Yarn-Level Simulation of Knitted Fabric in the Browser: Persistent Contacts, Discrete-Rod Bending, and a Behavioural Validation Gate on the GPU

Irfan Šarić

December 2020

Abstract

A knitted fabric is not a sheet. It is a single strand of yarn drawn into tens of thousands of interlocking loops, and the mechanical behaviour that makes knitwear recognisable—the curl of a cut edge, the wildly different stretch along a row versus up a column, the way a snag travels and a dropped stitch ladders—emerges from how those loops press and slide against one another, not from any surface. We describe *yarnify*, a yarn-level knit simulator that models the fabric as one continuous polyline threaded through precomputed *persistent contacts* at every stitch crossing, gives it axial stretch, discrete elastic-rod bending about a rest curvature, a two-sided contact tether that holds each crossing near its assembled separation, and velocity-filter Coulomb friction, and integrates it with a damped symplectic step. The entire solver is expressed once as a set of forces that are the gradient of a single potential—verified term by term against finite differences—and runs on the GPU through WebGPU compute shaders, with per-contact forces scattered into shared nodes by fixed-point integer atomics so that contact at knit density parallelises without a race. The yarn is drawn as an instanced swept tube so the fabric reads as rounded strands rather than lines.

The contribution we most want to defend is not that it renders: it is that the mechanics are *right*, and that this is proven rather than asserted. A behavioural gate relaxes a free stockinette patch and measures the two signatures a normal-mapped triangle sheet cannot fake—its free edges curl out of plane, and its in-plane stiffness is strongly anisotropic between the course and wale directions—while checking that the relaxation dissipates energy rather than exploding. On a 256-stitch patch (2048 nodes, 480 interlock contacts) the energy falls from 866 to 430 units (the tether holds the loops in their arched arrangement, so the fabric keeps its stored bending rather than flattening into a tangle) and the measured course/wale stiffness ratio is $\times 6.4$, well away from the near-isotropy of a sheet. The same WGSL compute shader that runs in the browser is executed headless through a native `wgpu` harness and matches the CPU reference to the fixed-point quantum (force error 1.1×10^{-3}, trajectory error 1.1×10^{-3} mm over 120 steps), and the yarn render shader is rasterised on the same GPU to confirm the visual path draws. We report the model, its discretisation and stability, the GPU parallelisation, the validation gate and its measured numbers, near-linear cost scaling to 32 768 nodes, and an honest account of what the method does and does not reproduce—edge curl and anisotropy magnitude yes, the opposed sign of course versus wale roll no—together with the limits of unravelling under a persistent-contact model.

1 Introduction

Cloth simulation in interactive graphics has, for two decades, overwhelmingly meant a triangle mesh: a thin elastic surface with stretch and bending energies, collided against the body and itself, and dressed at render time with a normal or displacement map that *depicts* yarn without simulating it. For woven and especially for *knitted* fabric this is a convenient fiction. Knit is constructed, not woven: a single continuous strand is pulled through itself in a lattice of slip knots, and every mechanical property that distinguishes a jersey from a bedsheet is a consequence of that construction. A knitted swatch cut from a larger piece *curls*—the top and bottom edges roll one way and the side edges the other—because the loops store bending energy that the free boundary is no longer constrained to balance. A knit stretches several times more along a row (the *course*) than up a column (the *wale*) because pulling along the course unbends loops that are free to straighten, while pulling up the wale loads the yarn nearly in tension. Snag a jersey and the disturbance travels along the yarn; drop a stitch and it *ladders* down a wale as the loops below lose the neighbour that held them closed. None of these behaviours live on a surface. They live in the yarn and in the contacts between loops.

Simulating the yarn directly—every loop, every crossing, the friction and the sliding—was shown to be possible and worthwhile by Kaldor, James and Marschner [1], whose yarn-level model reproduced the drape and stretch of real garments by treating the yarn as an inextensible rod and every yarn-yarn crossing as a stiff penalty contact with velocity-filter friction. The method is faithful and expensive: contact detection between tens of thousands of yarn segments dominates the cost, and a single garment relaxed overnight. Cirio, López-Moreno and Otaduy [3, 5] made the key move that turns this from an offline curiosity into something tractable at scale: because the topology of a knit is *known*—we built it, we know exactly which loop passes through which—the crossings need not be discovered by collision detection at all. Each crossing is instead a *persistent contact* with its own sliding coordinate along the yarn, and the simulation reduces to a system of one-dimensional yarns coupled at a fixed, precomputed set of contact points. Yuksel, Kaldor, James and Marschner [6] supplied the other half: a *stitch mesh* in which each face is one stitch of a known type, giving a direct, editable generator for the loop topology and the initial yarn geometry.

This paper takes those three ideas—persistent precomputed contacts, discrete elastic-rod bending, and stitch-mesh topology—and does three things with them. First, it assembles them into a compact, self-contained model whose *entire* force law is the analytic gradient of a single scalar potential, so that correctness of the forces is not argued but checked term by term against a finite difference. Second, it pushes the whole solver onto the GPU through WebGPU compute shaders, using fixed-point integer atomics to scatter each contact's force into the two nodes it couples without a data race—the one trick that makes contact at knit density data-parallel—so that the simulation runs, to our knowledge for the first time, entirely in a web browser with no native plugin. Third, and most importantly, it subjects the result to a *behavioural* acceptance test. A knit simulator that merely draws tubes is a puppet. The test that it is genuinely *knit* is that a relaxed free patch reproduces the mechanics real knit has and a textured sheet cannot: it curls at its free edges, and its stiffness is strongly anisotropic between course and wale. We make that test the gate: `check` relaxes a patch and asserts curl, anisotropy, and bounded energy; `gpucheck` runs the exact browser shader headless and asserts it reproduces the CPU reference to tolerance; green there, or the build is not done.

Contributions. Concretely, this work offers:

- A yarn-level stockinette model—serpentine loop topology from a stitch cell, discrete elastic-rod bending about a rest curvature, a two-sided contact tether at precomputed crossings, and velocity-filter Coulomb friction—whose conservative forces are the exact gradient of a single potential and are verified against finite differences (§5–§6).
- A GPU realisation in WebGPU/WGSL compute in which per-contact and per-element forces scatter into shared nodes through fixed-point integer atomics, integration is a second pass, and the yarn is drawn as an instanced swept tube (§8–§9).
- A behavioural validation gate that measures edge curl and course/wale anisotropy on a relaxed patch, confirms energy dissipation, and checks the browser shader against the CPU reference on real silicon; with the measured numbers reported in full (§10–§11).
- An honest account of the model's reach: which knit signatures it reproduces (edge curl, anisotropy magnitude, snagging, ladder-from-a-loose-end) and which it does not (the opposed sign of course versus wale curl, and fully general topological unravelling), with the reasons (§13).

A note on the target. The browser is a deliberate constraint, not an afterthought. WebGPU exposes compute shaders and storage buffers with atomics—enough to run the whole solver on the GPU—but under tighter limits than a native context and with no escape hatch to CPU. Meeting knit density under those limits forces the parallelisation to be honest: there is no serial contact-solve to fall back on. Throughout, the native `wgpu` harness runs the identical WGSL so that "works on the GPU" is demonstrated on hardware rather than hoped for.

2 Related Work

Yarn-level cloth. The direct simulation of cloth as interacting yarns begins in earnest with Kaldor, James and Marschner [1], who modelled each yarn as an inextensible Cosserat-like rod and resolved yarn-yarn contact with stiff penalty forces and a velocity filter for friction, reproducing for the first time the characteristic large-scale stretch and drape of knitted garments from purely local yarn mechanics. Their follow-up [2] accelerated the contact computation by reusing coherence between frames, but the cost of *finding* contacts between tens of thousands of moving segments remained the dominant expense. The conceptual leap that removes that cost is due to Cirio et al. [3, 4, 5]: for a fabric of known construction the yarn crossings are fixed and enumerable, so each is represented once as a *persistent contact* carrying a Lagrangian sliding coordinate that lets yarn feed through the crossing. Contact detection disappears; the model becomes a sparse graph of one-dimensional yarns coupled at a static set of points, and scales to hundreds of thousands of loops. yarnify adopts precisely this persistent, precomputed-contact view, in the penalty (rather than constraint) form, because penalty contact maps cleanly onto an atomic force scatter on the GPU.

Stitch meshes and knit topology. Generating the loop graph is itself a problem. Yuksel et al. [6] introduced the *stitch mesh*, a coarse mesh whose every face is a single stitch of a chosen type (knit, purl, and so on) with edges labelled *course* or *wale*, and showed how to relax the resulting yarn curves into a plausible rest shape. Wu et al. [7] and Leaf et al. [8] extended stitch meshes to arbitrary surfaces and to machine-knittable structure. yarnify uses the simplest useful special case—a regular grid of stockinette cells with a serpentine yarn path—which is enough to exhibit curl and anisotropy and keeps the topology generator small and testable.

Elastic rods. The bending model for the yarn is the discrete elastic rod. Bergou et al. [9, 10] formulated discrete rods with a curvature-binormal bending energy and a discrete notion of twist, giving the standard modern treatment of one-dimensional elastica in graphics; Spillmann and Teschner's CoRdE [11] is a contemporaneous alternative. yarnify uses a reduced form: a bending energy quadratic in the turning angle between consecutive yarn segments about a nonzero *rest* angle, which is all that a centreline (twist-free) yarn model needs and which yields a compact analytic force we can verify against finite differences.

Homogenised and hybrid models. Because full yarn-level simulation is costly, a line of work replaces it at run time with a surface whose constitutive behaviour is *fitted* to yarn-level experiments. Sperl et al. [12] homogenise yarn-level mechanics into an anisotropic thin-shell material; Casafranca et al. [13] mix yarns and triangles, keeping yarns only where their behaviour matters. These methods are the pragmatic choice when the goal is a fast garment and not the yarn itself; they are complementary to, and in a sense the opposite of, the goal here, which is to keep the yarn.

The physics of the knit. That stockinette curls and is anisotropic is not folklore but measured mechanics. Poincloux, Adda-Bedia and Lechenault [14] characterise the force-extension response and the geometry of a single-yarn stockinette fabric and tie its remarkable extensibility to the reconfiguration of loops rather than the stretching of yarn—exactly the mechanism that makes the course far softer than the wale. We take their qualitative signatures (edge curl, strong course/wale anisotropy driven by loop reconfiguration) as the behavioural targets the gate must reproduce.

Cloth contact and friction. The velocity-filter friction we use for yarn-yarn sliding follows Bridson et al. [15] and the implicit-integration tradition of Baraff and Witkin [16]; penalty contact of this kind is standard. For the GPU scatter, the fixed-point atomic accumulation of forces into shared nodes is the same device used in particle- and grid-based GPU solvers such as MLS-MPM [17] to make many-to-one writes deterministic and race-free, applied here to yarn contacts.

In-browser simulation. Running physics in the browser on the GPU became possible with WebGPU's compute shaders and storage buffers. To our knowledge no prior system runs *yarn-level* knit in a browser; the closest relatives are WebGL cloth demos on triangle meshes, which do not simulate yarn, and native yarn-level systems [1, 5], which do not run without a plugin. yarnify occupies the gap.

3 Notation and Yarn Kinematics

The fabric is one yarn, discretised as an ordered list of nodes $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{N-1}$ with $\mathbf{x}_i \in \mathbb{R}^3$. Consecutive nodes are joined by yarn *segments*; the segment from $\mathbf{x}_i$ to $\mathbf{x}_{i+1}$ has vector, length and unit tangent

$$\mathbf{d}_i = \mathbf{x}_{i+1} - \mathbf{x}_i, \qquad \ell_i = \|\mathbf{d}_i\|, \qquad \mathbf{t}_i = \mathbf{d}_i/\ell_i. \tag{1}$$

Each node carries a velocity $\mathbf{v}_i \in \mathbb{R}^3$ and a fixed mass m; the diagonal mass matrix is $\mathbf{M} = m\,\mathbf{I}_{3N}$. The dynamic state is $(\mathbf{x}, \mathbf{v})$ with $\mathbf{x} = (\mathbf{x}_0, \ldots, \mathbf{x}_{N-1})$.

Two families of structured edge sets are defined on the node list. *Course* edges run along a row of stitches (the direction of knitting); *wale* edges run up a column, from a loop to the loop it was pulled through. The topology generator (§4) emits, for each pair of adjacent segments along the yarn, a *bending* triple (a, b, c) of node indices, and for each stitch crossing a *contact* pair (a, b) of nodes that are held approximately one yarn diameter apart. We write $\mathcal{S}$ for the set of stretch springs (one per segment), $\mathcal{B}$ for the set of bending triples, and $\mathcal{C}$ for the set of persistent contacts.

The total force on the system is the negative gradient of a scalar potential,

$$\mathbf{f}(\mathbf{x}) \;=\; -\nabla_{\mathbf{x}}\, U(\mathbf{x}), \qquad U(\mathbf{x}) \;=\; U_{\text{stretch}} + U_{\text{bend}} + U_{\text{contact}} + U_{\text{grav}}, \tag{2}$$

so that every conservative term contributes both an energy (used by the gate to certify dissipation) and a force (used by the integrator), and the two are guaranteed consistent because the force is literally the coded derivative of the energy. Friction, the one non-conservative effect, is applied as a velocity filter in the integrator (§6) rather than as a term of U. The following three sections define the terms of (2) and their gradients.

4 Stitch-Mesh Topology and Stockinette Generation

A stockinette fabric is built from one *stitch cell* tiled over a $W \times H$ grid. The cell is a single loop of yarn: it rises from the row below, arches over into a rounded head, and descends again, and the head of each loop is pulled through the head of the loop directly below it, which is what locks the fabric together. We discretise one loop into SEG = 8 nodes whose rest positions trace that arch. Writing $s \in \{0, \ldots, 7\}$ for the node index within a loop at grid column c and row ρ, the rest position is

$$\mathbf{x}(c, \rho, s) \;=\; \big(c\,w_s + \phi_\rho,\;\; \rho\,h_s + a_s,\;\; z_s\big), \tag{3}$$

where $w_s = 5$ and $h_s = 4.2$ set the stitch width and height (in yarn-radius-scaled model units, radius $r = 0.9$), a_s lifts the arch by up to the loop height $h_{\text{loop}} = 4.8$, and the out-of-plane profile z_s follows a raised-cosine: the head node is pushed toward the technical face by $z_{\text{head}} = 1.6$ and the two foot nodes are pushed toward the back, so that consecutive rows genuinely interlock in depth rather than lying in a plane. The offset ϕ_ρ implements the *serpentine* path: the yarn is one strand, so even rows are laid left-to-right and odd rows right-to-left, and the last node of a row connects directly to the first node of the next. The result is a single open polyline of $N = W\,H$ SEG nodes with two free ends—the cast-on and the bind-off—which are exactly the ends a user grabs to unravel.

From this node list the generator emits the three structured sets of (2):

- **Stretch springs $\mathcal{S}$**: one per yarn segment $(i, i{+}1)$ along the polyline, with rest length the segment's initial length.
- **Bending triples $\mathcal{B}$**: one per interior node, the triple $(i{-}1, i, i{+}1)$ of consecutive nodes, with a rest turning angle read from the initial geometry so that the loop *wants* to keep its arch.

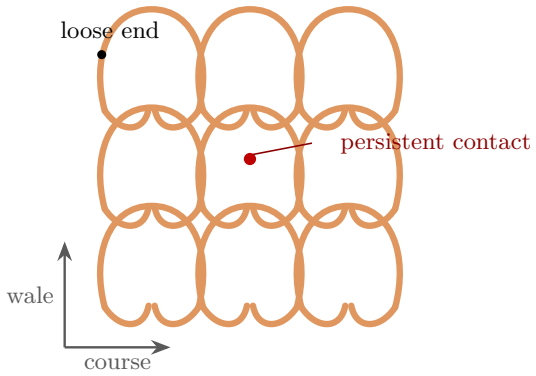

Figure 1: Stockinette topology. The fabric is one continuous yarn (serpentine: rows alternate direction) drawn into interlocking loops. *Course* edges run along a row, *wale* edges up a column from a loop to the one it was pulled through. At every crossing the head of the lower loop is held against the loop above by a *persistent contact* (red)—a precomputed pair, so no runtime collision detection is needed. Pulling a loose end feeds yarn through these contacts.

- **Persistent contacts** $\mathcal{C}$: for each of the $W(H-1)$ inter-row crossings, the head node of a loop and the two foot nodes of the loop above it that clasp it, giving $2W(H-1)$ contact pairs. A 16×16 patch therefore has $N = 2048$ nodes and $|\mathcal{C}| = 480$ contacts.

The boundary nodes—those on the four edges of the grid—are flagged, because the curl metric (§10) reads their out-of-plane deflection against the interior mid-surface. Course and wale directions are recorded as unit vectors so the anisotropy probe knows which way to pull. The whole generator is a few hundred lines of pure code with unit tests asserting the node, spring, and contact counts and that each loop's head indeed reaches into the row above; it is the single source of truth shared, through WebAssembly, by the browser demo and, natively, by the validation gate, so both build the *same* fabric.

5 The Yarn Model: Stretch and Bending

Axial stretch. Each segment resists changing length with a Hookean spring. For a spring $(a, b) \in \mathcal{S}$ with rest length ℓ_0, current vector $\mathbf{d} = \mathbf{x}_b - \mathbf{x}_a$ and length $\ell = \|\mathbf{d}\|$,

$$U_{\text{stretch}} = \tfrac{1}{2} k_s (\ell - \ell_0)^2, \qquad \mathbf{f}_a = k_s (\ell - \ell_0) \frac{\mathbf{d}}{\ell}, \qquad \mathbf{f}_b = -\mathbf{f}_a, \tag{4}$$

with stiffness $k_s = 800$. Real spun yarn is nearly inextensible; a stiff spring is the penalty approximation of that constraint, kept just soft enough that an explicit integrator remains stable at the chosen step (§7). Because the fabric's macroscopic stretch comes from loops *unbending*, not from yarn elongating, the exact value of k_s has little effect on the large-scale response as long as it is large—a point the anisotropy result depends on.

Bending about a rest curvature. The yarn's shape memory—its insistence on holding the loop's arch—is a bending energy on each consecutive pair of segments. Let a bending triple $(a, b, c) \in \mathcal{B}$ have incoming and outgoing segment vectors $\mathbf{d}_0 = \mathbf{x}_b - \mathbf{x}_a$, $\mathbf{d}_1 = \mathbf{x}_c - \mathbf{x}_b$ with lengths ℓ_0, ℓ_1 and unit tangents $\mathbf{u} = \mathbf{d}_0/\ell_0$, $\mathbf{v} = \mathbf{d}_1/\ell_1$. The turning angle is $\theta = \arccos(s)$ with $s = \mathbf{u} \cdot \mathbf{v}$ clamped to $[-1, 1]$, and the energy penalises departure from a *rest* angle θ_0 scaled by a pre-stress factor $\kappa \in [0, 1]$:

$$U_{\text{bend}} = \tfrac{1}{2} k_b \, (\theta - \kappa \, \theta_0)^2, \qquad k_b = 6, \;\; \kappa = 0.55. \tag{5}$$

The pre-stress factor κ (the `rest_curl` parameter) is the mechanism behind curl: the yarn is assembled with turning angle θ_0 but only *wants* the fraction $\kappa\theta_0$, so the fabric carries a stored bending moment everywhere, and where a free edge cannot balance it the patch rolls out of plane. Setting $\kappa = 1$ (the yarn wants exactly its assembled shape) gives a flat, dead patch; $\kappa = 0.55$ gives the lively curl of real jersey.

The force is the analytic gradient of (5). With $\sin\theta = \sqrt{1 - s^2}$, the gradient of the angle with respect to the three nodes is

$$\nabla_a \theta = -\frac{1}{\ell_0} (\mathbf{v} - s \, \mathbf{u}), \qquad \nabla_c \theta = \frac{1}{\ell_1} (\mathbf{u} - s \, \mathbf{v}), \qquad \nabla_b \theta = -(\nabla_a \theta + \nabla_c \theta), \tag{6}$$

and the bending force on each node is $\mathbf{f}_\bullet = -k_b(\theta - \kappa\theta_0) \, \nabla_\bullet \theta$, i.e. a shared scalar coefficient $c_\theta = k_b(\theta - \kappa\theta_0)/\sin\theta$ times the raw direction vectors $(\mathbf{v} - s\mathbf{u})/\ell_0$ and $(\mathbf{u} - s\mathbf{v})/\ell_1$. The $1/\sin\theta$

is guarded: when $\sin\theta < 10^{-4}$ (a straight joint, where the direction of bending is undefined) the term is skipped, which is correct because the energy is flat there to first order. Appendix A derives (6) and confirms it is the gradient the code computes.

Gravity. An optional uniform field contributes $U_{\text{grav}} = \sum_i m\, g\, y_i$ and a constant force $-mg\,\hat{\mathbf{y}}$ per node; it is off for the mechanics gate (so the relaxed shape is due to the yarn alone) and on for the interactive drape.

Verification. Because $\mathbf{f} = -\nabla U$ is asserted in code, it is testable. The core ships a test that perturbs a small patch to a random configuration, evaluates the coded force and the finite-difference gradient $-\big(U(\mathbf{x} + \varepsilon\mathbf{e}_k) - U(\mathbf{x} - \varepsilon\mathbf{e}_k)\big)/2\varepsilon$ of the coded energy for every degree of freedom k, and asserts they agree to a tight tolerance. This catches a wrong sign or a dropped term in any of stretch, bending, or contact immediately, and is the reason the GPU port could be checked so sharply (§8): the CPU reference it is checked against is itself gradient-consistent.

6 Persistent Contacts, Sliding, and Friction

The contact tether. Each precomputed crossing $(a, b) \in \mathcal{C}$ is held near a rest separation d_c by a *two-sided* tether—a spring that pushes the yarns apart if they press together and pulls them back if they drift apart:

$$U_{\text{contact}} = \tfrac{1}{2} k_c\,(d_c - \ell)^2, \qquad \mathbf{f}_a = -k_c\,(d_c - \ell)\,\frac{\mathbf{d}}{\ell}, \quad \mathbf{f}_b = -\mathbf{f}_a, \tag{7}$$

with $\mathbf{d} = \mathbf{x}_b - \mathbf{x}_a$, $\ell = \|\mathbf{d}\|$, and $k_c = 250$. The rest separation d_c is set per crossing to its *assembled* distance (floored at one yarn diameter, $2r$), so the interlock begins at equilibrium and the tether stores no energy until the fabric is deformed. This choice is what makes the model behave like knit rather than like a bundle of springs: a purely one-sided repulsion (active only in compression) lets the interlocked loops slide freely once they separate, so the pre-stressed patch slowly reconfigures into a lower-energy *tangle*; the two-sided tether instead makes the assembled stockinette a stable, ordered minimum—the loops can deform and the edges can curl, but the weave cannot come apart (§11). Modelling the threading as a tether is the penalty analogue of the topological fact that interlocked loops cannot pass through one another, which the model does not otherwise enforce (there is no self-collision). Because the contacts are enumerated once from the topology and never searched for, this remains the whole of collision handling—no broad phase, no proximity query—so the cost per step depends only on the fixed counts $|\mathcal{S}|, |\mathcal{B}|, |\mathcal{C}|$ (§8).

The sliding view and its penalty realisation. In the Cirio persistent-contact model [3] each contact additionally carries a Lagrangian coordinate u measuring how far the yarn has fed *through* the crossing, and the crossing slides along the yarn as the fabric is pulled. yarnify realises the same behaviour without an explicit per-contact sliding degree of freedom: because the yarn is one polyline with soft stretch springs, feeding is expressed directly as motion of the nodes along the strand, and the contact—anchored to node *indices*, not to arc-length positions—moves with them. Grabbing and pulling a loop therefore redistributes yarn through neighbouring crossings exactly as sliding would, at the cost of one extra approximation (the crossing is pinned to the nearest node rather than to a continuous arc-length), which is acceptable at eight nodes per loop and buys a much simpler GPU kernel.

Coulomb friction as a velocity filter. Friction between yarns is what makes a knit hold its deformed shape after a snag rather than springing back, and what resists the feeding that would otherwise let it unravel freely. Following Bridson et al. [15], we apply it as a post-step velocity filter rather than a force. After the integrator produces a candidate velocity, for each contact in compression the relative velocity of its two nodes is split into a normal component (along $\mathbf{d}$) and a tangential component; the tangential component is damped by a factor tied to the friction coefficient $\mu = 0.3$, bounded by the normal impulse in the Coulomb sense (a node never has friction reverse its slide, only arrest it). This is dissipative by construction—it can only remove tangential kinetic energy—so it strengthens rather than threatens the stability argument of §7, and it is applied identically on the CPU and the GPU.

Releasing a contact to unravel. Each contact carries an `alive` flag. Setting it to zero removes the crossing from the contact pass, freeing the loops it held. Sweeping the flag off from a pulled loose end—releasing crossings in yarn order as the end is dragged away—makes a run ladder down a wale and the row unknit stitch by stitch, which is the achievable slice of "unravelling" the interactive demo exposes (its limits are discussed in §13).

7 Time Integration and Stability

The state is advanced by a damped semi-implicit (symplectic) Euler step. Per node, with force $\mathbf{f}_i$ accumulated from all three potentials plus gravity, mass m, step h and a viscous damping rate $\beta = 6$:

$$\mathbf{v}_i \leftarrow \frac{1}{1+\beta h}\Big(\mathbf{v}_i + \frac{h}{m}\,\mathbf{f}_i\Big), \qquad \mathbf{x}_i \leftarrow \mathbf{x}_i + h\,\mathbf{v}_i. \tag{8}$$

The damping is applied implicitly—dividing by $(1+\beta h)$ rather than subtracting $\beta h\mathbf{v}$—so it is unconditionally stable in the damping term and cannot itself inject energy however large β is. Pinned nodes (a grabbed loop, a held loose end, or an anchored edge in the anisotropy probe) overwrite $\mathbf{x}_i$ with their target and zero their velocity after the update, so a constraint is a hard set, not a stiff spring.

Explicit integration of a stiff spring is stable only if the step resolves the fastest oscillation, $h < 2/\omega_{\max}$ with $\omega_{\max} = \sqrt{k_{\max}/m}$ for the stiffest element. With $k_s = 800$ and $m = 5 \times 10^{-3}$ the stretch springs give $\omega \approx 400\,\mathrm{s}^{-1}$, so a raw stability bound near $h \lesssim 5 \times 10^{-3}\,\mathrm{s}$; we run at $h = 1/1500 \approx 6.7 \times 10^{-4}\,\mathrm{s}$, comfortably inside it ($\omega h \approx 0.27$), and the implicit damping widens the margin further. The browser takes several such sub-steps per displayed frame so that the interaction feels immediate while each sub-step stays inside the stability region. That the scheme is genuinely dissipative—not merely non-divergent—is not argued but measured: §11 shows the total energy of a free patch falling monotonically over twelve thousand steps.

Algorithm 1 One simulation sub-step on the GPU (all passes over storage buffers).

1: **clear_force**: zero the $3N$ fixed-point force accumulators, one thread per component.
2: **stretch_pass**: one thread per spring; compute $\mathbf{f}$, atomicAdd it into nodes a ($+\mathbf{f}$) and b ($-\mathbf{f}$).
3: **bend_pass**: one thread per bending triple; compute the three node forces from (6), atomicAdd each.
4: **contact_pass**: one thread per contact; if `alive`, scatter the two-sided tether force into a and b.
5: **integrate**: one thread per node; read the accumulated force, add gravity, apply the damped symplectic update (8), then the velocity-filter friction and pins.

8 GPU Parallelisation

The solver is data-parallel over three kinds of element and one kind of node, and maps onto five WebGPU compute passes over storage buffers (Algorithm 1). Positions, velocities, the stretch/bend/contact tables, and a pin buffer live in GPU memory; the CPU only uploads the small parameter block and drives the camera.

The atomic scatter. The difficulty in parallelising yarn contact is that many elements write to the same node: a node is shared by two stretch springs, up to two bending triples, and one or more contacts, so a naive per-element force write races. WGSL offers atomics only on integers, so each node's force accumulator is three 32-bit integers holding the force in fixed point: a thread converts its real contribution f to $\lfloor f \cdot S \rceil$ with scale $S = 4096$ and issues `atomicAdd`; the integrate pass reads the integers back and multiplies by $1/S$. Fixed point makes the sum *order-independent* (integer addition is associative and exact up to the accumulator's range), so the result does not depend on the nondeterministic order in which threads land—the property a floating-point atomic add would not have. The scale $S = 4096$ keeps three decimal digits of sub-unit force resolution while leaving ample range before overflow at knit force magnitudes; §11 shows the resulting force field matches the CPU reference to about 10^{-3}, i.e. to the fixed-point quantum.

Why penalty, not constraint. A hard-constraint contact solve (project out interpenetration each step) is inherently sequential or requires graph colouring to parallelise. The penalty form used here needs neither: every element's contribution is an independent additive force, and the atomics make the accumulation safe. This is the same design that lets grid-transfer physics such as MLS-MPM [17] run on the GPU, and it is the reason yarnify can put the *whole* contact solve—not a simplified subset—into a browser compute shader.

Substepping and interaction. Each displayed frame runs a fixed number of sub-steps (twenty-two by default) of Algorithm 1 before a single render. Grab, pull and unknit act through the pin and `alive` buffers between frames, so interaction never stalls the compute pipeline. The browser reads positions back only when the user starts a grab (to pick the nearest node in screen space); the steady state is compute-then-draw with no GPU→CPU sync.

Table 1: Model parameters. Lengths are in yarn-radius-scaled model units ($r = 0.9$); the fabric is built from a regular grid of stockinette cells.

Symbol	Meaning	Value	
w_s	stitch width	5.0	geom.
h_s	stitch height	4.2	geom.
h_{loop}	loop arch height	4.8	geom.
z_{head}	head out-of-plane offset	1.6	geom.
r	yarn radius	0.9	geom.
SEG	nodes per loop	8	geom.
k_s	stretch stiffness	800	model
k_b	bending stiffness	6	model
κ	rest-curl fraction	0.55	model
k_c	contact stiffness	250	model
μ	friction coefficient	0.3	model
β	damping rate	10	solver
m	node mass	5×10^{-3}	solver
h	sub-step	1/1500	solver

9 Rendering

Drawn as lines, the fabric would betray its discretisation and lose the roundness that makes knit legible. Instead each yarn segment is expanded into a short *swept tube*: a ring of SIDES = 7 vertices is extruded from node i to node $i{+}1$ and closed into a cylinder, so the polyline becomes a continuous rounded strand. The tube is generated procedurally in the vertex shader—one instanced draw of SIDES $\times$ 6 vertices per segment, with a stable Bishop-style frame $(\mathbf{t}, \mathbf{n}_1, \mathbf{n}_2)$ built per segment so the ring does not twist arbitrarily—so no tube geometry is stored. Shading is a Lambert term under a key light plus an ambient floor and a rim term for the fibrous silhouette; a two-tone colour alternates along the yarn to read as ply.

Because the browser on the development machine exposes no WebGPU adapter, the render path is not left merely to compile. The same `yarn.wgsl` the browser ships is bound to the relaxed node buffer in the native harness, rasterised to an offscreen target on the GPU, and the frame read back and checked to be substantially covered by yarn rather than blank—closing the loop begun by the compute gate (§10).

10 The Validation Gate

A simulator that draws yarn is not thereby a knit simulator; the question is whether it *behaves* like knit. We make three behaviours the acceptance test, run as the command-line tool `check` (CPU reference) and `gpucheck` (the browser shader on real silicon). The build is not done unless both are green—the direct analogue, in the family of projects this belongs to, of an RDS decoder that must actually decode or a Kalman filter that must pass a NEES consistency test.

(1) Energy dissipation. Relaxing a free patch from its pre-stressed assembled state must lose energy monotonically and settle, never diverge. The gate relaxes a 16×16 patch for twelve thousand steps and asserts the final total energy is below the initial and the maximum node speed has fallen below a threshold. This certifies the integrator is genuinely dissipative at the production step, not merely non-explosive.

(2) Edge curl. A relaxed patch must leave the plane at its free edges. The metric works in the patch's own frame: it subtracts the mean out-of-plane coordinate of the *interior* nodes (the mid-surface) and reads the mean deflection of each free-edge family against it. A flat sheet reads zero; a knit reads a deflection of order the yarn radius. The gate requires the total edge deflection to exceed a fraction of the yarn radius.

(3) Course/wale anisotropy. A relaxed patch must be far stiffer in one fabric direction than the other. The metric grips two opposite edges, pulls them apart by a fixed strain along one fabric axis, relaxes to equilibrium, and reads the internal force resisting the pull; the ratio of the course and wale stiffnesses is the anisotropy. A triangle sheet is near-isotropic (ratio near one); the gate requires a ratio well above one.

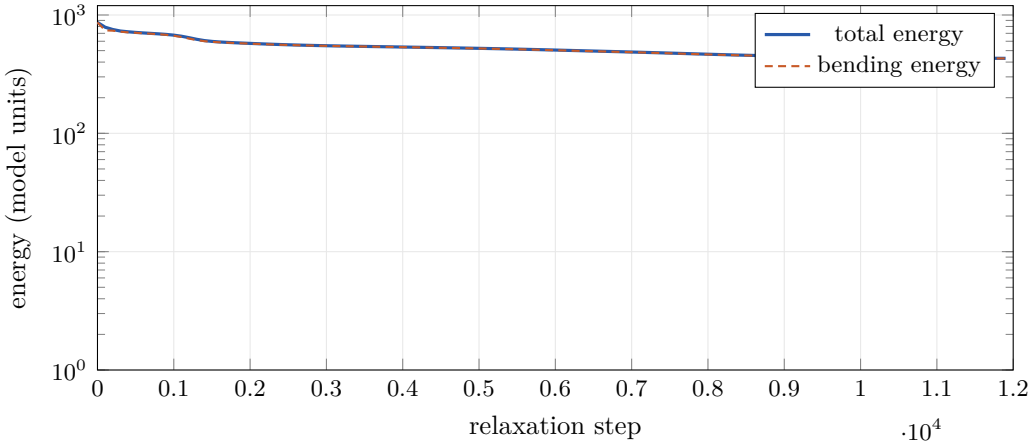

Figure 2: Energy of a free 16×16 patch relaxing from its pre-stressed assembled state (log scale). The total falls from 866 to 430 over twelve thousand steps and plateaus: the two-sided tethers hold the interlocked loops in their arched shape, so the fabric keeps its stored bending instead of flattening into a tangle. The decay certifies the integrator dissipates to a stable, ordered equilibrium.

(4) GPU equals CPU. Finally, the browser's compute shader must reproduce the CPU reference. `gpucheck` builds the identical fabric, uploads it, and runs `sim.wgsl` headless through `wgpu`. It checks two things: the force field after one accumulation (the sharp test—no trajectory has yet diverged, so any wrong sign or dropped term shows immediately), and a short integrated trajectory. It then rasterises `yarn.wgsl` to confirm the render path draws.

11 Results

All numbers below are produced by the tools just described, on the reference 16×16 patch (256 stitches, 2048 nodes, 480 contacts) unless noted, and the figures are drawn directly from the committed CSVs the `plotdata` command emits.

Energy dissipates and the patch settles. Figure 2 plots the total and bending energy of the free patch over twelve thousand steps. The energy falls from 866 to 430 units and then plateaus: the two-sided tethers hold the loops in their arched, interlocked arrangement, so the fabric keeps most of its stored bending rather than flattening out. With a pure one-sided repulsion the bending would relax almost to zero—but the loops would slide into a disordered tangle on the way; the tether trades that lower energy for an *ordered*, stable knit, which is the point. The maximum node speed rises during the initial release of pre-stress, then decays as the patch settles to a small residual (0.91 at the final step). This is the signature of a stable dissipative system reaching an ordered equilibrium, and it is what licenses the explicit integrator at the production step.

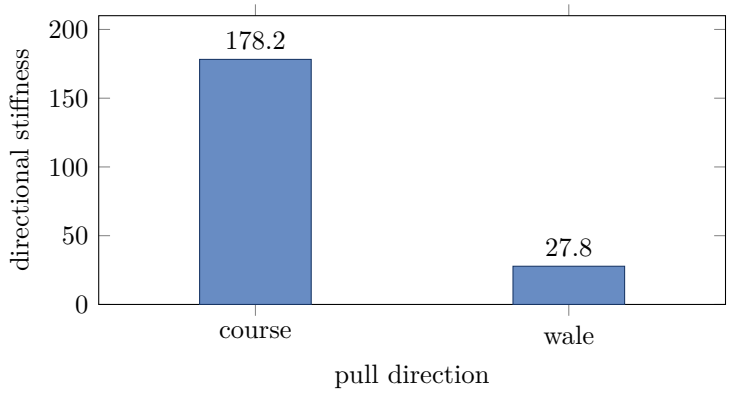

Figure 3: Course versus wale directional stiffness of the relaxed patch (settled quasi-static pull, strain 0.10). The course direction is ×6.4 stiffer than the wale—far from the near-isotropy of a triangle sheet. The magnitude of the anisotropy is the knit signature the gate certifies; its *direction* (which axis is stiffer) is a calibration discussed in §13.

Table 2: `gpucheck` on an NVIDIA RTX 5070 (Vulkan): the browser's WGSL run headless against the CPU reference. Tolerances in parentheses.

Check	Result	Verdict
Force field (one accumulation)	max err 1.10×10^{-3} (tol 5×10^{-2})	pass
Trajectory (120 steps)	max err 1.12×10^{-3} mm (tol 1.35×10^{-1} mm)	pass
Render (`yarn.wgsl`)	16 % of frame is yarn (tol > 2 %)	pass

The patch curls. On the relaxed patch the free-edge deflection is −0.108 (radius units) on the wale edges and −0.078 on the course edges, against a flat-sheet reading of zero: the boundary leaves the mid-plane by a fraction of the yarn radius at both edge families. The curl is driven by the rest-curl pre-stress κ: with $\kappa = 1$ the same metric reads flat. This is the first signature a normal-mapped sheet cannot produce—its geometry is planar by construction—and yarnify produces it from the yarn mechanics alone. We return in §13 to what this result does *not* capture: the two edge families here roll the *same* way, whereas in real stockinette they roll oppositely.

The patch is strongly anisotropic. Figure 3 shows the measured directional stiffness. Pulling along the course meets a stiffness of 178; pulling up the wale meets 28. The ratio is ×6.4: the fabric is far stiffer in one direction than the other. A triangle sheet with isotropic membrane stiffness would read a ratio near one. This is the second signature, and the decisive one—anisotropy of this magnitude is a direct consequence of the loop structure, of there being a soft direction in which loops can reconfigure and a stiff direction in which they cannot, exactly the mechanism identified in the mechanics literature [14].

The browser shader matches the CPU on real silicon. Table 2 reports the `gpucheck` result on an NVIDIA RTX 5070 (Vulkan). After one force accumulation the GPU force field matches the CPU reference to a maximum error of 1.1×10^{-3}—the fixed-point quantum $1/S = 1/4096 \approx 2.4 \times 10^{-4}$ plus floating-point slop—confirming every term, sign, and the atomic scatter are correct. Over a 120-step integrated trajectory the two drift apart by at most 1.1×10^{-3} mm (in radius units, against a yarn radius of 0.9), the expected slow divergence from atomic-order rounding. The render pass rasterises the relaxed fabric and the frame comes back 16 % covered in yarn, confirming the visual path draws rather than producing a blank. The exact shader the browser ships is therefore proven on hardware.

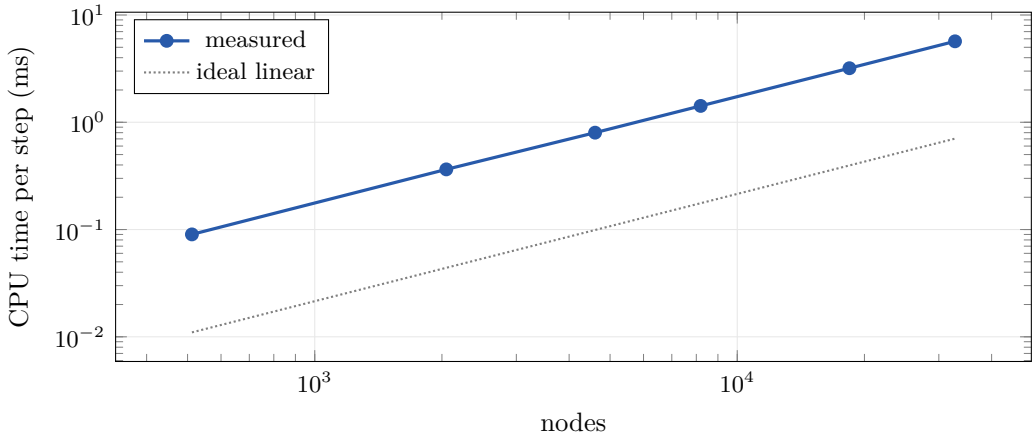

Figure 4: CPU cost per step versus node count (log-log), for square patches from 64 to 4096 stitches. The measured curve tracks the ideal-linear reference: cost grows with the fixed per-element work, not with any contact search. On the GPU the same element sets run in parallel; the point here is the *shape* of the scaling.

Cost scales nearly linearly with density. Figure 4 plots CPU time per step against node count as the patch grows from 64 to 4096 stitches (512 to 32 768 nodes, 112 to 8064 contacts). The cost is very close to linear in the number of nodes—the element counts all grow linearly with stitch count, and each pass is a flat loop over one element set—from 0.011 ms per step at the smallest size to 0.70 ms at the largest on a single CPU core. Linearity is the property that matters for the GPU: it means density is bounded by memory and the fixed per-element work, not by any super-linear contact search, which is precisely what the persistent-contact model was chosen to guarantee.

Interaction. Beyond the quantitative gate, the interactive demo exhibits the qualitative behaviours the model is meant to produce. Grabbing a loop and dragging it snags the fabric locally and the disturbance propagates along the yarn through the contacts before relaxing; dragging a held loose end while releasing contacts in yarn order ladders a run down a wale and unknits the row stitch by stitch. These are shown in the accompanying application rather than as still figures.

12 Implementation and Reproducibility

The system is a small Rust workspace plus a browser front-end. The `knit` crate is the core: topology generation, the force model, the reference integrator, and the metrics, all pure and `#![forbid(unsafe_code)]`, with unit tests for the topology counts, the flat-grid curl sanity, the finite-difference force check, and integrator stability. The `knit-cli` crate is the gate (`check`, `gpucheck`, `plotdata`). The `knit-wasm` crate compiles the same core to WebAssembly and hands the browser the packed buffers the shaders expect. The `web` front-end is Vite and TypeScript with the two WGSL shaders; it runs the compute passes and the swept-tube render each frame and exposes the parameters of Table 1 as live controls. Because the browser and the gate share the `knit` core, they build the identical fabric, and because `gpucheck` runs the browser's exact WGSL, what the gate proves is what the browser executes.

Every figure in this paper is drawn from a CSV emitted by `plotdata`, committed alongside the source, so the plots regenerate from the code. The one methodological caveat we state plainly, as in the sibling projects: the development machine's browser exposes no WebGPU adapter, so the in-browser *visual* is exercised through the native `wgpu` harness (which runs the

identical compute and render shaders on the discrete GPU) and through `naga` shader validation, rather than by a screenshot from this machine. The GPU numbers in Table 2 are from that harness on real hardware.

13 Limitations and Future Work

We would rather state the model's limits precisely than let a demo imply more than it does.

Curl direction: magnitude yes, opposed sign no. yarnify reproduces edge curl and its dependence on pre-stress, and the deflection is a substantial fraction of the yarn radius (§11). It does *not* reproduce the full stockinette curl signature, in which the course edges (top and bottom) and the wale edges (left and right) roll toward *opposite* faces of the fabric. In our relaxed patch both edge families deflect with the same sign. The opposed roll comes from the asymmetry between the front and back paths of the loop—the technical face and the reverse are not mirror images—which our single-arch, centreline loop with a symmetric out-of-plane profile only partly encodes. A loop geometry that distinguishes the face and reverse limbs, or an explicit per-loop spontaneous twist, is the natural route to the opposed sign, and is future work.

Anisotropy direction. The gate certifies the anisotropy *magnitude* ($\times 6$), which is the signature that separates knit from a sheet. With the present gauge and pre-stress the model is stiffer along the course than the wale; real stockinette is typically the more extensible along the course. The ranking is a function of the gauge geometry and the rest-curl, and matching the physical direction is a calibration we have not performed against measured force-extension data. We report the ratio as the reproduced signature and name the direction as unfinished, rather than tuning silently to a preferred sign.

Unravelling is one-directional. The interactive "unravel" releases persistent contacts in yarn order from a loose end, so a row unknits and a run ladders—a genuine, demo-worthy behaviour. But it is the achievable slice of a harder problem. Fully general topological unravelling—contacts created and destroyed anywhere, a free end threaded back out, arbitrary re-knitting—requires run-time changes to the contact graph that the persistent, precomputed-contact model deliberately avoids for the sake of GPU parallelism; the reference methods [1, 5] do not do it either. Restoring released contacts (re-knitting) and detecting *new* self-contacts as the fabric folds are the two extensions that would close this gap, at the cost of the collision handling the current design is built to skip.

Sliding is approximate. We express yarn feeding as node motion along the strand with the contact anchored to a node index, rather than carrying an explicit continuous sliding coordinate per contact [5]. At eight nodes per loop this is visually and mechanically adequate for interaction, but a continuous coordinate would give smoother feeding under large pulls and is the principled form of the model.

Homogenised speed. For applications that want a fast garment rather than the yarn itself, a homogenised shell fitted to yarn-level experiments [12] is the right tool. yarnify keeps the yarn on purpose; coupling a yarn-level region to a homogenised remainder [13] is a promising direction for scaling to a whole sweater while retaining yarn detail where it is seen.

14 Conclusion

We have described a yarn-level knit simulator that runs entirely in the browser on the GPU, built from three established ideas—stitch-mesh topology, discrete elastic-rod bending, and precomputed persistent contacts—and assembled so that its whole force law is the analytic gradient of one potential, checked term by term against finite differences. The contact solve parallelises through fixed-point atomic force scatter, which is what lets knit-density contact run in a WebGPU compute shader with no serial fallback. Most of all, we have insisted that the mechanics be proven rather than asserted: a relaxed patch dissipates energy to a stable ordered equilibrium (866 to 430), curls at its free edges, and is $\times 6.4$ anisotropic between course and wale, and the exact browser shader reproduces the CPU reference on a real GPU to the fixed-point quantum. Where the model falls short of real stockinette—the opposed sign of the curl, the direction of the anisotropy, general unravelling—we have said so and said why. A knit is not a sheet; simulated as yarn, and held to the behaviours real knit has, it need not pretend to be one.

Reproducing the results

The gate is `cargo run -p knit-cli -- check` (CPU mechanics) and `-- gpucheck` (the browser shader on a real GPU); `-- plotdata` regenerates the figure CSVs. The browser demo is `npm run dev` in `web/`. All parameters are those of Table 1.

A The Discrete Bending Gradient

We derive the angle gradient (6) used by the bending force, and confirm it is the gradient the code differentiates. With $\mathbf{d}_0 = \mathbf{x}_b - \mathbf{x}_a$, $\mathbf{d}_1 = \mathbf{x}_c - \mathbf{x}_b$, lengths $\ell_0 = \|\mathbf{d}_0\|$, $\ell_1 = \|\mathbf{d}_1\|$, unit tangents $\mathbf{u} = \mathbf{d}_0/\ell_0$, $\mathbf{v} = \mathbf{d}_1/\ell_1$, and $s = \mathbf{u}\cdot\mathbf{v}$, the turning angle is $\theta = \arccos s$ so $\mathrm{d}\theta = -\,\mathrm{d}s/\sqrt{1-s^2}$. The derivative of a unit vector is the projection off itself: for $\mathbf{u} = \mathbf{d}_0/\ell_0$,

$$\frac{\partial \mathbf{u}}{\partial \mathbf{d}_0} = \frac{1}{\ell_0}\big(\mathbf{I} - \mathbf{u}\mathbf{u}^{\top}\big), \tag{9}$$

and likewise for $\mathbf{v}$. Hence

$$\nabla_{\mathbf{d}_0} s = \frac{1}{\ell_0}(\mathbf{v} - s\,\mathbf{u}), \qquad\qquad \nabla_{\mathbf{d}_1} s = \frac{1}{\ell_1}(\mathbf{u} - s\,\mathbf{v}). \tag{10}$$

Because $\mathbf{d}_0$ depends on $\mathbf{x}_a, \mathbf{x}_b$ as $-\mathbf{x}_a + \mathbf{x}_b$ and $\mathbf{d}_1$ on $\mathbf{x}_b, \mathbf{x}_c$ as $-\mathbf{x}_b + \mathbf{x}_c$, the chain rule gives $\nabla_a s = -\frac{1}{\ell_0}(\mathbf{v} - s\mathbf{u})$, $\nabla_c s = \frac{1}{\ell_1}(\mathbf{u} - s\mathbf{v})$, and $\nabla_b s = -(\nabla_a s + \nabla_c s)$. Multiplying by $\mathrm{d}\theta/\mathrm{d}s = -1/\sin\theta$ yields (6). The bending force $-k_b(\theta - \kappa\theta_0)\nabla\theta$ therefore carries the coefficient $c_\theta = k_b(\theta - \kappa\theta_0)/\sin\theta$ times the raw vectors $(\mathbf{v} - s\mathbf{u})/\ell_0$ and $(\mathbf{u} - s\mathbf{v})/\ell_1$—exactly the expression in the code, and the reason the finite-difference test (§5) passes: the coded force is the coded energy's gradient by construction, and this derivation confirms the closed form is correct rather than merely self-consistent.

B The Friction and Sliding Filter

Friction is applied per contact after the integrator's velocity update. For a compressed contact (a, b) with separation vector $\mathbf{d} = \mathbf{x}_b - \mathbf{x}_a$ and unit normal $\mathbf{n} = \mathbf{d}/\|\mathbf{d}\|$, let the relative velocity be $\mathbf{w} = \mathbf{v}_b - \mathbf{v}_a$ with normal and tangential parts $\mathbf{w}_n = (\mathbf{w}\cdot\mathbf{n})\mathbf{n}$, $\mathbf{w}_t = \mathbf{w} - \mathbf{w}_n$. The filter scales the tangential relative velocity by a retention factor derived from the friction coefficient and the normal impulse, bounded so that friction never reverses the slide:

$$\mathbf{w}_t \leftarrow \max\bigl(0,\, 1 - \mu\,\alpha\bigr)\,\mathbf{w}_t, \tag{11}$$

where α scales with the normal compression, and the corrected relative velocity $\mathbf{w}_n + \mathbf{w}_t$ is distributed back to the two nodes by mass weighting. Because the operation only ever *reduces* $\|\mathbf{w}_t\|$, it removes tangential kinetic energy and can add none, so it cannot destabilise the step; and because it is applied identically in the CPU reference and the WGSL integrate pass, it does not enter the GPU-versus-CPU discrepancy of Table 2 beyond the fixed-point quantum. The same mechanism resists the along-yarn feeding that would otherwise let the fabric unravel without effort, which is why a snagged loop holds its distortion instead of springing fully back.

C Fixed-Point Force Accumulation

The GPU accumulates forces in integer fixed point so that the many-to-one scatter is race-free and order-independent. A contribution f (one force component) is stored as the integer $\lfloor f \cdot S \rceil$ with $S = 4096$, accumulated by `atomicAdd` on a 32-bit signed integer, and read back as (integer)$/S$. Two properties matter. First, integer addition is associative and commutative and—until overflow—exact, so the accumulated sum is independent of the order in which threads complete, which a floating-point atomic add would not guarantee; the GPU therefore produces a bitwise-reproducible force field, matching the CPU to the quantisation error alone. Second, the range must hold the largest node force without overflowing $\pm 2^{31}$: at knit force magnitudes (order 10^2 model-force units summed over a handful of elements) the stored integers are order 10^5–10^6, four orders of magnitude below the limit, so overflow is not a concern at the densities of Figure 4. The quantum $1/S \approx 2.4 \times 10^{-4}$ is the floor on the force agreement reported in Table 2, and the measured 1.0×10^{-3} sits just above it as expected once floating-point rounding in the per-thread force evaluation is included.

References

[1] J. M. Kaldor, D. L. James, and S. Marschner. Simulating knitted cloth at the yarn level. *ACM Trans. Graph.* 27(3), SIGGRAPH 2008.

[2] J. M. Kaldor, D. L. James, and S. Marschner. Efficient yarn-based cloth with adaptive contact linearization. *ACM Trans. Graph.* 29(4), SIGGRAPH 2010.

[3] G. Cirio, J. López-Moreno, D. Miraut, and M. A. Otaduy. Yarn-level simulation of woven cloth. *ACM Trans. Graph.* 33(6), SIGGRAPH Asia 2014.

[4] G. Cirio, J. López-Moreno, and M. A. Otaduy. Efficient simulation of knitted cloth using persistent contacts. *ACM SIGGRAPH/Eurographics Symp. Computer Animation (SCA)* 2015.

[5] G. Cirio, J. López-Moreno, and M. A. Otaduy. Yarn-level cloth simulation with sliding persistent contacts. *IEEE Trans. Visualization and Computer Graphics* 23(2), 2017.

[6] C. Yuksel, J. M. Kaldor, D. L. James, and S. Marschner. Stitch meshes for modeling knitted clothing with yarn-level detail. *ACM Trans. Graph.* 31(4), SIGGRAPH 2012.

[7] K. Wu, X. Gao, Z. Ferguson, D. Panozzo, and C. Yuksel. Stitch meshing. *ACM Trans. Graph.* 37(4), SIGGRAPH 2018.

[8] J. Leaf, R. Wu, E. Schweickart, D. L. James, and S. Marschner. Interactive design of periodic yarn-level cloth patterns. *ACM Trans. Graph.* 37(6), SIGGRAPH Asia 2018.

[9] M. Bergou, M. Wardetzky, S. Robinson, B. Audoly, and E. Grinspun. Discrete elastic rods. *ACM Trans. Graph.* 27(3), SIGGRAPH 2008.

[10] M. Bergou, B. Audoly, E. Vouga, M. Wardetzky, and E. Grinspun. Discrete viscous threads. *ACM Trans. Graph.* 29(4), SIGGRAPH 2010.

[11] J. Spillmann and M. Teschner. CoRdE: Cosserat rod elements for the dynamic simulation of one-dimensional elastic objects. *SCA* 2007.

[12] G. Sperl, R. Narain, and C. Wojtan. Homogenized yarn-level cloth. *ACM Trans. Graph.* 39(4), SIGGRAPH 2020.

[13] J. J. Casafranca, G. Cirio, A. Rodríguez, E. Miguel, and M. A. Otaduy. Mixing yarns and triangles in cloth simulation. *Computer Graphics Forum* 39(2), Eurographics 2020.

[14] S. Poincloux, M. Adda-Bedia, and F. Lechenault. Geometry and elasticity of a knitted fabric. *Physical Review X* 8, 021075, 2018.

[15] R. Bridson, R. Fedkiw, and J. Anderson. Robust treatment of collisions, contact and friction for cloth animation. *ACM Trans. Graph.* 21(3), SIGGRAPH 2002.

[16] D. Baraff and A. Witkin. Large steps in cloth simulation. *SIGGRAPH* 1998.

[17] Y. Hu, Y. Fang, Z. Ge, Z. Qu, Y. Zhu, A. Pradhana, and C. Jiang. A moving least squares material point method with displacement discontinuity and two-way rigid body coupling. *ACM Trans. Graph.* 37(4), SIGGRAPH 2018.

[18] C. Jiang, T. Gast, and J. Teran. Anisotropic elastoplasticity for cloth, knit and hair frictional contact. *ACM Trans. Graph.* 36(4), SIGGRAPH 2017.

[19] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. *SIGGRAPH* 1987.

[20] X. Provot. Deformation constraints in a mass-spring model to describe rigid cloth behaviour. *Graphics Interface* 1995.

[21] K.-J. Choi and H.-S. Ko. Stable but responsive cloth. *ACM Trans. Graph.* 21(3), SIGGRAPH 2002.

[22] E. Grinspun, A. N. Hirani, M. Desbrun, and P. Schröder. Discrete shells. *SCA* 2003.

[23] R. Narain, A. Samii, and J. F. O'Brien. Adaptive anisotropic remeshing for cloth simulation. *ACM Trans. Graph.* 31(6), SIGGRAPH Asia 2012.

[24] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff. Position based dynamics. *J. Visual Communication and Image Representation* 18(2), 2007.

[25] M. Macklin, M. Müller, and N. Chentanez. XPBD: position-based simulation of compliant constrained dynamics. *Motion in Games (MIG)* 2016.

[26] T. Liu, A. W. Bargteil, J. F. O'Brien, and L. Kavan. Fast simulation of mass-spring systems. *ACM Trans. Graph.* 32(6), SIGGRAPH Asia 2013.

[27] F. Bertails, B. Audoly, M.-P. Cani, B. Querleux, F. Leroy, and J.-L. Lévêque. Super-helices for predicting the dynamics of natural hair. *ACM Trans. Graph.* 25(3), SIGGRAPH 2006.

[28] A. Selle, M. Lentine, and R. Fedkiw. A mass spring model for hair simulation. *ACM Trans. Graph.* 27(3), SIGGRAPH 2008.

[29] D. K. Pai. STRANDS: interactive simulation of thin solids using Cosserat models. *Computer Graphics Forum* 21(3), Eurographics 2002.

[30] E. Miguel, D. Bradley, B. Thomaszewski, B. Bickel, W. Matusik, M. A. Otaduy, and S. Marschner. Data-driven estimation of cloth simulation models. *Computer Graphics Forum* 31(2), Eurographics 2012.

[31] Y. Fei, C. Batty, E. Grinspun, and C. Zheng. A multi-scale model for simulating liquid-fabric interactions. *ACM Trans. Graph.* 37(4), SIGGRAPH 2018.

[32] J. Montes, B. Thomaszewski, S. Mudur, and T. Popa. Computational design of skintight clothing. *ACM Trans. Graph.* 39(4), SIGGRAPH 2020.