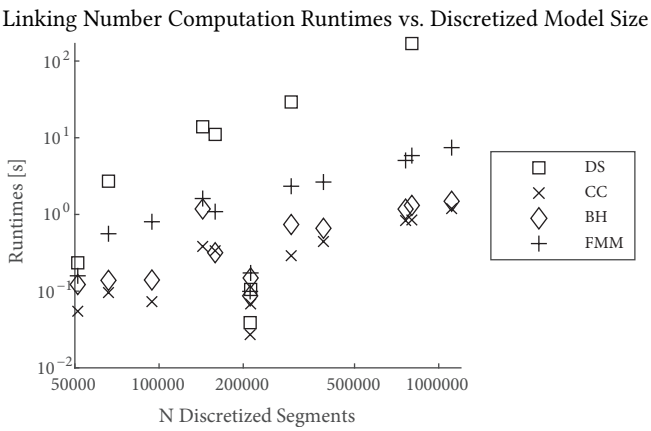


Fig. 8. **Runtimes for Linking Number Computation:** This plots the runtime in ms against the discretized model size, for the examples shown in the video (first 11 rows of Table 1 plus a few reparameterized models). Because the models vary in shape complexity, number of loops, and number of segments per loop, the input size cannot be used to perfectly predict the runtime.

before the rotation step, 50,000 times on the compressed chainmail (final), and 5,000 times on the alien sweater (initial), which based on their P values in Table 1, summing up to 3.08 billion projections. If our implementation encounters a degeneracy, the method would fail and exit. For each loop pair, we recomputed the linking number with a new projection every trial, and it completed with the correct result every time. We also tested it with very large inputs such as the dense tori and woundballs (Fig. 6), which span a large range of angles, and the linking number was successfully computed, albeit the crossings computation was very slow for the unlinked $\lambda = 0$ torus. This is likely because degeneracies are unlikely to appear with this input size in double precision. That said, we do not guarantee with our implementation that it is able to find a non-degenerate, regular projection on every input. If this is a concern, the Gauss summation implementations such as the Barnes-Hut algorithm, which do not require a projection, can also be used on most inputs with comparable speed.

4.2 Performance

For all results shown in the video (first 11 rows of Table 1), PLS and discretization took under 200 ms combined. For the third stage, we plot the runtimes versus discretized model size in Figure 8. Both the crossings-counting method and Barnes-Hut significantly outperform direct summation, and also outperform FMM in most cases. We also tested four classes of large input as stress tests; see Fig. 6 for an illustration.

4.2.1 Usual-case theoretical estimates: Similar to many tree-based methods, the performance depends on the distribution of input, and certain inputs can severely degrade performance. For inputs with relatively "short" segments (i.e. segments are short compared to the spacing between curves) and uniformly sized loops, we expect PLS to take $O(L \log L)$ time and Discretization to take $O(N \log N)$ time, where L is the number of loops and N is the number of input segments. The Barnes-Hut, Count-Crossings, and FMM computations are all expected to take $O(PN_L \log N_L)$ time, where P is the number of potentially linked pairs and N_L approximates the number of segments per loop (FMM is not linear in N_L because we compute the finite-segment correction, although for all the inputs we reported in Table 1, the finite-segment correction took under 1/8 of the FMM runtime). For chainmail, the total simplifies to $O(C \log C)$ where C is the number of rings, and for most stitch-mesh knit patterns where each row interacts with a small number of other rows, this simplifies to $O(R \log R + N \log N_L)$ where R is the number of rows.

However, if segments are extremely long compared to the distance between curves, which can happen after a simulation has destablized, every algorithm can slow down to $O(N^2)$ performance to finish. This is where the early exit approach can be a huge gain; furthermore, if our method runs periodically during a simulation, topology violations can be detected long before this occurs, as we will show in the Knit Tube Simulation (Figure 13).

4.3 Applications

In this section we demonstrate the versatility of our method across several applications. In most scenarios (except Example 4.3.1) we first compute the linking matrix of the initial model, and then compare it against the linking matrices of the deformed models.

4.3.1 Yarn model analysis: Many yarn-level models in graphics and animation are represented as a set of closed curves; a major example is the models generated from StitchMesh [Yuksel et al. 2012]. Implicit (and/or implicit constraint direction) yarn simulations solve smaller systems when the model uses closed curves, and the model stays intact without the requirement of pins. Our method can aid in the topology validation of these models, by computing a linking matrix certificate before and after a deformation, whether the deformation is a simulation, relaxation, reparameterization, or compression. In

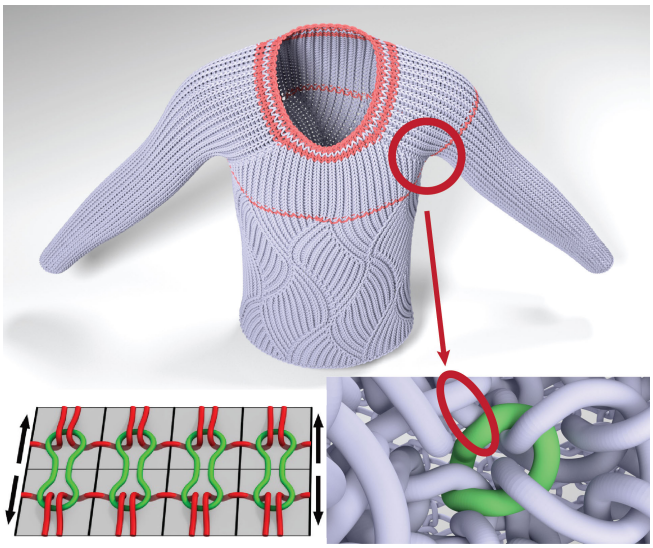


Fig. 9. **Yarn Model Analysis**: We examine the "sweater flame ribbing" yarn model from Stitch Mesh [Wu and Yuksel 2017; Yuksel et al. 2012], which was provided in relaxed form. This model contains small yarn loops to join rows with mismatched wale direction (Bottom Left, reproduced from Figure 8 in [Yuksel et al. 2012]). Even with these loops, the entire stitch mesh should have zero pairwise linkage between its yarn rows. We verify the relaxed yarn model using our algorithm, and mark linkages in red (Top). From this, we zoom in on one of the linkages, and find that the small green loop (Bottom Right), which should topologically match the small green loops in the bottom left figure, has only 3, rather than 4, curve segments passing through it, indicating there was yarn pull-through during relaxation.

addition, we can examine the integrity of input models from prior work. In our examples we analyze four models from [Wu and Yuksel 2017; Yuksel et al. 2012][2]: the "alien sweater", "sweater flame ribbing", "glove", and the "sheep sweater", which are provided after a first-order implicit yarn-level relaxation. Stitches between rows introduce zero pairwise linkage, and our method verifies whether the relaxed models have valid stitches. We found that the glove and the alien sweater have no pairwise linkages, indicating no obvious issues, while the sheep sweater and "sweater flame ribbing" contain linkage violations. See Figure 9 for a violation in the "sweater flame ribbing" model.

4.3.2 Reparameterizing and compressing yarn-level models: Our methods can be used to verify the topology of yarn-level cloth models following common processing operations. Spline reparameterization can be used to reduce the number of spline control points prior to simulation, or afterwards for model export. Furthermore, it is common to compress control point coordinates using quantization for storage and transmission. Unfortunately such operations can introduce topological errors, which limits the amount of reduction that can occur. Our method can be used to verify these reparameterization and compression processes. Since large modifications of the model may introduce multiple linking errors that could cancel each other out, we leverage our methods' speed to analyze a sweep of the reparameterization/compression process to better detect errors. See Figure 10 for results.

[2]Models downloaded from http://www.cemyuksel.com/research/yarnmodels

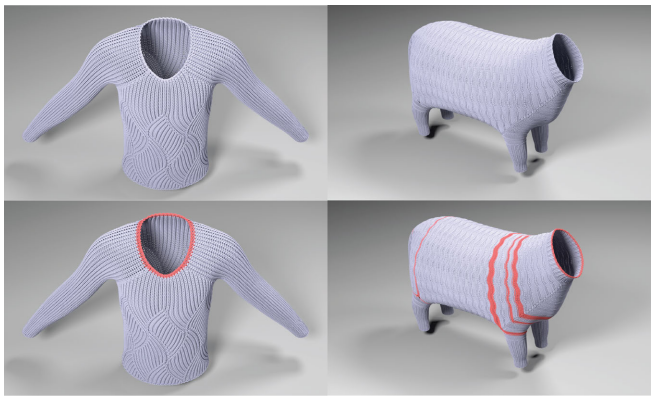


Fig. 10. **Compression and Reparameterization of Detailed Yarn Models:** (Left) We compress a sweater model down to 16-bit (Top Left) and 12-bit (Bottom Left) precision by quantizing coordinates. Our algorithm verifies that the 16-bit compression is valid, but highlights the loops that contain topology violations in the 12-bit compression. (Right) If we aggressively resample splines of the sheep sweater model to reduce the number of control points from 729628 (Top Right) to 547264 (Bottom Right), our method highlights violations in the latter.

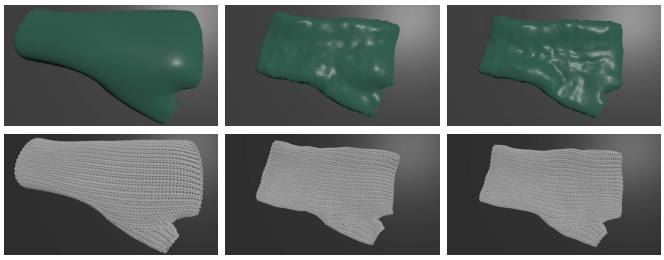


Fig. 11. **Embedded Deformation of Yarn-Level Models:** A coarse single-layer tetrahedral mesh (simulated by Houdini's Vellum solver) is used to place a yarn-level glove model on a surface using embedded deformation. We verify that no topology changes are introduced by this deformation.

4.3.3 Embedded deformation of yarn-level models: It is computationally appealing to animate and pose yarn-level models using cheaper deformers than expensive yarn-level physics, but also retain yarn topology for visual and physical merits. To this end, we animated the yarn-level glove model using embedded deformation based on a tetrahedral mesh simulated using Houdini 18.5 Vellum. For the modest deformations shown in Figure 11 the yarn-level model retains the correct topology throughout the animation, and therefore can be used for subsequent yarn-level processing.

4.3.4 Implicit yarn-level simulation: Simulation of yarn, as in [Kaldor et al. 2008], is challenging because of the large number of thin and stiff yarns in contact, and the strict requirement to preserve topology (by avoiding yarn-yarn "pull through"), both of which can necessitate small time steps. For large models, these relaxations

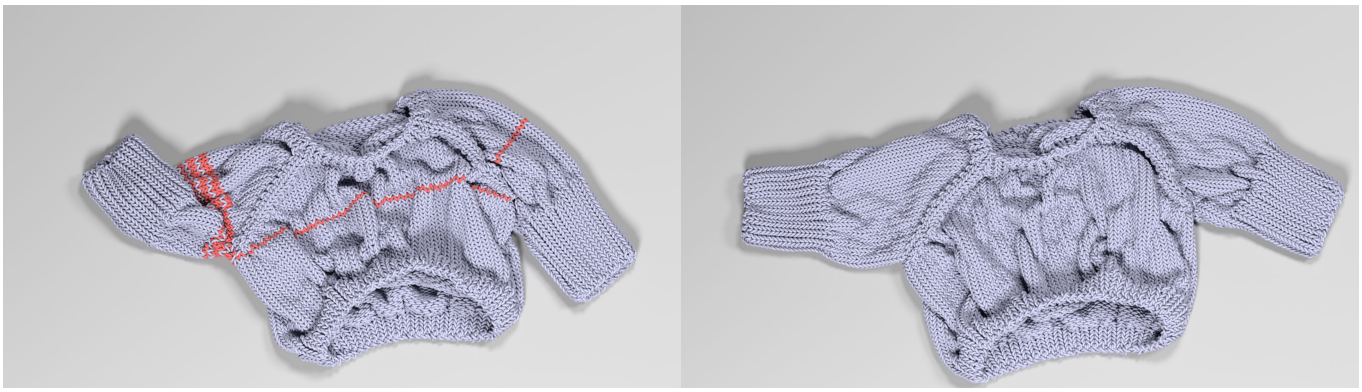


Fig. 12. **Verification of Sweater Drop Simulation:** Our method detected that (Left) a preview implicit simulation with larger step sizes failed to preserve yarn topology, while (Right) another simulation with smaller steps maintained it.

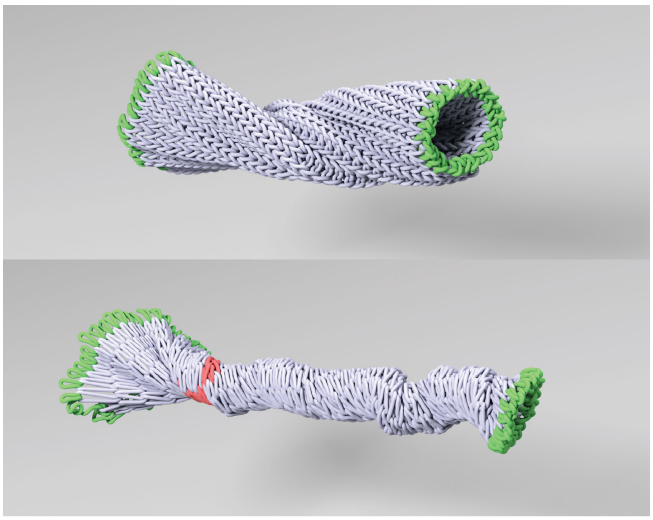


Fig. 13. **Verification of Tube Twist Simulation:** Twisting cloth is a common stress test for a simulator, and in this image, the end rows (green) are twisted using a stiff spring until breakage. Our method is able to cheaply detect topology violations in the model (red) over a thousand steps before the simulation visually explodes.

or simulations can take hours or days, and there may not be an expert user in the loop to visually monitor the simulations everywhere. While sometimes penalty-based contact forces will cause the simulation to blow up, alerting the user to restart, at other times pull-through failures can be silent and lead to incorrect results.

Our method is useful for validating yarn models during yarn-level relaxation, when large forces can cause topological changes, as well as large-step, preview implicit simulations of yarn-level models. We use implicit integration [Baraff and Witkin 1998; Bergou et al. 2010; Kaldor et al. 2008; Leaf et al. 2018] to step through the stiff forces, which enables larger timesteps at the risk of introducing topology errors; our PCG implementation uses Eigen [Guennebaud et al. 2010] and falls back to the SYM-ILDL package [Greif et al. 2015] when the energy gets too nonconvex [Kim 2020]. In Figure 12, we simulate the alien sweater model from [Yuksel et al. 2012] using larger implicit steps (1/2400 s) for a preview simulation as well as smaller implicit-explicit step sizes (1/6600 s). Our method detects and marks the yarn loops that have engaged in pull through violations in the former, and validates that the latter maintains its topology. In Figure 13, we perform a common stress test where we grab a knitted cylindrical tube by the ends and wring (twist) the ends in opposing directions until the simulation breaks. Even in this simulation with an obvious failure, our method is still useful because it detects the first linkage violations over a thousand steps before the simulation completes. In all of these simulations, the runtime of our method is small compared to the runtime of the simulation: we only run our method once every animation frame, whereas the simulation takes dozens of substeps per frame.

4.3.5 Open-curve verification: chevron 3×3 *stitch pattern relaxation:* Our method can be applied not only on closed loopy structures, but also on finite, open curves in specific circumstances, to detect topology violations between different curves. In particular, this notion is well defined when the curves form a braid, that is, when the curves connect fixed points on two rigid ends, and the rigid ends remain separated. This can be useful for verifying the relaxation of a yarn stitch pattern, if the top and bottom rows are either cast on or bound off, and the ends of each row are pinned to two rigid ends, turning the stitch pattern into a braid.

When the curves form a braid, we can connect them using virtual connections to form virtual closed loops (see Figure 14). We propose a method to automatically form virtual connections which we include in Appendix C. To test this, we use a stitch pattern that repeats a chevron three times horizontally and vertically (3×3). We apply our proposed method to the chevron 3×3 to verify the relaxation of the stitch pattern (see Figure 15). In the video, we also detect an early pull through in another relaxation with larger steps, hundreds of steps before it becomes visually apparent.

4.3.6 Chainmail simulation: In Figure 1, we simulated a fitted Kusari 4-in-1 chainmail garment with 14112 curved rings and 18752 links. Inspired by the 3D printing example from [Tasora et al. 2016], we simulated packing the garment into a small box for 3D printing using

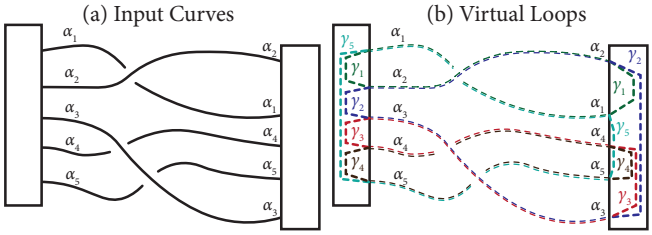


Fig. 14. **Verification of Open Curves Attached to Rigid Ends:** We can automatically validate curve topology for open curves with ends that are fixed with respect to each other. In this illustration, when we have five open curves, α_1 to α_5, we can form five virtual closed loops, γ_1 to γ_5 by attaching every two curves into a loop, with a virtual connection inside each end volume: $\gamma_i = \alpha_i \cup \alpha_{i+1}$ for $i = 1, 2, 3, 4$ and $\gamma_5 = \alpha_5 \cup \alpha_1$. As every curve participates in two loops, we modify PLS to exclude pairs of loops that share curves, and then generate the certificate. During deformation, as long as no curve crosses the virtual end connections and no virtual connection crosses each other (this is guaranteed by rigid ends), the certificate will detect pull throughs. See the chevron stitch pattern (Fig. 15), for example.

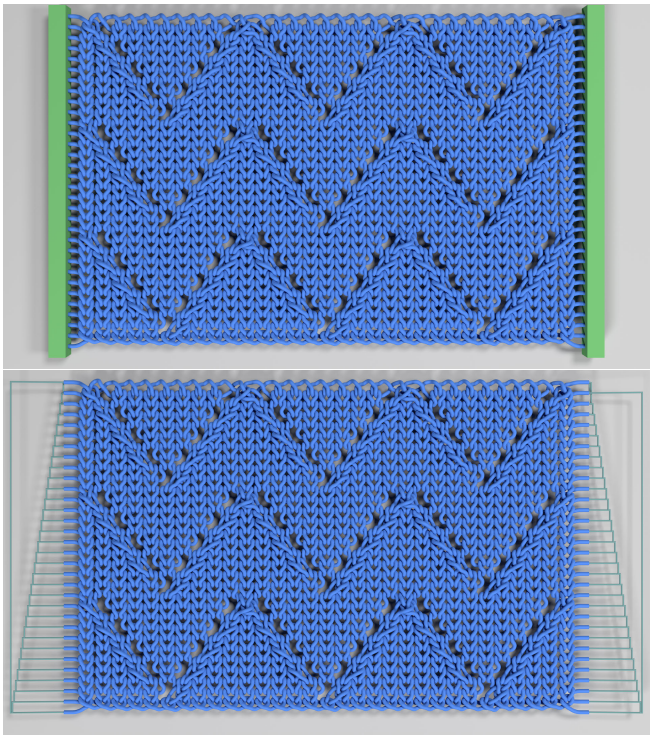


Fig. 15. **Verification of Chevron Stitch Pattern During Relaxation:** (Top) We attach a 3×3 chevron stitch pattern between two rigid bars and relax the pattern while slowly pulling the bars apart. Using our procedure to verify open curves attached to rigid ends, we verify that the relaxation preserves stitch topology. (Bottom) In this image, the virtual connections are shown, exaggerated outside their volumes, as green wires extending to the left and right. See the supplemental video for an alternate relaxation with an early pull through detected in the process. The violation there is detected in the first few steps, while the pull through only becomes visually apparent after hundreds of steps.

the Bullet rigid-body solver in Houdini 18.5. Without exceptionally small timesteps, the solver destroys links and creates spurious others as shown in Figure 1 and supplemental animations. By using varying numbers of substeps, we could illustrate linkage violations in Figure 1 (b) and (c), and certify that the result in (d) maintains its topological integrity, which would be otherwise difficult to ascertain until after expensive 3D printing.

4.3.7 Rubber band simulation: Errors in collision and contact handling can result in previously unlinked loops becoming entangled, and thus necessitate more accurate but expensive simulation parameters. We demonstrate this scenario by the simulation of 1024 unlinked rubber bands dropped onto the ground in Figure 16. Our methods can be used to monitor and detect topological errors introduced by low-quality simulation settings (too few substeps), and inform practitioners when to increase simulation fidelity.

4.3.8 Ribbons and DNA:. Some applications in computational biology require the computation of linking numbers (and related quantities such as writhe) for extremely long DNA strands, which can change their linking numbers as a result of topoisomerase, thereby producing interesting effects such as super-coiling [Clauvelin et al. 2012]. For computational purposes, the strands are essentially ribbons with two twisted double-helix curves along each edge. While existing approaches typically only consider direct evaluation methods for relatively small models, e.g., [Sierzega et al. 2020], we demonstrate performance on very long and closely separate closed ribbons (akin to circular DNA) depicted in the top left of Fig. 6. Results shown in Table 1 illustrate excellent performance by several of the algorithms; most notably, the GPU Barnes-Hut approach finishes 200,000-segment ribbons in under 30 milliseconds and, with sufficiently high β, finishes 20-million-segment ribbons in 5.3 seconds or faster; combined with the first two stages, our method takes up to 157 ms to compute the linking matrix for the former ribbon and 12.6 seconds for the larger ribbon.

5 CONCLUSIONS

We have explored practical algorithms and implementations for computing linking numbers for efficient monitoring and verification of the topology of loopy structures in computer graphics and animation. Linking-number certificates provide necessary (but not sufficient) conditions to ensure the absence of topology errors in simulation/processing that would go undetected previously. We hope that the community can benefit from these improved correctness monitoring techniques especially given the many immediate uses in graphics applications, e.g., yarn-level cloth simulation, that have previously relied on linkage correctness but lacked practical tools for verification.

By comparing multiple exact and approximate linking-number numerical methods, and CPU and GPU implementations, we discovered that several methods give accurate and fast results for many types of problems, but alas there are no clear winners. Direct summation (DS) using exact formulae, although popular outside graphics for its simplicity and accuracy, was uncompetitive with the other methods for speed except for very small examples; GPU acceleration of the double summation was competitive in some cases. Our

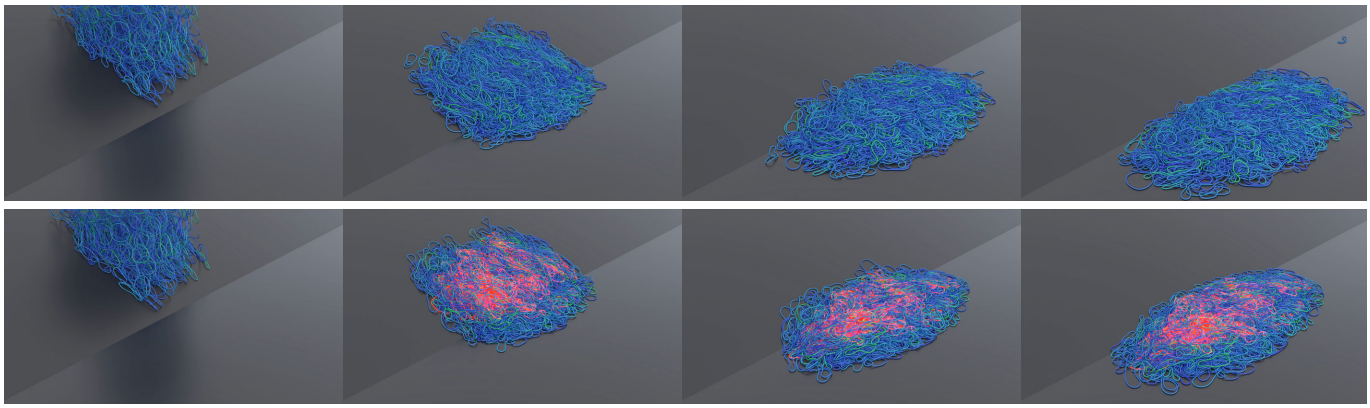


Fig. 16. **Rubber Band Test:** An initially unlinked set of 1024 rubber bands falls onto an incline then slides onto the floor, as simulated with hair-like strands in Houdini's Vellum solver. We verify that (for typical contact/constraint iterations) (Top) the loops avoid spurious linking using 20 substeps per frame, whereas (Bottom) only 4 substeps per frame resulted in 322 linked rubber bands (shown in red).

CPU parallel implementation of counting crossings (CC) was perhaps the simplest and fastest method for many problems; it relies on finding a suitable regular projection which is often cited as a weakness and potential source of numerical problems, but we were unable to generate incorrect results with it in double precision. CPU Barnes-Hut was a strong performer and competitive with CC in most cases, and GPU-accelerated Barnes-Hut was the fastest on several of the very largest examples, most notably for DNA ribbons. In some cases the GPU method struggles with accuracy due to having only dipole terms and single precision, and our GPU hardware and implementation were unable to store Barnes-Hut trees for large numbers of small loops, e.g., chainmail, for which CC or DS did best. The CPU-based Fast Multipole Method (FMM) was able to achieve high-accuracy fast summations due to the increased number of multipole terms, e.g., relative to Barnes-Hut, but it did not translate to beating CPU Barnes-Hut in general.

5.1 Limitations and Future Work

There are several limitations of our approaches, and opportunities for future work. While changes in the linking number between two curves imply a change in topology, the same linking number does not rule out intermediate topological events, e.g., two knit yarn loops with λ zero could "pull through" twice during a simulation and still have zero linkage at the end. Furthermore, collision violations can occur without a topology change: for example, given a key ring with three rigid keys, it is physically impossible for two adjacent keys to swap positions, yet the topology would still be maintained after a swap. Therefore, our approach can flag linking number changes, but a passing certificate does not preclude other errors.

We use AABB-Trees to find overlapping loop AABBs in Potential Link Search (PLS) and overlapping spline segment AABBs in Discretization, but tighter bounds than AABB might reduce the number of linking number calculations and discretized segments enough to provide further speedups.

We have considered linking numbers between closed loops and virtually closed loops for fixed patterns, but many loopy structures involve open loops, e.g., knittable yarn models [Guo et al. 2020; Narayanan et al. 2018, 2019; Wu et al. 2018, 2019] and hair, and it would be useful to generalize these ideas for open and "nearly closed" loops, e.g., to reason about "pull through" between adjacent yarn rows.

Topology checks could be closely integrated with simulations and geometry processing to encourage topologically correct results and enable roll-back/restart modes. Our proposed method with the "early exit" strategy could be better optimized to aggressively verify links before finishing PLS and Discretization on the entire model.

Finally, our GPU implementations and comparisons could be improved by specialized implementations for linking-number computations, all-at-once loop-loop processing instead of serial, better memory management for many-loop scenarios, e.g., chainmail, and also more recent hardware.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for the constructive and detailed feedback. We thank Jonathan Leaf, Xinru Hua, Paul Liu, Gilbert Bernstein, and Madeleine Yip for helpful discussions, and Steve Marschner, Purvi Goel, and Jiayi Eris Zhang for proofreading the final manuscript. We also thank Cem Yuksel for the stitch mesh source code. Ante Qu's research was supported in part by the National Science Foundation (DGE-1656518). We acknowledge SideFX for donated Houdini licenses and Google Cloud Platform for donated compute resources. Mitsuba Renderer was also used to produce renders. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Zin Arai. 2013. A rigorous numerical algorithm for computing the linking number of links. *Nonlinear Theory and Its Applications, IEICE* 4, 1 (2013), 104–110.

David Baraff and Andrew Witkin. 1998. Large Steps in Cloth Simulation. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98)*. Association for Computing Machinery, New York, NY, USA, 43–54. https://doi.org/10.1145/280814.280821

David Baraff, Andrew Witkin, and Michael Kass. 2003. Untangling Cloth. *ACM Trans. Graph.* 22, 3 (July 2003), 862–870. https://doi.org/10.1145/882262.882357

Jernej Barbič and Doug L. James. 2010. Subspace Self-Collision Culling. *ACM Trans. on Graphics (SIGGRAPH 2010)* 29, 4 (2010), 81:1–81:9.

Gavin Barill, Neil G. Dickson, Ryan Schmidt, David I. W. Levin, and Alec Jacobson. 2018. Fast winding numbers for soups and clouds. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 1–12.

Josh Barnes and Piet Hut. 1986. A hierarchical O(N log N) force-calculation algorithm. *Nature* 324, 6096 (1986), 446–449.

Julien Basch. 1999. *Kinetic Data Structures.* Ph.D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) Guibas, Leonidas J. AAI9943622.

Mitchell Berger. 2009. Topological Quantities: Calculating Winding, Writhing, Linking, and Higher Order Invariants. *Lecture Notes in Mathematics* 1973 (03 2009). https://doi.org/10.1007/978-3-642-00837-5_2

Miklós Bergou, Basile Audoly, Etienne Vouga, Max Wardetzky, and Eitan Grinspun. 2010. Discrete Viscous Threads. *ACM Trans. Graph.* 29, 4, Article 116 (July 2010), 10 pages. https://doi.org/10.1145/1778765.1778853

Miklós Bergou, Max Wardetzky, Stephen Robinson, Basile Audoly, and Eitan Grinspun. 2008. Discrete Elastic Rods. *ACM Transactions on Graphics (SIGGRAPH)* 27, 3 (aug 2008), 63:1–63:12.

Gilbert Louis Bernstein and Chris Wojtan. 2013. Putting Holes in Holey Geometry: Topology Change for Arbitrary Surfaces. *ACM Trans. Graph.* 32, 4, Article 34 (July 2013), 12 pages. https://doi.org/10.1145/2461912.2462027

Enrico Bertolazzi, Riccardo Ghiloni, and Ruben Specogna. 2019. Efficient computation of linking number with certification. *arXiv preprint arXiv:1912.13121* (2019).

Robert Bridson, Ronald Fedkiw, and John Anderson. 2002. Robust Treatment of Collisions, Contact and Friction for Cloth Animation. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques* (San Antonio, Texas) *(SIGGRAPH '02)*. Association for Computing Machinery, New York, NY, USA, 594–603. https://doi.org/10.1145/566570.566623

Tyson Brochu, Essex Edwards, and Robert Bridson. 2012. Efficient Geometrically Exact Continuous Collision Detection. *ACM Trans. Graph.* 31, 4, Article 96 (July 2012), 7 pages. https://doi.org/10.1145/2185520.2185592

Hervé Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra. 2021. 2D and 3D Linear Geometry Kernel. In *CGAL User and Reference Manual* (5.2.1 ed.). CGAL Editorial Board. https://doc.cgal.org/5.2.1/Manual/packages.html#PkgKernel23

Martin Burtscher and Keshav Pingali. 2011. An efficient CUDA implementation of the tree-based Barnes Hut N-body algorithm. In *GPU Computing Gems Emerald Edition*, Wen mei W. Hwu (Ed.). Morgan Kaufmann, Boston, 75 – 92. https://doi.org/10.1016/B978-0-12-384988-5.00006-1

Cris Cecka and Simon Layton. 2015. FMMTL: FMM Template Library A Generalized Framework for Kernel Matrices. In *Numerical Mathematics and Advanced Applications - ENUMATH 2013*, Assyr Abdulle, Simone Deparis, Daniel Kressner, Fabio Nobile, and Marco Picasso (Eds.). Springer International Publishing, Cham, 611–620.

Johnny T Chang, Jingyi Jin, and Yizhou Yu. 2002. A practical model for hair mutual interactions. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*. 73–80.

Hongwei Cheng, Leslie Greengard, and Vladimir Rokhlin. 1999. A Fast Adaptive Multipole Algorithm in Three Dimensions. *J. Comput. Phys.* 155, 2 (1999), 468 – 498. https://doi.org/10.1006/jcph.1999.6355

Nicolas Clauvelin, Wilma K. Olson, and Irwin Tobias. 2012. Characterization of the Geometry and Topology of DNA Pictured As a Discrete Collection of Atoms. *Journal of Chemical Theory and Computation* 8, 3 (2012), 1092–1107. https://doi.org/10.1021/ct200657e

Paul De Casteljau. 1959. Outillages méthodes calcul. *Andr e Citro en Automobiles SA, Paris* (1959).

Tamal K. Dey, Fengtao Fan, and Yusu Wang. 2013. An efficient computation of handle and tunnel loops via Reeb graphs. *ACM Transactions on Graphics (TOG)* 32, 4 (2013), 1–10. Source code here: https://web.cse.ohio-state.edu/~dey.8/handle/ReebHanTun-download/reebhantun.html.

Tamal K. Dey, Kuiyu Li, and Jian Sun. 2009. Computing handle and tunnel loops with knot linking. *Computer-Aided Design* 41, 10 (2009), 730 – 738. https://doi.org/10.1016/j.cad.2009.01.001

Herbert Edelsbrunner and Afra Zomorodian. 2001. Computing linking numbers of a filtration. In *Algorithms in Bioinformatics*, Olivier Gascuel and Bernard M. E. Moret (Eds.). Springer, Springer, Berlin, Heidelberg, 112–127.

Francisco R. Feito and Juan Carlos Torres. 1997. Inclusion test for general polyhedra. *Computers & Graphics* 21, 1 (1997), 23 – 30. https://doi.org/10.1016/S0097-8493(96)00067-2

F. Brock Fuller. 1978. Decomposition of the linking number of a closed ribbon: a problem from molecular biology. *Proceedings of the National Academy of Sciences* 75, 8 (1978), 3557–3561.

Leslie Greengard and Vladimir Rokhlin. 1987. A fast algorithm for particle simulations. *J. Comput. Phys.* 73, 2 (1987), 325 – 348. https://doi.org/10.1016/0021-9991(87)90140-9

Chen Greif, Shiwen He, and Paul Liu. 2015. SYM-ILDL: Incomplete LDL^T Factorization of Symmetric Indefinite and Skew-Symmetric Matrices. *CoRR* abs/1505.07589 (2015). arXiv:1505.07589 http://arxiv.org/abs/1505.07589

Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org.

Leonidas Guibas, An Nguyen, Daniel Russel, and Li Zhang. 2002. Collision Detection for Deforming Necklaces. In *Proceedings of the Eighteenth Annual Symposium on Computational Geometry* (Barcelona, Spain) *(SCG '02)*. Association for Computing Machinery, New York, NY, USA, 33–42. https://doi.org/10.1145/513400.513405

Runbo Guo, Jenny Lin, Vidya Narayanan, and James McCann. 2020. Representing Crochet with Stitch Meshes. In *Symposium on Computational Fabrication* (Virtual Event, USA) *(SCF '20)*. Association for Computing Machinery, New York, NY, USA, Article 4, 8 pages. https://doi.org/10.1145/3424630.3425409

David Harmon, Daniele Panozzo, Olga Sorkine, and Denis Zorin. 2011. Interference-Aware Geometric Modeling. *ACM Trans. Graph.* 30, 6 (Dec. 2011), 1–10. https://doi.org/10.1145/2070781.2024171

Edmond S.L. Ho and Taku Komura. 2009. Character Motion Synthesis by Topology Coordinates. *Computer Graphics Forum* 28, 2 (2009), 299–308. https://doi.org/10.1111/j.1467-8659.2009.01369.x

Edmond S.L. Ho, Taku Komura, Subramanian Ramamoorthy, and Sethu Vijayakumar. 2010. Controlling humanoid robots in topology coordinates. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 178–182. https://doi.org/10.1109/IROS.2010.5652787

Libo Huang, Torsten Hädrich, and Dominik L. Michels. 2019. On the Accurate Large-Scale Simulation of Ferrofluids. *ACM Trans. Graph.* 38, 4, Article 93 (July 2019), 15 pages. https://doi.org/10.1145/3306346.3322973

Geoffrey Irving, Joseph M. Teran, and Ronald Fedkiw. 2004. Invertible Finite Elements for Robust Simulation of Large Deformation. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Grenoble, France) *(SCA '04)*. Eurographics Association, Goslar, DEU, 131–140. https://doi.org/10.1145/1028523.1028541

Alec Jacobson, Ladislav Kavan, and Olga Sorkine-Hornung. 2013. Robust Inside-Outside Segmentation Using Generalized Winding Numbers. *ACM Trans. Graph.* 32, 4, Article 33 (July 2013), 12 pages. https://doi.org/10.1145/2461912.2461916

Jonathan M. Kaldor, Doug L. James, and Steve Marschner. 2008. Simulating Knitted Cloth at the Yarn Level. In *ACM SIGGRAPH 2008 Papers* (Los Angeles, California) *(SIGGRAPH '08)*. Association for Computing Machinery, New York, NY, USA, Article 65, 9 pages. https://doi.org/10.1145/1399504.1360664

Jonathan M. Kaldor, Doug L. James, and Steve Marschner. 2010. Efficient Yarn-Based Cloth with Adaptive Contact Linearization. In *ACM SIGGRAPH 2010 Papers* (Los Angeles, California) *(SIGGRAPH '10)*. Association for Computing Machinery, New York, NY, USA, Article 105, 10 pages. https://doi.org/10.1145/1833349.1778842

Danny M. Kaufman, Rasmus Tamstorf, Breannan Smith, Jean-Marie Aubry, and Eitan Grinspun. 2014. Adaptive Nonlinearity for Collisions in Complex Rod Assemblies. *ACM Trans. Graph.* 33, 4, Article 123 (July 2014), 12 pages. https://doi.org/10.1145/2601097.2601100

Theodore Kim. 2020. A Finite Element Formulation of Baraff-Witkin Cloth. *Computer Graphics Forum* 39, 8 (2020), 171–179. https://doi.org/10.1111/cgf.14111

Theodore Kim, Fernando De Goes, and Hayley Iben. 2019. Anisotropic Elasticity for Inversion-Safety and Element Rehabilitation. *ACM Trans. Graph.* 38, 4, Article 69 (July 2019), 15 pages. https://doi.org/10.1145/3306346.3323014

Konstantin Klenin and Jörg Langowski. 2000. Computation of writhe in modeling of supercoiled DNA. *Biopolymers: Original Research on Biomolecules* 54, 5 (2000), 307–317. https://doi.org/10.1002/1097-0282(20001015)54:5<307::AID-BIP20>3.0.CO;2-Y

Brad A. Krajina, Audrey Zhu, Sarah C. Heilshorn, and Andrew J. Spakowitz. 2018. Active DNA Olympic Hydrogels Driven by Topoisomerase Activity. *Phys. Rev. Lett.* 121 (Oct 2018), 148001. Issue 14. https://doi.org/10.1103/PhysRevLett.121.148001

Jonathan Leaf, Rundong Wu, Eston Schweickart, Doug L. James, and Steve Marschner. 2018. Interactive Design of Periodic Yarn-Level Cloth Patterns. *ACM Trans. Graph.* 37, 6, Article 202 (Dec. 2018), 15 pages. https://doi.org/10.1145/3272127.3275105

Charles A. Micchelli and Hartmut Prautzsch. 1989. Uniform refinement of curves. *Linear Algebra Appl.* 114–115 (1989), 841–870. https://doi.org/10.1016/0024-3795(89)90495-3

John Milnor. 1954. Link groups. *Annals of Mathematics* (1954), 177–195.

Edward L. F. Moore. 2006. *Computational topology of spline curves for geometric and molecular approximations.* Ph.D. Dissertation. University of Connecticut. Advisor(s) Peters, Thomas J.

Vidya Narayanan, Lea Albaugh, Jessica Hodgins, Stelian Coros, and James Mccann. 2018. Automatic Machine Knitting of 3D Meshes. *ACM Trans. Graph.* 37, 3, Article 35 (Aug. 2018), 15 pages. https://doi.org/10.1145/3186265

Vidya Narayanan, Kui Wu, Cem Yuksel, and James McCann. 2019. Visual Knitting Machine Programming. *ACM Trans. Graph.* 38, 4, Article 63 (July 2019), 13 pages. https://doi.org/10.1145/3306346.3322995

Florian T. Pokorny, Johannes A. Stork, and Danica Kragic. 2013. Grasping objects with holes: A topological approach. In *2013 IEEE International Conference on Robotics and Automation*. 1100–1107. https://doi.org/10.1109/ICRA.2013.6630710

Dale Rolfsen. 1976. *Knots and links.* Publish or Perish, Berkeley, CA.

David Salesin, Jorge Stolfi, and Leonidas Guibas. 1989. Epsilon Geometry: Building Robust Algorithms from Imprecise Computations. In *Proceedings of the Fifth Annual Symposium on Computational Geometry* (Saarbruchen, West Germany) *(SCG '89)*. Association for Computing Machinery, New York, NY, USA, 208—-217. https://doi.org/10.1145/73833.73857

Andrew Selle, Michael Lentine, and Ronald Fedkiw. 2008. A Mass Spring Model for Hair Simulation. *ACM Trans. Graph.* 27, 3 (Aug. 2008), 1–11. https://doi.org/10.1145/1360612.1360663

Jonathan Richard Shewchuk. 1997. Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates. *Discrete & Computational Geometry* 18, 3 (Oct. 1997), 305–363.

Zachary Sierzega, Jeff Wereszczynski, and Chris Prior. 2020. WASP: A software package for correctly characterizing the topological development of ribbon structures. *bioRxiv* (2020). https://doi.org/10.1101/2020.09.17.301309

Jonas Spillmann and Matthias Teschner. 2008. An Adaptive Contact Model for the Robust Simulation of Knots. *Computer Graphics Forum* 27, 2 (2008), 497–506. https://doi.org/10.1111/j.1467-8659.2008.01147.x

Alessandro Tasora, Radu Serban, Hammad Mazhar, Arman Pazouki, Daniel Melanz, Jonathan Fleischmann, Michael Taylor, Hiroyuki Sugiyama, and Dan Negrut. 2016. Chrono: An Open Source Multi-physics Dynamics Engine. In *High Performance Computing in Science and Engineering*, Tomáš Kozubek, Radim Blaheta, Jakub Šístek, Miroslav Rozložník, and Martin Čermák (Eds.). Springer International Publishing, Cham, 19–49.

Adriaan Van Oosterom and Jan Strackee. 1983. The Solid Angle of a Plane Triangle. *IEEE Transactions on Biomedical Engineering* BME-30, 2 (1983), 125–126. https://doi.org/10.1109/TBME.1983.325207

Pascal Volino and Nadia Magnenat-Thalmann. 2006. Resolving Surface Collisions through Intersection Contour Minimization. In *ACM SIGGRAPH 2006 Papers* (Boston, Massachusetts) *(SIGGRAPH '06)*. Association for Computing Machinery, New York, NY, USA, 1154–1159. https://doi.org/10.1145/1179352.1142007

Monan Wang and Jiaqi Cao. 2021. A review of collision detection for deformable objects. *Computer Animation and Virtual Worlds* (2021).

Kui Wu, Xifeng Gao, Zachary Ferguson, Daniele Panozzo, and Cem Yuksel. 2018. Stitch Meshing. *ACM Trans. Graph.* 37, 4, Article 130 (July 2018), 14 pages. https://doi.org/10.1145/3197517.3201360

Kui Wu, Hannah Swan, and Cem Yuksel. 2019. Knittable Stitch Meshes. *ACM Trans. Graph.* 38, 1, Article 10 (Jan. 2019), 13 pages. https://doi.org/10.1145/3292481

Kui Wu and Cem Yuksel. 2017. Real-time Fiber-level Cloth Rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D 2017)* (San Francisco, CA). ACM, New York, NY, USA, 8. https://doi.org/10.1145/3023368.3023372

Rundong Wu, Joy Xiaoji Zhang, Jonathan Leaf, Xinru Hua, Ante Qu, Claire Harvey, Emily Holtzman, Joy Ko, Brooks Hagan, Doug James, François Guimbretière, and Steve Marschner. 2020. Weavecraft: An Interactive Design and Simulation Tool for 3D Weaving. *ACM Trans. Graph.* 39, 6, Article 210 (Nov. 2020), 16 pages. https://doi.org/10.1145/3414685.3417865

Christopher Yu, Henrik Schumacher, and Keenan Crane. 2020. Repulsive Curves. *ACM Trans. Graph.* (2020). Conditionally accepted preprint.

Cem Yuksel, Jonathan M. Kaldor, Doug L. James, and Steve Marschner. 2012. Stitch Meshes for Modeling Knitted Clothing with Yarn-Level Detail. *ACM Trans. Graph.* 31, 4, Article 37 (July 2012), 12 pages. https://doi.org/10.1145/2185520.2185533

Dmitry Zarubin, Vladimir Ivan, Marc Toussaint, Taku Komura, and Sethu Vijayakumar. 2012. Hierarchical Motion Planning in Topological Representations. In *Proceedings of Robotics: Science and Systems*. Sydney, Australia. https://doi.org/10.15607/RSS.2012.VIII.059

Liangjun Zhang, Young J. Kim, Gokul Varadhan, and Dinesh Manocha. 2007. Generalized penetration depth computation. *Computer-Aided Design* 39, 8 (2007), 625 – 638. https://doi.org/10.1016/j.cad.2007.05.012 Solid and Physical Modeling 2006.

Changxi Zheng and Doug L. James. 2012. Energy-Based Self-Collision Culling for Arbitrary Mesh Deformations. *ACM Trans. Graph.* 31, 4, Article 98 (July 2012), 12 pages. https://doi.org/10.1145/2185520.2185594

A DISCRETIZATION PROOFS

These proofs accompany the Discretization discussion in §3.3.1.

Theorem A.1. *If all input loops have only polynomial spline segments, and no two loops intersect, then Algorithm 3 terminates in finite time if it uses tight AABBs as the bounding volumes in its overlap check.*

Proof. Write each input spline segment using a parameter t that has a domain of $(0, 1)$. Since there is a finite number of input spline segments and each segment is polynomial over t with a finite domain, the derivative along the curve with respect to t has a global maximum, M. Since the loops don't intersect, let d be the minimum separation between two loops. After pass P of the algorithm, the parameter t of any "under-refined" spline segment now spans a domain of length 2^{-P}. Because polynomial spline segments are continuous, their arc lengths must be at most $2^{-P}M$. Their bounding AABBs must therefore have diameters that are also at most $2^{-P}M$. If the minimum separation between any two loops is d, then after $P > \log_2(2M/d)$ passes, no two bounding boxes of different loops can overlap, and so all splines segments must have been discretized. □

Theorem A.2. *If Algorithm 3 terminates, then it produces a line segment discretization that is homotopically equivalent to the input loops.*

Proof. We can deform the input following the steps of the algorithm. At each pass, let us deform one loop, ξ, at a time. All the spline segments of this loop, ξ, that are "processed" during the pass can be deformed continuously to their line segments (which share the same endpoints) by parameterizing the line segment to have the same t domain and performing a point-wise straight-line deformation at each t. Because our bounding boxes are convex, this deformation sweeps a surface that is entirely within the bounding box. The box overlaps with no boxes from other loops, so loop ξ does not cross any other loop during the deformation. Notice, however, that we do not guarantee that a loop does not cross itself during its deformation; this is allowed in link homotopy and does not affect the linking number. After all the passes, every loop has been deformed into its output polyline, and we are done. □

B IMPLEMENTATION DETAILS OF LINKING NUMBER COMPUTATION ROUTINES

B.1 Direct Summation

We reproduce the direct sum expression (4) here for reference. For loops γ_1, γ_2, consisting of N_l, N_k line segments respectively, with j, i enumerating the segments respectively, we want to compute

$$\lambda(\gamma_1, \gamma_2) = \sum_i^{N_k} \sum_j^{N_l} \lambda_{ji}. \tag{16}$$

$$\lambda_{ji} = \frac{1}{2\pi}\left(\operatorname{atan}\left(\frac{\mathbf{a}\cdot(\mathbf{b}\times\mathbf{c})}{|\mathbf{a}||\mathbf{b}||\mathbf{c}| + (\mathbf{a}\cdot\mathbf{b})|\mathbf{c}| + (\mathbf{c}\cdot\mathbf{a})|\mathbf{b}| + (\mathbf{b}\cdot\mathbf{c})|\mathbf{a}|}\right) + \operatorname{atan}\left(\frac{\mathbf{c}\cdot(\mathbf{d}\times\mathbf{a})}{|\mathbf{c}||\mathbf{d}||\mathbf{a}| + (\mathbf{c}\cdot\mathbf{d})|\mathbf{a}| + (\mathbf{a}\cdot\mathbf{c})|\mathbf{d}| + (\mathbf{d}\cdot\mathbf{a})|\mathbf{c}|}\right)\right). \tag{17}$$

In our implementation, both on the CPU and for large examples on the GPU, we multithread the outer loop of i, and perform angle summations (using the arctan addition formula) in the inner loop with a single arctangent at the end of each loop, and perform a sum reduction on the total. This results in N_k arctangents. When performing the inner loop, we split the j loop into batches, and each threadblock fetches the entire $\mathbf{l}_j$ batch in parallel into shared memory before each i thread iterates through the batch. See Alg. 7 for more details. To maximize GPU usage, for small examples ($N_l N_k < 1.4 \times 10^6$), we parallelize both for-loops, and within each iteration we perform one arctangent.

ALGORITHM 7: ComputeLink: Direct Summation via Angle Summation, CPU or Large Input on the GPU

Input : $\{\mathbf{l}_j\}$, $\{\mathbf{k}_i\}$, the two polyline loops
Output: λ, the linking number

// This function computes the linking number between two closed polylines based on [Bertolazzi et al. 2019]. The function $s(x, y)$ outputs +1 if (($y > 0$) or ($y = 0$ and $x < 0$)), −1 otherwise.

Function ComputeLinkDS($\{\mathbf{l}_j\}$, $\{\mathbf{k}_i\}$):
  $\lambda \leftarrow 0$;
  $N_l \leftarrow |\{\mathbf{l}_j\}|$; $N_k \leftarrow |\{\mathbf{k}_i\}|$;
  parallel for $i \in [0, N_k - 1]$ **do**
    $x_S \leftarrow 1$; $y_S \leftarrow 0$;
    $S \leftarrow -1$; $\lambda_i \leftarrow 0$;
    // If this is the GPU, preload a batch of $\mathbf{l}_j$ values here into shared memory, and then add a threadfence.
    for $j \in [0, N_l - 1]$ **do**
      $\mathbf{a} \leftarrow \mathbf{l}_j - \mathbf{k}_i$;
      $\mathbf{b} \leftarrow \mathbf{l}_j - \mathbf{k}_{i+1}$;
      $\mathbf{c} \leftarrow \mathbf{l}_{j+1} - \mathbf{k}_{i+1}$;
      $\mathbf{d} \leftarrow \mathbf{l}_{j+1} - \mathbf{k}_i$;
      $p \leftarrow \mathbf{a} \cdot (\mathbf{b} \times \mathbf{c})$;
      $d_1 \leftarrow |\mathbf{a}||\mathbf{b}||\mathbf{c}| + \mathbf{a} \cdot \mathbf{b}|\mathbf{c}| + \mathbf{b} \cdot \mathbf{c}|\mathbf{a}| + \mathbf{c} \cdot \mathbf{a}|\mathbf{b}|$;
      $d_2 \leftarrow |\mathbf{a}||\mathbf{d}||\mathbf{c}| + \mathbf{a} \cdot \mathbf{d}|\mathbf{c}| + \mathbf{d} \cdot \mathbf{c}|\mathbf{a}| + \mathbf{c} \cdot \mathbf{a}|\mathbf{d}|$;
      $x' \leftarrow d_1 d_2 - p^2$;
      $y' \leftarrow p(d_1 + d_2)$;
      if $((s(d_1, p)s(d_2, p) > 0)$ **and** $(s(d_1, p)s(x', y') < 0))$ **then**
        $\lambda_i \leftarrow \lambda_i + s(d_1, p)$;
      end
      $x'' \leftarrow x_S x' - y_S y'$;
      $y'' \leftarrow x_S y' + y_S x'$;
      if $((s(x', y')\, S > 0)$ **and** $(s(x'', y'')\, S < 0))$ **then**
        $\lambda_i \leftarrow \lambda_i + S$;
      end
      $S \leftarrow s(x'', y'')$;
      $x_S \leftarrow x''/\max(|x''|, |y''|)$;
      $y_S \leftarrow y''/\max(|x''|, |y''|)$;
    end
    $\lambda_i \leftarrow \lambda_i + \text{atan2}(y_S, x_S)/(2\pi)$;
    $\lambda \leftarrow \lambda + \lambda_i$; // Reduce this summation.
  end
  return λ;

B.2 Barnes-Hut: Derivatives of $G(\tilde{\mathbf{r}}_1, \tilde{\mathbf{r}}_2)$, Moment Computation, and Parallel Implementation

Let $\mathbf{r} = \tilde{\mathbf{r}}_2 - \tilde{\mathbf{r}}_1$. Then

$$\boldsymbol{\nabla} G(\tilde{\mathbf{r}}_1, \tilde{\mathbf{r}}_2) = \frac{\mathbf{r}}{4\pi|\mathbf{r}|^3}. \tag{18}$$

$$\boldsymbol{\nabla}^2 G(\tilde{\mathbf{r}}_1, \tilde{\mathbf{r}}_2) = \frac{I_{3\times 3}|\mathbf{r}|^2 - 3\mathbf{r}\otimes\mathbf{r}}{4\pi|\mathbf{r}|^5}. \tag{19}$$

$$\boldsymbol{\nabla}^3 G(\tilde{\mathbf{r}}_1, \tilde{\mathbf{r}}_2) = \frac{-3\sum_i (\mathbf{r}\otimes\mathbf{e}_i\otimes\mathbf{e}_i + \mathbf{e}_i\otimes\mathbf{r}\otimes\mathbf{e}_i + \mathbf{e}_i\otimes\mathbf{e}_i\otimes\mathbf{r})}{4\pi|\mathbf{r}|^5} + \frac{15\mathbf{r}\otimes\mathbf{r}\otimes\mathbf{r}}{4\pi|\mathbf{r}|^7}, \tag{20}$$

where $\mathbf{e}_i$ is the i-th basis element and repeated indices are summed. Note that these are identical to the basis functions in Appendix A of [Barill et al. 2018], except the third derivative there (Eq. 24 in [Barill et al. 2018]) has a misprint.

For a line segment element, we set $\tilde{\mathbf{r}}$ to the midpoint, and so $\mathbf{c}_M$ is the difference between the endpoints, $C_D = 0$, and $C_Q = \frac{1}{12}\mathbf{c}_M \otimes \mathbf{c}_M \otimes \mathbf{c}_M$.

When we combine bounding boxes into their parents, we need to sum and shift moments. Let $\mathbf{r}_c$ be the displacement $\tilde{\mathbf{r}}_c - \tilde{\mathbf{r}}_p$ (from parent node to child node), and simply use

$$\mathbf{c}_{Mp} = \sum_c \mathbf{c}_{Mc}, \tag{21}$$

$$C_{Dp} = \sum_c C_{Dc} + \mathbf{c}_{Mc}\mathbf{r}_c^T, \tag{22}$$

$$C_{Qpijk} = \sum_c C_{Qcijk} + C_{Dpij}r_{ck} + C_{Dpik}r_{cj} + c_{Mci}r_{cj}r_{ck}. \tag{23}$$

On the CPU, we parallelized the two-tree method by gathering the list of child node pairs to evaluate after each recursive pass, and proceeding to children only after a pass is finished (breadth-first). We use a breadth-first pass to multithread the evaluation when it has between 1000 and 500,000 node pairs to evaluate. Beyond the upper limit, memory allocation overhead dominates, and we simply evaluate the depth-first recursive algorithm in parallel for the 500,000+ node pairs.

While we recommend the parallel two-trees algorithm when evaluating on the CPU, we parallelize a one-tree algorithm when evaluating on the GPU by basing it on the GPU Barnes-Hut N-Body algorithm from [Burtscher and Pingali 2011]. In particular, we use Kernels 1 through 5. We first build a tree (Kernels 1 through 4) for each loop, then for each loop pair, we evaluate Kernel 5 using the source tree from one loop against target points from the other loop. We modified kernel 3 to gather and compute the monopole and dipole moments of each node and also propagate the max segment length, l_M, from its children. Because segments are only inserted at their midpoints, we replace the β far-field condition in Alg. 6 by checking whether $|\mathbf{r}_1 - \text{tree}.\mathbf{r}| > \beta(\text{tree}.R + \text{tree}.l_M/2 + l_1/2)$, where l_1 is the length of the segment at $\mathbf{r}_1$. We modified Kernel 5 to evaluate the dipole expression when the β far-field condition is met by all threads; otherwise the arctan expression. However, one key difference from [Burtscher and Pingali 2011] is that because the source and target trees are different and our input is already spatially coherent, Kernel 5 is actually slower if it evaluates the points in the sorted order from Kernel 4 than if it evaluates the points in the input order. So we keep Kernel 4 for sorting the individual source trees, but we evaluate Kernel 5 on the target evaluation points in the input order instead of the sorted order.

C FORMING VIRTUAL CLOSED LOOPS IN BRAIDS

This section describes the formation of virtual closed loops for Open-Curve Verification §4.3.5.

Let $\alpha_1, \cdots, \alpha_L$ denote the open curves, and let one rigid end be the "left" end and the other be the "right" end. Furthermore, define a small rigid end volume attached to each end, such that no curve passes through it. Then follow this procedure, also illustrated in Figure 14:

(1) To ease bookkeeping, parameterize the curves so that they are oriented left to right.
(2) Form L total closed loops, γ_i. For each $i \in [1, L-1]$, form γ_i from α_i and α_{i+1} by doing the following:
   (a) Start with a copy of α_i,
   (b) Append a virtual connection, entirely within the right end volume, from the right end of α_i to the right end of α_{i+1}, carefully making sure it intersects no other virtual curve,
   (c) Append α_{i+1} in the reverse direction, and then
   (d) Append a virtual connection, entirely within the left end volume, from the left end of α_{i+1} to the left end of α_i, taking the same care to ensure it intersects no other curve.
(3) Also form γ_L using α_L and α_1 the same way as the last step. At this point, every curve α should be part of two closed loops γ, and every loop γ should contain two curves from the model.
(4) Run our method from §3 using the input loops $\{\gamma_i\}$, with the difference that, in the PLS stage, we exclude pairs (i, j) where γ_i and γ_j share an α as a component (which is when $i + 1 = j$ or ($i = 1$ and $j = L$)) from getting inserted into P. The rest of the method proceeds as usual, computing a sparse linking matrix as a certificate.
(5) When this procedure is used again on a deformed model, the virtual connections at each end *must be appended with the same exact paths*, up to a rigid transformation (this is possible because of the rigid ends), ensuring that the virtual connections don't pull through each other.

We also require that $L \geq 4$.

The certificate meaningfully verifies the topology between open curves in the model, as long as no real curve enters the rigid end volumes and the end volumes don't mutually interpenetrate (ensuring no real or virtual curve moves through a virtual connection). To see how, suppose a single pull through occurs, between curves α_i and α_j, $i < j$. Then this certificate must change, for these reasons:

- Call indices "cyclically adjacent" if they are either adjacent (such as i and $i + 1$) or the first and last (such as 1 and L). If (i, j) are not cyclically adjacent, then the linkage between γ_i and γ_j will be computed (because they don't share a curve), and found to be different, since the former contains α_i and the latter contains α_j.
- Otherwise if (i, j) are cyclically adjacent, then the linkage between $\gamma_{i-1 \text{ (modulo } L)}$ and γ_j will be calculated (because $L \geq 4$, they don't share a curve) and found to be different, because the former contains α_i, and the latter contains α_j.

Conversely, if a linking number changes, then there must have been at least one topology violation among the real curves, because no virtual connection participated in any pull through.

Furthermore, this procedure can be applied independently on many interacting braids as long as each braid has at least 4 curves.