Supplementary Document: Knit Sketching: from Cut & Sew Patterns to Machine-Knit Garments

ALEXANDRE KASPAR, KUI WU, YIYUE LUO, and LIANE MAKATURA, MIT CSAIL, USA
WOJCIECH MATUSIK, MIT CSAIL, USA

SUMMARY OF CONTENT

This supplementary material provides some implementation details as well as additional results and comparisons that would not fit in the main document.

The first few sections cover additional *technical* aspects. Sec. 1 describes our validity checks and the user feedback we provide. Sec. 2 introduces the additional notion of curvature as well as a measure of time stretch. Sec. 3 describes our topological opening procedure between the time and region computations. Sec. 4 provides additional details about the graph computations including the user threshold $\Delta t_{\min}$ for making a typical region graph more ideal. Sec. 5 details the implementation of our stitch sampling optimizations. Sec. 6 presents our geodesic distance computation, and the acceleration scheme we use to make it computationally affordable. Sec. 7 describes the issue of alignment underlying our stitch graph sampling strategy. Sec. 8 goes over the additional layout representations we use to allow mixed flat/tubular scheduling.

The second part discusses scalability and controlability within our system. Sec. 9 goes over the complexity of our results, their corresponding runtimes, and further provides convergence insights. Sec. 10 describes different aspects of the garment finish and provides visual comparisons showcasing lower-level user controls.

The remaining of the document focuses on providing additional results in larger sizes to allow proper inspection and interpretation. It starts with detailed views of each of our results including the inputs and intermediate visualizations, as well as user stitch programs. It ends by showing the evolution of some of the results illustrating their iterative nature, which our system enables.

1 VALIDITY OF TIME FUNCTION

The continuous time function is used to decompose the final garment into a set of simple knittable regions. After its computation, we provide two different types of feedback: 1) checks for the feasibility of a region decomposition and 2) warnings when the direction field changes too fast locally, as well as when the local *time stretch* becomes large (see Sec. 2 for its definition).

Feasible Region Decomposition. The main requirement is that local time extrema do not occur at vertices that are strictly inside the domain of a chart. This restriction is similar to that of Narayanan et al. [2018]. Our system further allows time extrema on chart boundaries that are not manifold boundaries (i.e., *closed* cast-on and cast-off seam locations), and which get automatically transformed into actual manifold boundaries via topological opening (see Sec. 3). To verify the time requirement, we compute the set of local time extrema at vertices in our sample mesh. Our interpolation scheme guarantees that point-wise extrema can only occur at mesh vertices.

Knittability. While those validity checks provide useful feedback, they are in no way sufficient to ensure that we end up with a "knittable" result. This is because we should also take into account the problem of scheduling the stitch graph which gets sampled a posteriori. That problem is much more involved, and we do not provide any guarantees. Instead we rely on a best-effort strategy which can unfortunately fail in some scenarios involving complex flat structure interactions.

Intuitively, the mixing of flat with tubular structures can eventually represent any form of 2-manifold surface, and while this leads to obvious scheduling scalability issues as discussed in Sec. 8, it also makes it hard to provide guarantees w.r.t. to knittability without restricting the design space.

2 CURVATURE AND TIME

We investigate the addition of a *curvature* term $\kappa = \|\nabla t\|$ as part of our time function decomposition. The decomposition presented in the main paper together with the integration scheme both assume that the sketch charts are contained in planes that are bound together. However, our time isoline constraints typically induce local curvature and thus lead us to violate such assumption.

We here consider an extension of the integration scheme that includes the notion of curvature κ as the magnitude of the local time gradient of t. Our time integration update stays the same

$$t(v) \leftarrow \frac{1}{|\mathcal{L}(v)|} \sum_{s \in \mathcal{L}(v)} \frac{1}{|\mathcal{N}(s)|} \sum_{s_\mathcal{N} \in \mathcal{N}(s)} [t(s_\mathcal{N}) + dt(s_\mathcal{N} \rightarrow s)] \quad (1)$$

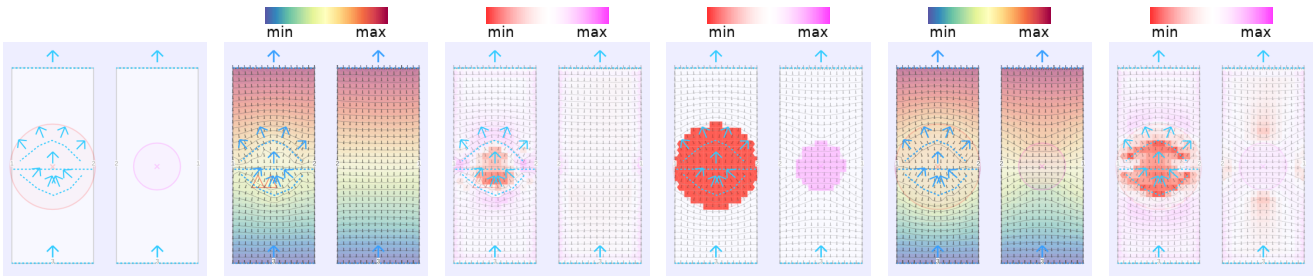
whereas the local time difference $dt(\cdot \rightarrow \cdot)$ now includes the local magnitude $\kappa(\cdot)$ to scale the direction as

$$dt(s_\mathcal{N} \rightarrow s) = \frac{1}{2}[\kappa(s_\mathcal{N})\phi(s_\mathcal{N}) + \kappa(s)\phi(s)] \cdot [p(s) - p(s_\mathcal{N})]. \quad (2)$$

In practice, our system tends to work and converge in a stable way without requiring any curvature information (i.e., $\kappa = 1$), as long as the user time constraints are not contradicting and do not induce large curvature.

Specifically, close-by isoline constraints can induce large local curvature, and those specific cases tend to make our integration unstable in the region of induced high curvature as shown in Fig. 1. In such cases, specifying the local curvature $\kappa(\cdot)$ becomes necessary to get a proper time function without local time extrema in the interior of the sketch domains.

Authors' addresses: Alexandre Kaspar, akaspar@mit.edu; Kui Wu, kuiwu@mit.edu; Yiyue Luo, yiyueluo@mit.edu; Liane Makatura, makatura@mit.edu; Wojciech Matusik, wojciech@csail.mit.edu, Massachusetts Institute of Technology, 32 Vassar Street, Cambridge, Massachusetts, 02139, USA..



(a) *Annotations* **(b)** *Time,* $\kappa = 1$ **(c)** *Time stretch,* $\kappa = 1$ **(d)** *User-defined* κ_u **(e)** *Time,* κ_u **(f)** *Time stretch,* κ_u

Fig. 1. *Notion of time stretch and its correspondence with the local curvature in a case where it is needed for proper time convergence. From left to right: (a) the closed tubular rectangle with its full annotations (time isolines + curvature), (b) the time result without user curvature, which has invalid local time extrema in the center-left section (triangular warning signs); (c) the corresponding time stretch (red when* $\kappa < 1$*, pink when* $\kappa > 1$*); (d) the user-defined curvature; (e) the time result using the user curvature converges properly without local time extrema in the curvature region; (f) the time stretch is similar although more pronounced.*

One way to visualize some form of *induced* curvature from the time is by using what we call the *time stretch*, which we provide as a visualization layer. It appears to be helpful in selecting where to introduce curvature when needed. Computationally speaking, we define it as

$$ts(v) = 2 \times \frac{\sum_{s \in \mathcal{N}(v)} |t(s) - t(v)|}{|\mathcal{N}(v)|}, \tag{3}$$

where $\mathcal{N}(v) = \bigcup_{s \in \mathcal{L}(v)} \mathcal{N}(s)$ is the union set of the sample neighbors from each sample image s of vertex v.

Intuitively, when the flow is straightforward and there is no curvature, the average absolute delta time around a sample should be approximately $\frac{1}{2}$ (thus the $2\times$ factor in front). This is because the delta time forward is $+1$, the delta time backward is -1, whereas both lateral sides have delta time 0. This measure behaves similarly to the curvature $\kappa(s)$, which is 1 by default, smaller than 1 when the time is going slower, and larger when going faster.

We use the time stretch to provide feedback to the user when we detect abnormal values, which corresponds to large local curvature, and thus a higher likelihood of local time extrema.

3 TOPOLOGICAL OPENING

As a post-processing step after the time function computation (or pre-processing step before the region computation), we topologically open the sketch domain at boundary locations where we have closed sources or sinks of the time function. By assumption, valid sources and sinks must be either (1) on the boundary of the manifold (i.e., unlinked borders of the sketches), or (2) at a local extremum on linked borders of the sketches.

The topological opening targets the latter case and makes it appear the same as the former so that the region computation becomes simpler. Furthermore, our system currently keeps those openings in the final garment artifact. The closing of those regions can be done automatically using specific cast-off procedures, but keeping them open simplifies scheduling and code generation.

There are two scenarios for the topological opening. Fig. 2 illustrates both, using the top of the beanie as an example sink to be opened. In general, the sources/sinks are either:

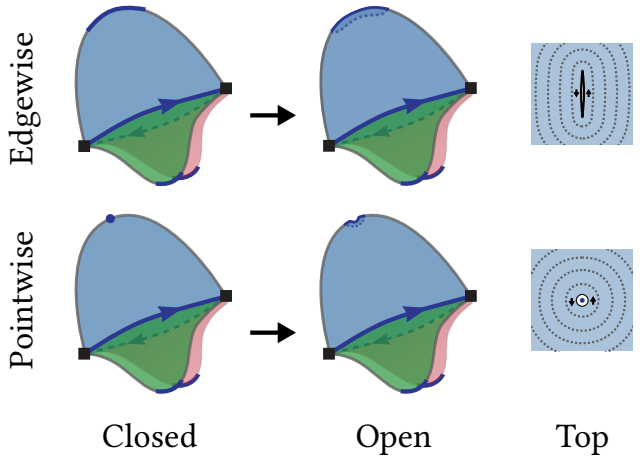


Fig. 2. *Illustration of the topological opening at the closed top of the beanie for two different time extrema: edgewise and pointwise. The top views show the opening in the center and a corresponding isoline profile (dashed lines).*

(1) *Edgewise* - distributed on a portion of the sketch boundaries, or
(2) *Pointwise* - concentrated at a single vertex.

In the first *edgewise* case, we can simply break the link connections on the mesh in the interior of the isoline. Tracing does the rest. In the second *pointwise* case, we use an offset isoline at the closest vertex nearby to represent the source/sink isoline.

The physical beanie result is of the *singular* case, which we knit with an open top, and manually close by passing thread across all last stitches and pulling a thread which we close inside of the beanie.

4 REGION GRAPH PROCESSING DETAILS

The first part considers how many dependency paths are necessary to properly resolve the garment regions. The second part explains the user control $\Delta t_{\min}$ on the maximum region extents.

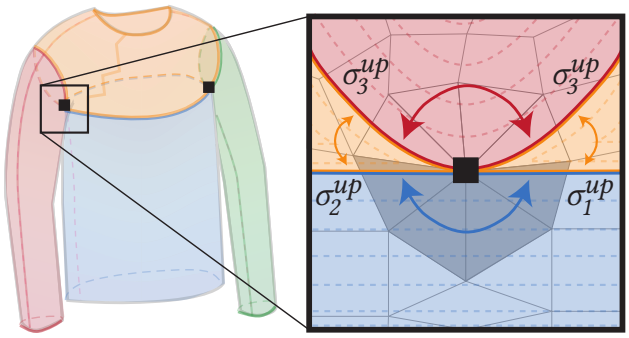


Fig. 3. *Illustration of the adjacent region merging at a separating vertex. Segments σ_1^{up} and σ_2^{up} from the trunk region can reach each others, thus the front and back span the same region. Circulating around the sleeve region has no effect since it is upper-bounded by a single segment σ_3^{up}. As for the segments on the other side of the isoline (lower segments of the upper neck region), all get merged together: σ_1^{low} reaches σ_3^{low} on the right, and σ_2^{low} reaches σ_3^{low} on the left.*

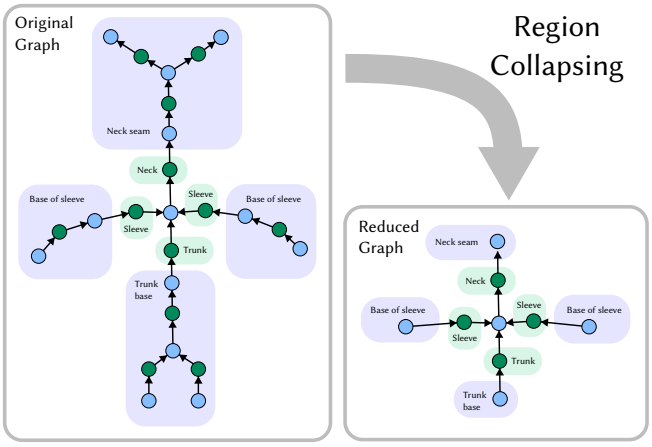


Fig. 4. (Left) *the original graph of the shirt example when the sources and sinks are not sufficiently constrained – e.g., no hard time isoline constraints are set on the sketch boundaries –,* (right) *its reduced graph that looks identical to the ideal one.*

4.1 Necessary Dependency Paths

Our region computation initially allocates two regions per isoline segment. Intuitively, one should not need more because the segments represent the potential sides of the regions at the interfaces, and different regions neighbor either (1) different segments, or (2) same segments, but on different sides.

The choice of dependency path is actually important to ensure we properly merge all regions. Tracing two dependency paths per isoline segment (one upward and one downward) is sufficient to resolve all the initial regions since each allocated region gets resolved. However, this can lead to too many regions if the side regions are not merged properly. We consider two options to properly resolve regions laterally:

(1) Merging regions by circulating around *separating vertices*;
(2) Explicitly tracing all sketch boundaries as dependency paths.

The former option actively merges regions around separating vertices when they are reachable. Fig. 3 shows such merging happening at the original sweater's armpit.

The latter option relies on the fact that *separating vertices* all arise on the sketch boundaries, and thus, by tracing dependency paths along all sketch boundaries, we end up automatically merging all regions that need merging around those separating vertices. This is the strategy we use.

4.2 Graph Post-Processing and $\Delta t_{\min}$

In practice, due to small asymmetries in the user input, we often end up with a graph that is not ideal.

Our region graph can easily end up with many small regions in between (or at the boundaries of) larger regions. For example, the case of the 3-way merge interface of the sweater assumes that we end up with an identical time at both armpits. Here, we measure specifically the *time extents* $\Delta t(r)$ of some region $r = (\mathbf{e}_i, \mathbf{e}_j) \in \mathcal{R}$ as the time range across its boundaries: $\Delta t(r) = |\mathrm{t}(L_j) - \mathrm{t}(L_i)|$.

Our system allows the user to tune the minimum allowable time extents $\Delta t_{\min}$. Given that threshold, we reduce the original region graph by iteratively collapsing any simple region whose extents are too low, until either all regions have sufficient time extents, or there remains only one simple region. Simple regions that collapse become parts of new interface nodes.

Fig. 4 illustrates the impact of region collapsing on an insufficiently constrained sketch atlas so that its graph is *reduced* into one that is a *more ideal* one. The example corresponds to the sweater with the cat colorwork in the main paper.

In practice, this control is desirable because slight variations of the charts and time function can lead to large variations of the region topology, notably near the source and sink locations. In particular, small offsets may inadvertently introduce clusters of critical isolines that are very close to one another, resulting in simple regions that may be too small to hold any stitches during the later sampling and instantiation processes. By pruning these small regions, we increase the robustness of our algorithm, and allow the user to control for any ϵ-errors in the chart sketches and constraints *a posteriori*.

5 IQP PROBLEMS

Our stitch graph computation involves several optimizations that are formulated as *Integer Quadratic Programming* problems with linear constraints. While constrained IQPs are NP-hard in the general case, we explain how our formulations can be solved efficiently.

Our general strategy is to start with the relaxed version of the problem for which we can get a reasonable solution quickly using the NLOpt [Johnson 2014] library. We use the Improved Augmented Lagrangian Method [Birgin and Martínez 2008] as global solver, and L-BFGS [Liu and Nocedal 1989] as local solver. The result is then rounded to the closest integers, which results in the first initial variable state we start from. From there, we use a branch-and-bound strategy to explore the integer solution space, subject to some limited time budget.

5.1 Global Stitch Problem

The first interface sampling optimization searches for global stitch numbers n_i on the edges $\mathbf{e}_i$ of our bipartite region graph:

$$\arg\min_{\mathbf{n}} \lambda_{\text{crs}} \sum_{\mathbf{e}_i \in \mathcal{E}} E_{\text{crs}}(n_i) + \lambda_{\text{smpl}} \sum_{(\mathbf{e}_i, \mathbf{e}_j) \in \mathcal{R}} E_{\text{smpl}}(n_i, n_j)$$
$$\text{s.t. } \forall\, \eta \in \mathcal{I}_{\text{internal}}, \sum_{\mathbf{e}_i \in \mathcal{E}^{\text{in}}_\eta} n_i = \sum_{\mathbf{e}_j \in \mathcal{E}^{\text{out}}_\eta} n_j.$$

First, note that our graph's bipartite structure simplifies the problem, as the constraints must have mutually disjoint variable supports: each n_i can only appear in at most one constraint. This means that the constraints cannot introduce any complex variable dependencies. Thus, in practice, a relaxed non-integer solution can effectively be made into a suitable integer solution by simply rounding the values, and then locally adjusting any variables that violate the constraints. While this does not ensure that we get to the global optimum quickly, it at least ensures that we can get a valid solution quickly.

The other aspect to consider is the number of equality constraints that grows linearly with $|\mathcal{I}|$. To improve convergence, we remove the interface equality constraints via *variable aliasing*. With this approach, we only allocate an explicit variable for $q - 1$ of the q unknowns associated with any particular equality constraint; the value of the remaining unknown is defined implicitly. In the trivial 1-to-1 case, we only need one variable per interface (instead of two). For the general N-to-M merge/split, we can use only $N + M - 1$ variables, with a single inequality constraint that requires the remaining variable to be positive. This reduces the number of variables greatly, while removing all equality constraints, and adding a small number of inequality constraints.

5.2 Local Stitch Problems

The local region optimization tackles two different sizing problems: (1) along the wale direction, and (2) along the course direction. Both are solved for each region $(\mathbf{e}_a, \mathbf{e}_b) \in \mathcal{R}$.

5.2.1 Wale Problem. Recall that the first sizing problem seeks a number N of isoline segment sets $\mathcal{S}_i \in \mathcal{U}$ to allocate along a region, as well as short-row densities r_k in between each pair $(\mathcal{S}_i, \mathcal{S}_j) \in \mathcal{A}$:

$$\arg\min_{N,\, \mathbf{r}} \sum_{(\mathcal{S}_i, \mathcal{S}_j) \in \mathcal{A}} \underbrace{\left(\lambda_{\text{wale}} E_{\text{wale}}(\mathcal{S}_i, \mathcal{S}_j) + \lambda_{\text{srs}} E_{\text{srs}}(\mathcal{S}_i, \mathcal{S}_j)\right)}_{E^N_{i,j}}.$$

We first estimate the expected number N^* based on D_{wale}, which indicates the expected distance between the centers of adjacent wale-connected stitches. For region $(\mathbf{e}_a, \mathbf{e}_b)$, that number is

$$N^* = |t(\mathbf{e}_b) - t(\mathbf{e}_a)| / D_{\text{wale}}. \tag{4}$$

Then, to select our final value of N, we evaluate several integer values around our initial guess N^* and keep the one with the lowest energy $\sum E^N_{i,j}$. This energy involves solving for $\mathbf{r}$ in each of the independent sub-problems $E^N_{i,j}$.

For a given N, we solve for $\mathbf{r}$ in $E^N_{i,j}$ by: (1) finding the relaxed solution with NLOpt, initialized with the solution given by the wale term E_{wale}, then (2) directly rounding it to the closest integer, and (3) enforcing that at least one sample gets $r_k = 0$. While we could use a more complex branch-and-bound strategy to find the global optimum, the computational cost would become prohibitive, as it must be executed for every sub-region, for each selection of N, over every simple region.

5.2.2 Course Problem. The second sizing problem seeks local stitch numbers with a formulation that appears very similar to the global stitch problem. The main difference is the set of constraints and their inter-dependencies:

$$\arg\min_{m} \lambda_{\text{crs}} \sum_{\mathcal{S}_i \in \mathcal{U}} E_{\text{crs}}(m_i) + \lambda_{\text{smpl}} \sum_{(\mathcal{S}_i, \mathcal{S}_j) \in \mathcal{A}} E_{\text{smpl}}(m_i, m_j)$$
$$\text{s.t. } \forall\, (\mathcal{S}_i, \mathcal{S}_j) \in \mathcal{A}, \quad \lceil m_j / F_{\max} \rceil \le m_i \le \lfloor m_j\, F_{\max} \rfloor.$$

The constraints of this local problem are box constraints that interact in a sequential manner. This sequential interaction makes it again possible to quickly find at least one solution. Furthermore, by constraining the minimum number of subdivisions N based on the global stitch counts at the region's start and end interfaces, we can ensure that we have a feasible solution.

6 GEODESIC COMPUTATIONS

We describe our strategy to compute the geodesic distance $\mathrm{G}(p, q)$ between two locations p and q, possibly in different charts of a same atlas. Practically speaking, we focus on computing the shortest path from p to q (i.e., the geodesic path), whereas the distance $\mathrm{G}(p, q)$ is the length of that path.

We require the distance to measure the degree of alignment between different locations within our stitch graph sampling algorithm. We further require the path so as to sample short-row stitches along stitch wales. Having proper sketch embedding of the stitches matters for two purposes: 1) for visualization, which matters for seam editing, and 2) to allow location-based pattern queries and editing.

Our strategy is of hierarchical nature. Our main observation is that when the distance is large, we do not need to be very precise, whereas when we get closer to the target, we would appreciate to get the exact geodesic path.

In a precomputation stage, we store the geodesic distance between any two samples of the finest level of the mesh data structure holding the time function. The distances do not necessarily need to be exact, so a simple strategy is to use N instantiation of Dijkstra based on the mesh connectivity (for N vertices). We instead use the Heat Method [Crane et al. 2017] to get a more precise continuous measure.

Given the precomputation table, upon a query between locations p and q, we compute their sample neighborhood in the mesh (edge or face). If the neighborhood is the same, the geodesic path is trivial and we're done. If they are not, we create an approximate path by linking p and q to all their sample neighbors (2 on an edge, 3 in a triangle and 4 in a quad) and picking the shortest path between any pair of those samples across sides.

This approximation is then tested against a refinement threshold – we set it to $3\Delta_s$ where Δ_s is the distance between two adjacent grid samples. If it is above the threshold, we return the approximate path and its distance.

If it is below, we refine the path by computing the exact geodesic path between two points as described in Surazhsky et al. [2005]. To

further restrict the search space during the edge-window propagation, we only consider the neighborhoods adjacent to our initial approximate geodesic path.

7 STITCH SAMPLING AND ALIGNMENT

Multiple steps of our sampling algorithm rely on the notion of geodesic distance between sampled locations on the sketch atlas. While our geodesic computation strategy deals with one part of the problem, how we pick the sampled locations can matter as much in practice. The issue of sample location naturally arises in two steps: (1) during the short-row density computation of region sampling, and (2) from stitch instantiation to wale distribution.

7.1 Short-row Density Alignment

The computation of the local short-row densities r_i assumes K pairs of samples that are uniformly sampled between two adjacent isoline segment sets $\mathcal{S}_i$ and $\mathcal{S}_j$. Recall that the goal of that computation is to maximize the wale accuracy across a simple region.

Instead of computing the number of short-row stitches directly, the computation uses representative short-row densities that are sampled along the isoline pairs. In practice, we uniformly sample K samples independently on each isoline, and then we create the pairs by matching the samples across both sides with *Dynamic Time Warping* (DTW).

The number of samples pairs K is chosen based on the isoline lengths and the mesh resolution Δ_s. Since the mesh interpolation of t is linear, sampling beyond the mesh resolution brings very limited benefit. However, one subtle issue is that our *uniformly spaced* K samples on both sides $\mathcal{S}_i$ and $\mathcal{S}_j$ have a potential unknown global shift (rotation of circular courses). Increasing the resolution decreases the impact of such global shift on the systematic alignment error, but increasing the stitch resolution also enhances that issue.

An adaptive solution to this problem is to complement the sample-pair DTW-based alignment with a further refinement that iteratively attempts to reduce the global sample shift through a sub-sample alignment procedure. Given K aligned samples $\{s_{i,k}\}$ and $\{s_{j,k}\}$ along adjacent isoline segment sets $\mathcal{S}_i$ and $\mathcal{S}_j$, we successively look for global shifts that would improve the full alignment. We pick the potential shifts by subdividing the interval between two neighboring samples in a binary fashion while searching for a shift that reduces the alignment error. We use a fixed number of subdivision levels (5) but this could be adapted with the stitch scale.

Instead of applying this procedure to each sub-region, we only apply it if the global alignment shows an average distance that is unexpectedly large (i.e., short-row densities above 0 on at least half of the samples). For smooth time functions, short-rows are only needed sparsely and such adaptive sub-sample alignment procedure ends up only firing where (i) short-rows are needed, or (ii) the random offset is large.

7.2 Stitch Course Alignment

The uniform stitch distribution during the creation of courses leads to a similar potential misalignment during wale distribution. However the impact is very different because wale distribution implicitly distributes any form of local misalignment, and thus it is typically not notable when considering purely the topological graph structure. And even if it was noticeable, the seam penalty provides user control to override any local misalignment.

Fig. 5. *Examples of flat layouts for $N = 8$ stitches.* ***Left****: single-fold with $r = 2$,* ***right****: c-shaped with $s = F$, $l = 1$, $r = 2$ and $m = 5$.*

8 MIXED FLAT/CIRCULAR SCHEDULING

For tubular structures, Narayanan et al. [2018] use two parameters: a *roll* that represents the rotation of the cycle on the needle bed, and a *nibble* that describes the different possible corner deformations from the ideal aligned layout. This leads to $4N$ and $5N$ possible layouts for N stitches, depending on whether N is odd or even.

Perhaps counter-intuitively, describing a flat layout involves a much larger design space since needles can still be rotated along and across the beds, but they have much fewer restrictions on their location. Two simple layouts we consider are the *single-fold* layout, and the *C-shape* layout, illustrated in Fig. 5.

8.1 Single-Fold Layout

The single-fold layout parameterizes the layout of N stitches by using that of a tubular layout with $2N$ stitches, while using only its N first stitches. In practice, there are some *nibble* configurations that become invalid as the *roll* changes since we only have at most two corners available at any time (on the side where the fold happens).

A priori, that puts us in the same space as circular layouts for complexity (linear in N), but the catch is that scheduling doesn't require only the layout itself for binding, but also to constrain the available locations of other layouts. In the case of the circular layouts, the pair (N, nibble) is sufficient to compute the extents of the layout – i.e., the rotation doesn't matter. In the flat case, however, the rotation matters when packing cycles next to each others, which increases the search space substantially.

8.2 C-Shape Layout

A common needle layout is called "C-shaped" which refers to a flat structure that is potentially folded twice over the bed, to form a C-shape (modulo some rotation).

One potential parameterization for such C-shaped layout is as follows, sequentially:

- A number of *main* stitches $m \in [\lceil N/2 \rceil; N]$,
- A *side* for the main stitches $s \in \{F, B\}$, and
- A number of secondary *left* stitches $l \in [0; N - m]$.

From N, m and l, we can infer the number r of secondary stitch locations on the *right* side (for the secondary fold) as

$$r = N - m - l. \tag{5}$$

Obviously, one can change the parameterization through that substitution as is done in Fig. 5 with (s, l, r) instead of (m, s, l). Furthermore, note that we do not consider *nibbles*, although they would matter in the general case.

Sample		Complexity					Parameters		
Target Size	**Sketch**	**Charts**	**Constraints**	**Regions**	**Stitches**	**Instructions**	$\Delta t_{\min}$	λ_{srs}	**Options**
4 feet mannequin	beanie	2	8	3	13184	34892	0.25	0.0	Uniform branching
	sweater	2	19	4	47624	97987	0.25	0.1	—
	trousers	12	22	6	57254	120364	0.5	0.1	—
16 inch mannequin	cardigan	4	12	4	12290	31351	0.25	0.1	Reverse time
	dress	14	26	4	17238	41704	0.5	0.1	Reverse split
	hoodie	6	18	5	12874	29136	0.5	0.3	—
	jacket	5	17	4	11252	31184	1.0	0.1	Reverse split
	turtleneck	8	24	4	13426	24752	0.25	0.1	—
	shorts	4	23	3	2842	7673	0.25	0.1	Reverse split
	l trousers	12	22	6	11104	25226	0.5	0.1	—
	w trousers	6	14	3	14804	25014	0.25	0.1	Reverse split

Table 1. *Statistics about the result samples shown in the paper. The number of stitches corresponds to the number of* traced stitches *which are used to generate the schedule. Given that the yarn is traced twice over, this is twice the amount of stitches in the stitch graph.*

N	2	5	10	20	50	100	200	500	1000
$C(N)$	2	12	40	130	700	2650	10300	63250	251500

Table 2. *The number of possible C-shape layouts given N stitches, without taking the layout offset into account.*

Unfortunately, because we need two parameters whose extents vary proportionally to N, the number of possible layouts $C(N)$ becomes now quadratic in N, as further supported by Table 2.

The second issue concerns the extents of the layouts during left-to-right packing on the bed (and collision avoidance between layouts). In the general case, the scheduler should allow other flat layouts to nest in between two folds of one C-shaped layout.

8.3 Simplified C-Shape Layout

Given some of the issues with the general *C-Shape Layout*, we instead use a simplified version of it. Since our scheduler implementation does not support any form of nesting of layouts, the simplified layout can assume that we don't do any form of nesting. Furthermore, to make the layout exploration space linear, we restrict it to use a single parameter that scales with the stitch number N.

We basically keep only the two first parameters of the previous layout: a number $m \in [\lceil N/2 \rceil; N]$ of *main* stitches, and their *side* $s \in \{F, B\}$. The remaining values l and r are inferred while assuming that the secondary side spreads the folded stitches uniformly between left and right side.

To avoid having to choose between layouts, we add an additional parameter $a \in$ {left, right, both} that describes the secondary layout: either all packed on the *left*, *right* or spread evenly across *both* sides. This is the default flat layout we use in practice.

9 SCALABILITY

9.1 Complexity

Table 1 provides statistics about each of the results presented in this work. As can be observed, the number of charts varies a lot, but most of our garments are made of a small number of simple regions (from 3 to 6).

9.2 Parameters

Table 1 further includes a list of varying parameters across our results. The default $\Delta t_{\min}$ threshold was set to 0.25 and varied for some of our samples so as to ensure simple merging interfaces. The sampling tradeoff parameters were modulated through the simplicity weights λ_{smpl}, λ_{srs} and the seam weight λ_{seam}. The other weights were fixed: $\lambda_{\text{crs}} = \lambda_{\text{wale}} = \lambda_{\text{dist}} = 1$. By default, we initially set the course simplicity $\lambda_{\text{smpl}} = 0$ to try and get perfect accuracy and increased it between 0.1 and 0.3 when our initial knitting results had issues with the scheduling (e.g. for the crotch section of the trousers). By later adjusting the sketch, the course simplicity can typically be reduced, if not completely removed (i.e. set back to $\lambda_{\text{smpl}} = 0$). The only final result which still required the simplicity term was the hoodie with $\lambda_{\text{smpl}} = 0.1$. The short-row simplicity was set to $\lambda_{\text{srs}} = 0.1$ by default. The two cases were it was changed were the *beanie* for which we disabled short-rows completely, and the *hoodie* which took us a few attempts to knit properly.

The last column of Table 1 lists options which are associated with the individual results. *Uniform branching* was used to enforce that the two ear flaps of the beanie would end up with the same number of stitches on both sides, which leads to a much simpler layout space. *Reverse time* is a simple toggle that allows the user to reverse the sketch time instead of manually reversing the constraints. The initial design of the cardigan had a time function from bottom to top, which was then reversed. *Reverse split* corresponds to using a more advanced form of stitch increase instead of the default, simpler *kickback increase*. For those results, we used the *reverse split inward* variant shown in Fig. 8.

9.3 Interactivity

Table 3 lists runtimes of different sections of our system. The following paragraphs provide an interpretation of these runtimes.

Mesh-based Timings. The *left* group (*time, segment, geo*) is mainly mesh-dependent. The number of mesh levels highly impacts the runtimes. The first two parts (*time* and *segment*) deal with the iterative system for specifying the time function and getting its corresponding region graph. This all happens within a second, and feedback

Sketch	Charts	Levels	Time	Segment	Geo	Regions	Local	Binding	Stitches	Wales	Itfs	Nodes	Code
beanie	2	3	0.2	0.1	1.1	3	8.6	0.1	13184	12.1	0.4	0.8	1.1
sweater	2	3	0.1	0.1	0.3	4	5.2	13.5	47624	27.9	2.1	2.9	3.7
trousers	12	3	0.3	0.2	0.8	6	9.8	13.7	57254	40.9	1.9	3.9	6.6
cardigan	4	3	0.1	0.1	0.6	4	2.8	0.0	12290	3.3	0.0	0.0	0.7
dress	14	2	0.8	0.4	0.8	4	8.4	1.9	17238	17.3	1.3	0.8	1.8
hoodie	6	3	0.8	0.2	24.4	5	36.4	2.1	12874	7.8	17.6	0.1	1.3
jacket	5	3	0.5	0.1	12.9	4	18.8	1.7	11252	7.7	0.4	0.3	1.2
turtleneck	8	3	0.8	0.1	1.8	4	11.0	3.0	13426	10.6	1.2	0.4	0.8
shorts	6	2	0.1	0.1	1.0	3	4.2	0.4	2842	2.7	0.5	0.1	0.3
l trousers	12	3	0.2	0.4	0.8	6	7.8	1.9	11104	8.2	22.2	0.3	2.5
w trousers	6	3	0.6	0.2	2.5	3	13.0	0.7	14804	7.3	0.2	0.2	0.8

Table 3. *Runtimes using a single computation thread. Sections that are not included (e.g., global sampling, short-row insertion, offset optimization) are too fast to be relevant (typically less than 100 milliseconds is spent). The column values correspond either to number counts or time measurements in* seconds.

typically comes in even less time given the coarse-to-fine, iterative nature of our computations.

The *Geo* column considers the geodesic distance precomputation, which is not triggered until sampling. Sec. 6 presents our default strategy based on the Heat Method [Crane et al. 2017]. For the sketches `hoodie`, `jacket` and `w trousers`, we had to resort to a simpler *Dijkstra*-based precomputation because of issues with the underlying meshing implementation. This leads to a major overhead.

Region-based Timings. The *center* group (*local, binding*) is bound to the number of regions and their interfaces. While local sampling is one of the two most expensive stages of our computations, it could easily be parallelized (at least by region). Similarly, the binding computation could be parallelize, but we note that the current large times are for cases where such binding is spent mostly at a single interface, and thus would be hard to parallelize. However, since a lot of the computations done during that step are very similar to the scheduling of interfaces, we may benefit from sharing information across both sides (to speed up the scheduler).

Stitch-based Timings. Wale distribution is the other most intensive computation of our system, but it can also easily be parallelized, and at even larger scale. One element that is less visible in this table is that the variance of the remaining scheduling operations can highly vary depending on the symmetries of the structure to be scheduled (up to the eveness of the number of stitches).

9.4 Convergence of the Optimizations

The *time function* computation is prone to local extrema. As discussed in the *curvature* section, this is highly dependent on the user-specified constraints and their interactions. For example, close-by time isoline constraints can lead to large time stretching which make the underlying system poorly conditioned (because the constrained sketches do not represent flat intrinsic geometry anymore). Another example is that of nearby conflicting directions constraints. Our strategy is focused on getting early visual feedback (both through a coarse-to-fine computation, and fast iterative updates), so that the user can explore those issues interactively and address them.

The *stitch graph sampling* has two main hierarchical steps that behaves quite differently in terms of convergence. In practice, none showed cases of obvious local extrema, but this is likely because our garment results had simple shaping constraints.

9.4.1 Global Sampling. typically converges well because the variables interact in small groups, purely locally, for which branch and bound can quickly reach a global extremum.

9.4.2 Local Sampling. has a more complex, *sequential* interaction profile (constraints between adjacent isolines) that can supposedly lead to bad local extrema in case of wild shaping. We did not encounter odd behaviors in our examples. We had mainly two regimes: (1) fast single-direction shaping regions for which the local region boundaries would induce an obvious single optimal solution (e.g., top of sweater), and (2) slowly shaping regions for which the the local constraint interactions were reasonably far, and thus with good convergence. We expect that the main cases where local extrema would occur are for reasonably fast shaping regions that alternate increase/decrease within nearby locations. One solution to those would be to allow the user to subdivide the regions locally, which would break ambiguities at the local region level.

10 THE IMPORTANCE OF DETAILS

We highlight three different aspects that have important impacts on the final garment appearance: the impact of colorwork and *customizable stitch patterns*, the implementation of specific *knitting procedures*, the problem of proper *sizing*, and the placement of *seams*.

Customizing stitch patterns. Fig. 6 shows that for a same shape in our system, the addition of some color work can have a dramatic impact on the perceived quality of the result. Having proper tools to design this on top of the stitch graph would make our system much more effective for customization.

Sizing. Sizing is a critical part of garment design. Our system allows to specify the final scale, but getting the proper scale can be tricky. In the sweater of Fig. 7, by slightly changing the scale, we go from a shirt that looks pretty tight, to one that is appropriately

Fig. 6. *The addition of color work and stitch patterns can highly improve the final appearance, which calls for dedicated means to specify those.*

Fig. 7. *Two slight scale variations of a same shirt input showing the importance of proper sizing.*

loose, if not too loose. Having dedicated size constraints as part of our time constraint design tool would be very helpful.

Knitting Procedures. Fig. 8 shows that the type of knitting procedures for local aspects of shaping can have a big impact on the final appearance. We highlight here the case of the shaping increase procedure, which is purely local to the stitch it happens at. It results in varying degrees of tightness, and potential visible hole artifacts that can be important.

Seams. As mentioned in the main paper, we provide support for editing the seam placement interactively. Here we show closer looks at some triangular patterns in Fig. 9 that highlight some of what works and some of what does not. The seam location is not the only part of the design that matters for the final appearance of the seam. One needs sufficient alignment between successive irregular stitches to produce an appealing seam. Furthermore, the general clusters of wale directions also plays an important role in making the seams appear more or less visible.

11 INDIVIDUAL RESULTS

The rest of this document provides top-down closeup pictures of the results, together with visualization of the inputs (linking, constraints), the results of the time computation and decomposition (time, time stretch, regions) and the sampled stitch graph.

The format of those figures is as follows:

- the *left*/*top* of the first figure shows top-down views of different sides of the samples (including potential inside-out view for colorwork),
- the *right*/*bottom* of the first figure shows successively:
  - the input *linking*,
  - its direction and time *constraints*,
  - a visualization of the underlying *time function*,
  - a visualization of the corresponding *time stretch*,
  - a visualization of its *regions*, and
  - its stitch graph overlaid on the sketch atlas.
- the second figure provides the stitch program that was used to define the operations on the stitches of the final stitch graph. By default, all stitches are associated with a program that depends on their connectivity: *Knit* for 1-1 and 2-1 connectivity, and *Kickback Knit*, *Split*, *Reverse-Split* or *Miss* for each elements of an increase pair 1-2.

12 VARIANTS OF KNITTED SAMPLES

Our knitting process was obviously not without flows. We end the supplementary by highlighting the evolution of some of our samples, which shows the iterative nature of weft knitting.

Most of our samples required multiple iterations, notably to get the size right, adjusting the shape when the induced scheduling was leading to failure, or to test different types of stitch patterns and color work.

REFERENCES

Ernesto G Birgin and José Mario Martínez. 2008. Improving ultimate convergence of an augmented Lagrangian method. *Optimization Methods and Software* 23, 2 (2008), 177–195.

Keenan Crane, Clarisse Weischedel, and Max Wardetzky. 2017. The Heat Method for Distance Computation. *Commun. ACM* 60, 11 (Oct. 2017), 90–99.

Steven G Johnson. 2014. The NLopt nonlinear-optimization package. http://github.com/stevengj/nlopt

Dong C Liu and Jorge Nocedal. 1989. On the limited memory BFGS method for large scale optimization. *Mathematical programming* 45, 1-3 (1989), 503–528.

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.

Vitaly Surazhsky, Tatiana Surazhsky, Danil Kirsanov, Steven J. Gortler, and Hugues Hoppe. 2005. Fast Exact and Approximate Geodesics on Meshes. *ACM Trans. Graph.* 24, 3 (July 2005), 553–560.

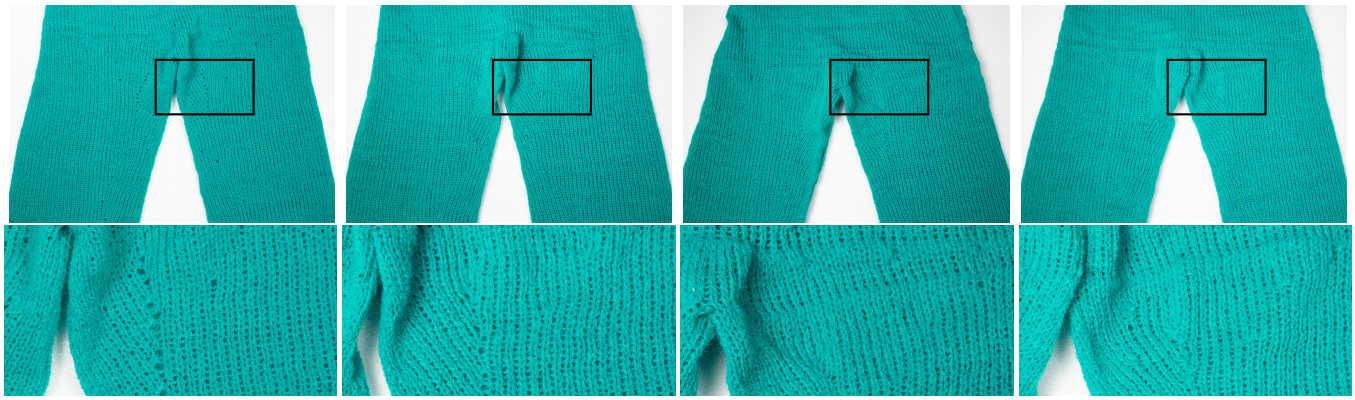


Fig. 8. *Local appearance of different stitch increase procedures, from left to right: kickback, reverse split inward, reverse split outward, split outward.*

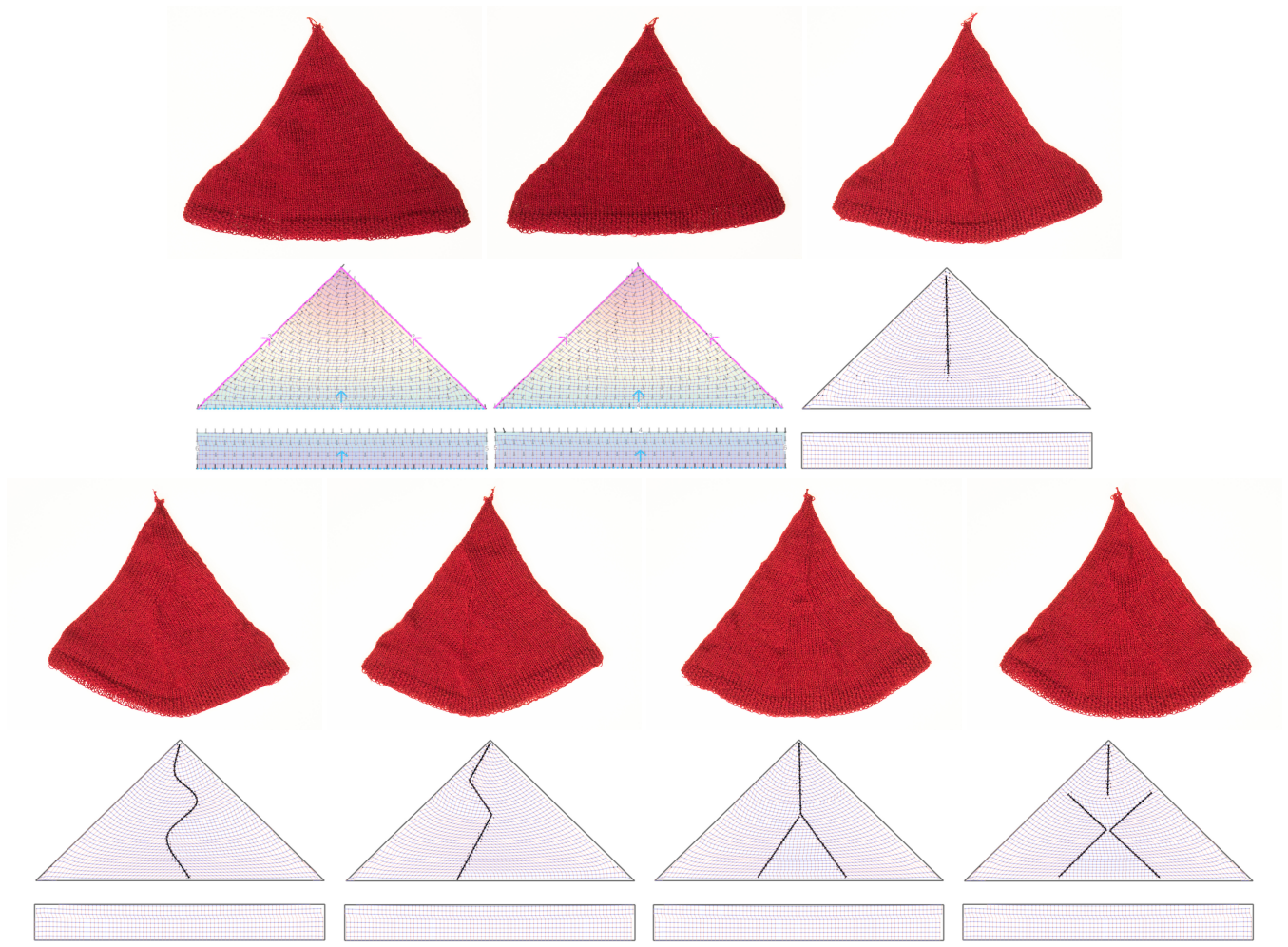


Fig. 9. *Illustration of the impact of seam annotations with the corresponding irregular stitch placement.*

Fig. 10. *Beanie with color work*

```
const reg = s => !s.fromDecrease() && s.getPrevWales().some(ps =>
      !ps.toIncrease())  && s.stitch.countCourses() === 2;
const rev = n => n.otherHook();
const purl = Action.register({
  pre: (k, d, [n]) => k.xfer(n, rev(n)),
  main: (k, d, [n], cs) => k.knit(d, rev(n), cs),
  post: (k, d, [n]) => k.xfer(rev(n), n),
  splitBySide: true
});
const ears = prog.node(0).or(prog.node(1));
const bnds = ears.boundaries().neighbors(0:2);
const ins = ears.minus(bnds);
ins.filter(s => reg(s) && s.index % 2).prog(purl);



// ---------------------------------------------------
// mountain colorwork --------------------------------
// ---------------------------------------------------

const cs2 = ['2'];

// main mountain region
const back = Action.register({
  main: [
    ({ k, d, n, cs }) => k.knit(d, n, cs),
    ({ k, d, n }) => k.miss(d, n, cs2)
  ],
  splitBySide: true
});
const front = Action.register({
  main: [
    ({ k, d, n, cs }) => k.miss(d, n, cs),
    ({ k, d, n }) => k.knit(d, n, cs2)
  ],
  splitBySide: true
});

// second yarn handling
const yarnIn = back.extend({
  pre: ({ k, d, n, e }) => {
    k.inhook(cs2);
    k.tuck(d, n, cs2);
    k.tuck(d, e.stepNeedle(2), cs2);
    k.tuck(-d, e.stepNeedle(1), cs2);
    k.releasehook(cs2);
  }
});
const yarnOut = back.extend({
  post: ({ k }) => k.outhook(cs2)
});

const mounturl = '<dataurl>';

// interface for color work
const itf = prog.node(2).and(prog.node(0).or(prog.node(1)).up()).
      up();
const grid = itf.waleGrid(0:end, 30);
grid.prog(back);
const img = prog.parseImage(mounturl);
grid.tileMap(img, {
  0: front,
  255: back
}, 30, 4, 0);

// yarn handling
grid.first().prog(yarnIn);
// rows.first().up().prog(yarnRelease);
grid.last().prog(yarnOut);
```

Listing 1. *beanie.js*

Fig. 11. *Stitch program of the beanie including color work*

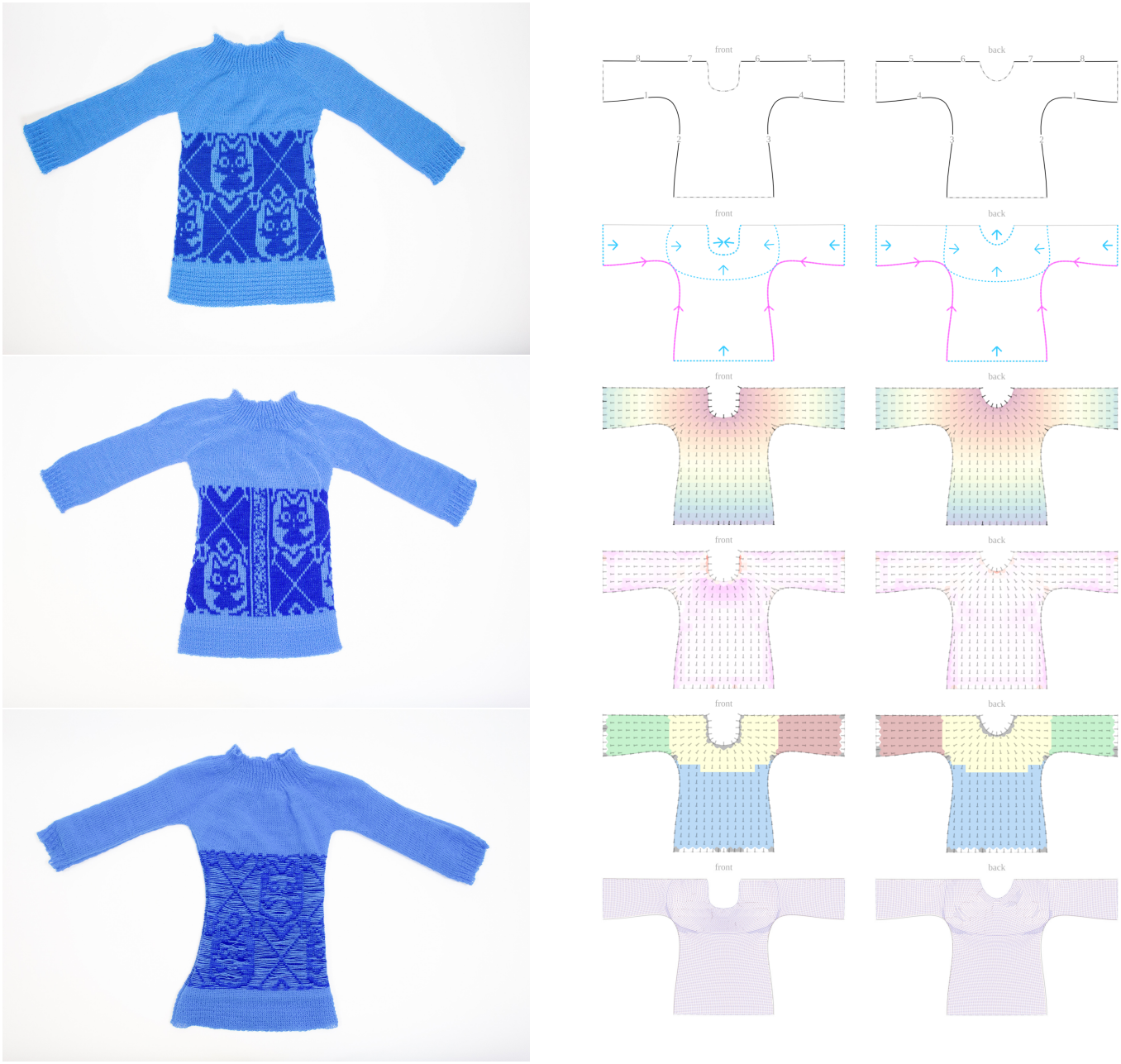


Fig. 12. *Sweater with cat color work*

```
const purl = Action.register({
  pre: ({ k, n, rn }) => k.xfer(n, rn),
  main: ({ k, d, rn, cs }) => k.knit(d, rn, cs),
  post: ({ k, rn, n }) => k.xfer(rn, n),
  splitBySide: true
});
const twoInches = prog.lengthToWaleStitches('2 in');

// neck edge
const neckLen = twoInches + twoInches % 4;
const neckEdge = prog.filter(s => s.countNextWales() === 0);
neckEdge.waleGrid(0:end, -2).tile(0b1001, 2).prog(purl);
// march down to create waffle pattern
for(let i = 0, crs = 0; i < neckLen; i += 4, crs += 2){
  const top = prog.courses(-2 - crs);
  // knit line (nothing to do)
  // purl line (purl all)
  top.pass(0).prog(purl);
  // garter bottom
  const bot = prog.courses(-3 - crs);
  bot.pass(0).waleGrid(0:end, 2).tile(0b1001, 2).prog(purl);
}

// trunk base
const bot = prog.node(0).filter(s => s.countPrevWales() === 0).
      waleGrid(0:end, 8);
bot.tile(0b1001, 2).prog(purl);

// wrists
const wriPat = prog.strToGrid(`
kkppkk
kkppkk
pkppkp
pkppkp
pkkkkp
pkkkkp
`);
const wriLen = Math.max(
  wriPat.length * 2,
  twoInches - twoInches % wriPat.length
);
const wri = prog.node(1).or(prog.node(2)).filter(s => s.
      countPrevWales() === 0).waleGrid(0:end, wriLen).tileMap(
      wriPat, {
  k: Action.Knit,
  p: purl
});


// -----------------------------------------------------
// cat colorwork -------------------------------------
// -----------------------------------------------------

const cs2 = ['2'];

// main cat region
const back = Action.register({
  main: [
    ({ k, d, n, cs }) => k.knit(d, n, cs),
    ({ k, d, n }) => k.miss(d, n, cs2)
  ],
  splitBySide: true
});
const front = Action.register({
  main: [
    ({ k, d, n, cs }) => k.miss(d, n, cs),
    ({ k, d, n }) => k.knit(d, n, cs2)
  ],
  splitBySide: true
});

// secondary foreground region (for tucks)
const tuck = Action.register({
  main: [
    ({ k, d, n, e, cs }) => e.actIdx % 5 === 2 ? k.tuck(d, n, cs)
       : k.miss(d, n, cs),
    ({ k, d, n }) => k.knit(d, n, cs2)
  ],
  splitBySide: true
});

// second yarn handling
const yarnIn = back.extend({
  pre: ({ k, d, n, e }) => {
    k.inhook(cs2);
    k.tuck(d, n, cs2);
    k.tuck(d, e.stepNeedle(2), cs2);
    k.tuck(-d, e.stepNeedle(1), cs2);
    k.releasehook(cs2);
  }
});
const yarnOut = back.extend({
  post: ({ k }) => k.outhook(cs2)
});

const caturl = '<imgdata...>';

// center stitch on front
const cs = prog.sketch(0).nearPosition({ x: 0, y: 150 });
const bl = cs.down(27).left(14);

const rows = bl.down(2).stitchGrid(Infinity, 63);
rows.prog(back);

rows.tile(prog.strToGrid(`
---o-
--o--
-o---
o----
-o---
--o--
`, c => c === 'o')).prog(tuck);

const grid = bl.stitchGrid(30, 60);
const img = prog.parseImage(caturl);
const cat = grid.stretch(img, 30, 4, 0, v => !v);
// create active background by expanding the cat front
const nearcat = cat.withPrev(3).withNext(3);
nearcat.prog(back);
cat.prog(front);

// yarn handling
rows.first().prog(yarnIn);
// rows.first().up().prog(yarnRelease);
rows.last().prog(yarnOut);
```

Listing 2. *sweater.js*

Fig. 13. *Stitch program of the sweater including cat color work*

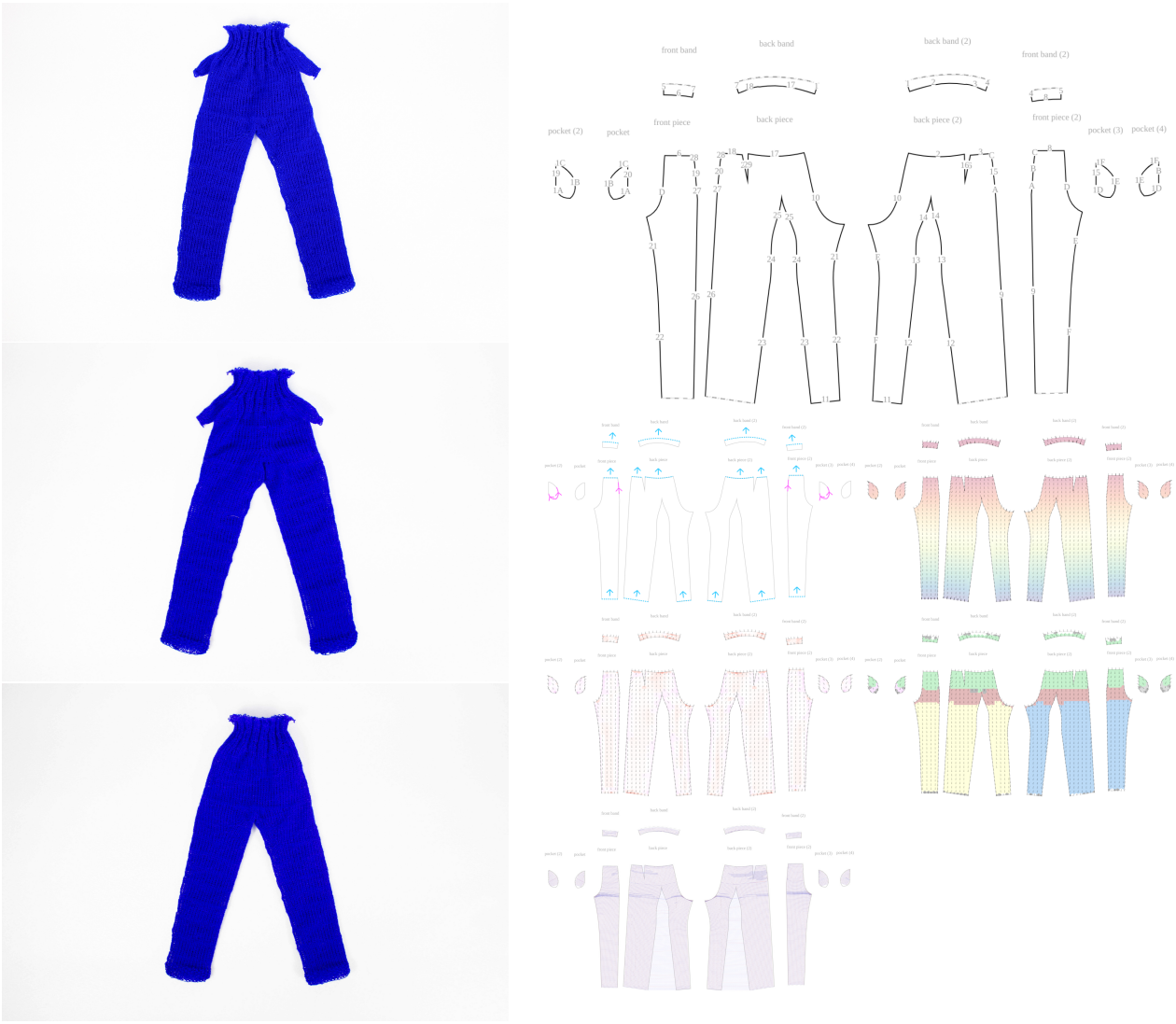


Fig. 14. *Trousers with pockets*

```
const purl = Action.register({
  pre: ({ k, n, rn }) => k.xfer(n, rn),
  main: ({ k, d, rn, cs }) => k.knit(d, rn, cs),
  post: ({ k, rn, n }) => k.xfer(rn, n),
  splitBySide: true
});
const ribLen = prog.lengthToWaleStitches('2 in');
prog.filter(s => s.countNextWales() === 0).waleGrid(0:end, -
     ribLen).tile(0b1100, 4).prog(purl)
prog.filter(s => s.countPrevWales() === 0).waleGrid(0:end, 8).
     tile(0b1001, 2).prog(purl)

// clear pocket of patterns
prog.node(3).prog(Action.Knit);
prog.node(4).prog(Action.Knit);
```

Listing 3. *trousers.js*

Fig. 15. *Stitch program of the trousers including pockets*

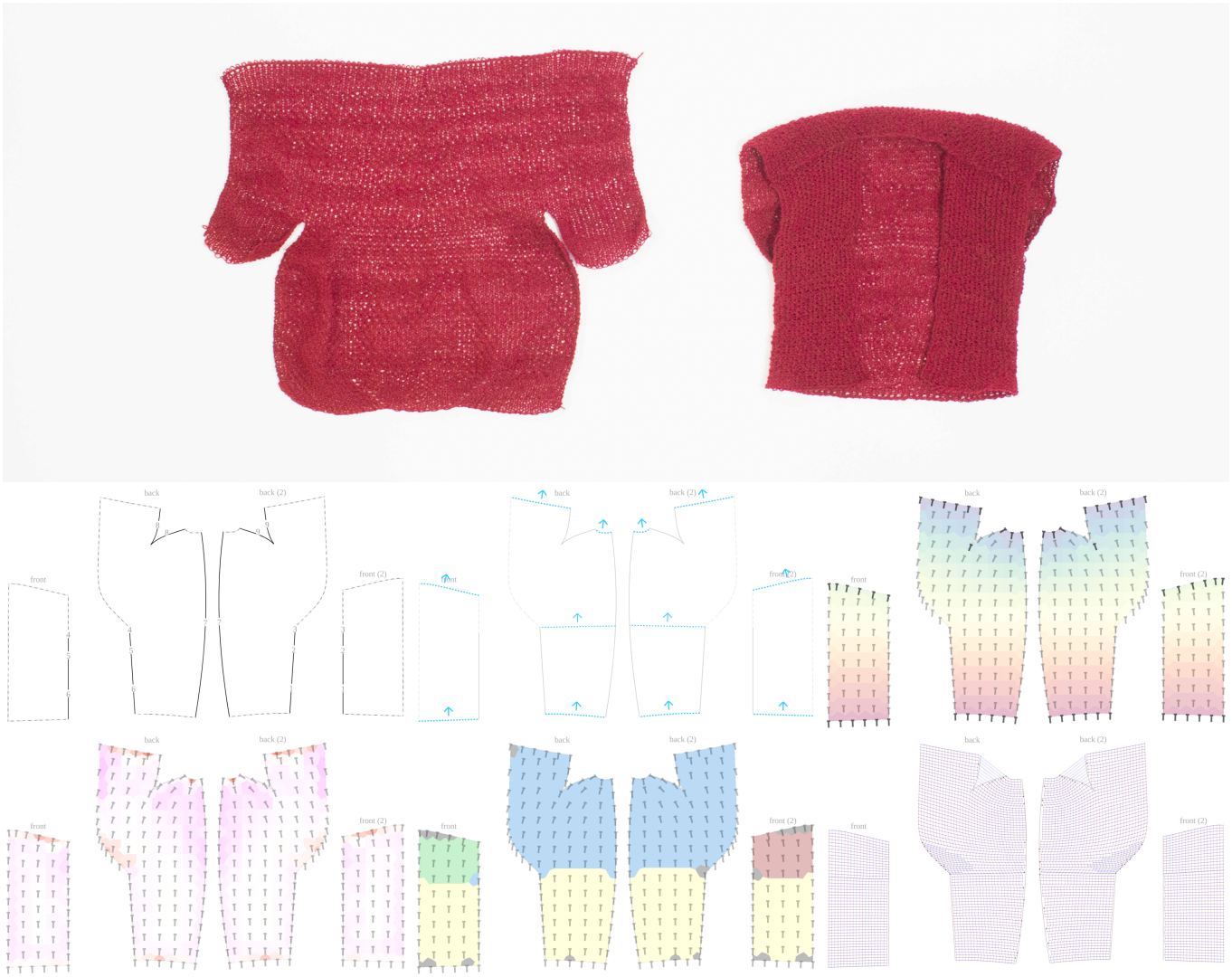


Fig. 16. *Cardigan*

```
const purl = Action.register({
  pre: ({ k, n, rn }) => k.xfer(n, rn),
  main: ({ k, d, rn, cs }) => k.knit(d, rn, cs),
  post: ({ k, rn, n }) => k.xfer(rn, n),
  splitBySide: true
});
```

```

// garter everywhere
prog.filter(s => (s.getGroupData()[0] > 0 || s.pass) && s.stitch.
      index % 2 === s.pass).prog(purl);
```

Listing 4. *cardigan.js*

Fig. 17. *Stitch program of the cardigan*

Fig. 18. *Princess dress*

```
const left = Action.register({
  main: Action.knit,
  post: ({ move }) => move(-1)
});
const right = Action.register({
  main: Action.knit,
  post: ({ move }) => move(1)
});
const purl = Action.register({
  pre: ({ k, n, rn }) => k.xfer(n, rn),
  main: ({ k, d, rn, cs }) => k.knit(d, rn, cs),
  post: ({ k, rn, n }) => k.xfer(rn, n),
  splitBySide: true
});

// 1 = top patterns
const topLen = prog.lengthToWaleStitches('1 in');
// - garter on top edge
const top = prog.filter(s => s.countNextWales() === 0);
top.waleGrid(0:end, -5).tile(0b1001, 2).prog(purl);

// - pointelle below garter
const ptlPat = prog.strToGrid(`
kk
kr
kk
lk`);
const ptlCrs = top.neighbors(0:5).boundaries().down().indices.
      reduce((min, idx) => {
  return Math.min(min, prog.stitches[idx].getGroupData()[0]);
}, Infinity);
prog.courses(ptlCrs).pass(1).waleGrid(0:end, -topLen).tileMap(
      ptlPat, {
  l: left,
  r: right,
  k: Action.Knit
});

// waist ribs
const verRibs = prog.strToGrid(`
kp`, c => c === 'p' ? 1 : 0);
const diaRibs = prog.strToGrid(`
kkkppp
kkpppk
kpppkk
pppkkk
ppkkkp
pkkkpp
`, c => c === 'p' ? 1 : 0);
const waistCrs = prog.node(0).filter(s => s.stitch.getTime() <
      3.9).indices.reduce((max, idx) => {
  const s = prog.stitches[idx];
  return s.getGroupData()[1] ? max : Math.max(max, s.getGroupData
      ()[0]);                                    }, 0);
const ribs = prog.courses(waistCrs).pass(0).waleGrid(0:end, -12);
ribs.tile(verRibs).prog(purl)


// wrist semi-ribs
const sleeves = prog.node(1).or(prog.node(2));
const wriRibs = prog.strToGrid(`
kkpk
kkpk
pkpk
pkpk
pkkk
pkkk
`);
sleeves.filter(s => s.countPrevWales() === 0).waleGrid(0:end, 12).
      tileMap(wriRibs, { k: Action.Knit, p: purl })


// skirt bottom hem
const bot = prog.strToGrid(`
krrrrrrkppkllllllk
kkkkkkkkppkkkkkkkk
pkrrrrrkppkllllkp
pkkkkkkkppkkkkkkp
ppkrrrrkppkllllkpp
ppkkkkkkppkkkkkppp
pppkrrrkppkllkppp
pppkkkkkppkkkkppp
ppppkrrkppkllkpppp
ppppkkkkppkkkkpppp
pppppkrkppklkppppp`);

prog.node(0).filter(s => s.countPrevWales() === 0).up().waleGrid
      (0:end, bot.length).tileMap(bot, {
  'k': Action.Knit,
  'l': left,
  'r': right,
  'p': purl
});
```

Listing 5. *princess dress.js*

Fig. 19. *Stitch program of the princess dress*

Fig. 20. *Hoodie*

```
const purl = Action.register({
  pre: ({ k, n, rn }) => k.xfer(n, rn),
  main: ({ k, d, rn, cs }) => k.knit(d, rn, cs),
  post: ({ k, rn, n }) => k.xfer(rn, n),
  splitBySide: true
});

// wrist ribbed cuffs
const ribLen = prog.lengthToWaleStitches('1 in');
prog.node(0).or(prog.node(1)).filter(s => s.countPrevWales() ===
      0).waleGrid(0:end, ribLen).tile(0b10, 2).prog(purl)

// base garter
prog.node(2).filter(s => s.countPrevWales() === 0).waleGrid(0:end
      , 8).tile(0b1001, 2).prog(purl);
```

Listing 6. *hoodie.js*

Fig. 21. *Stitch program of the hoodie*

Fig. 22. *Jacket*

```
const purl = Action.register({
  pre: ({ k, n, rn }) => k.xfer(n, rn),
  main: ({ k, d, rn, cs }) => k.knit(d, rn, cs),
  post: ({ k, rn, n }) => k.xfer(rn, n),
  splitBySide: true
});

// side garter
const sides = prog.filter(s => s.stitch.countCourses() === 1 && !
      s.stitch.isShortRow());
sides.neighbors(0:4).filter(s => s.index % 2).prog(purl)

const ribLen = prog.lengthToWaleStitches('4 in');
const top = prog.filter(s => s.countNextWales() === 0);
top.waleGrid(0:end, -ribLen).tile(0b10, 2).prog(purl);

const garter = prog.strToGrid(`
kp
pk
`);
prog.filter(s => s.countPrevWales() === 0).waleGrid(0:end, 8).
      tileMap(garter, {
  k: Action.Knit, p: purl
});

top.withDown(2).prog(Action.Knit);
```

Listing 7. *jacket.js*

Fig. 23. *Stitch program of the jacket*

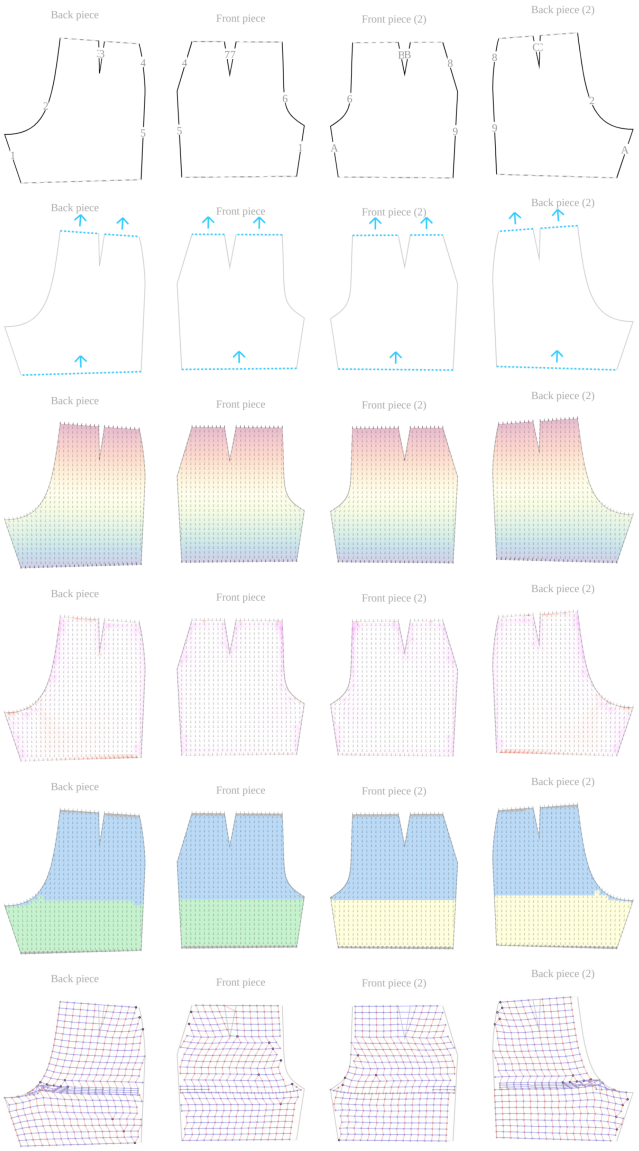


Fig. 24. *Shorts*

```
const purl = Action.register({
  pre: ({ k, n, rn }) => k.xfer(n, rn),
  main: ({ k, d, rn, cs }) => k.knit(d, rn, cs),
  post: ({ k, rn, n }) => k.xfer(rn, n),
  splitBySide: true
});

// waist 2x2 ribs
const ribLen = prog.lengthToWaleStitches('1 in');
prog.filter(s => s.countNextWales() === 0).waleGrid(0:end, -
      ribLen).tile(0b1100, 4).prog(purl);

// waffle pattern
prog.filter(s => s.countPrevWales() === 0).waleGrid(0:end, 8).
      tile(0b0010010011, 2).prog(purl)
```

Listing 8. *shorts.js*

Fig. 25. *Stitch program of the shorts*

Fig. 26. *Turtleneck dress*

```
const purl = Action.register({
  pre: ({ k, n, rn }) => k.xfer(n, rn),
  main: ({ k, d, rn, cs }) => k.knit(d, rn, cs),
  post: ({ k, rn, n }) => k.xfer(rn, n),
  splitBySide: true
});

// neck ribs
const ribLen = prog.lengthToWaleStitches('2 in');
prog.filter(s => s.countNextWales() === 0).waleGrid(0:end, -
      ribLen).tile(0b10, 2).prog(purl)

// base hem
const hem = prog.strToGrid(`
lkkkkppkk
kkrkkppkk`);
prog.node(0).filter(s => s.countPrevWales() === 0).up().waleGrid
      (0:end, 6).tile(0b1001, 2).prog(purl)
const base = prog.node(0).filter(s => s.countPrevWales() === 0).
      up(8);
base.waleGrid(0:end, 12).tileMap(hem, {
  k: Action.Knit, p: purl,
  l: left, r: right
});
const len = base.indices.length;
const rem = hem[0].length - (len % hem[0].length);
base.waleGrid([-rem,-2], 12).prog(Action.Knit)

// wrist diagonal ribs
const diaRibs = prog.strToGrid(`
kkkppp
kkpppk
kpppkk
pppkkk
ppkkkp
pkkkpp
`);
prog.node(1).or(prog.node(2)).filter(s => s.countPrevWales() ===
      0).up(1).waleGrid(0:end, 8).tileMap(diaRibs, {
  k: Action.Knit,
  p: purl
});
```

Listing 9. *turtleneck dress.js*

Fig. 27. *Stitch program of the turtleneck dress*

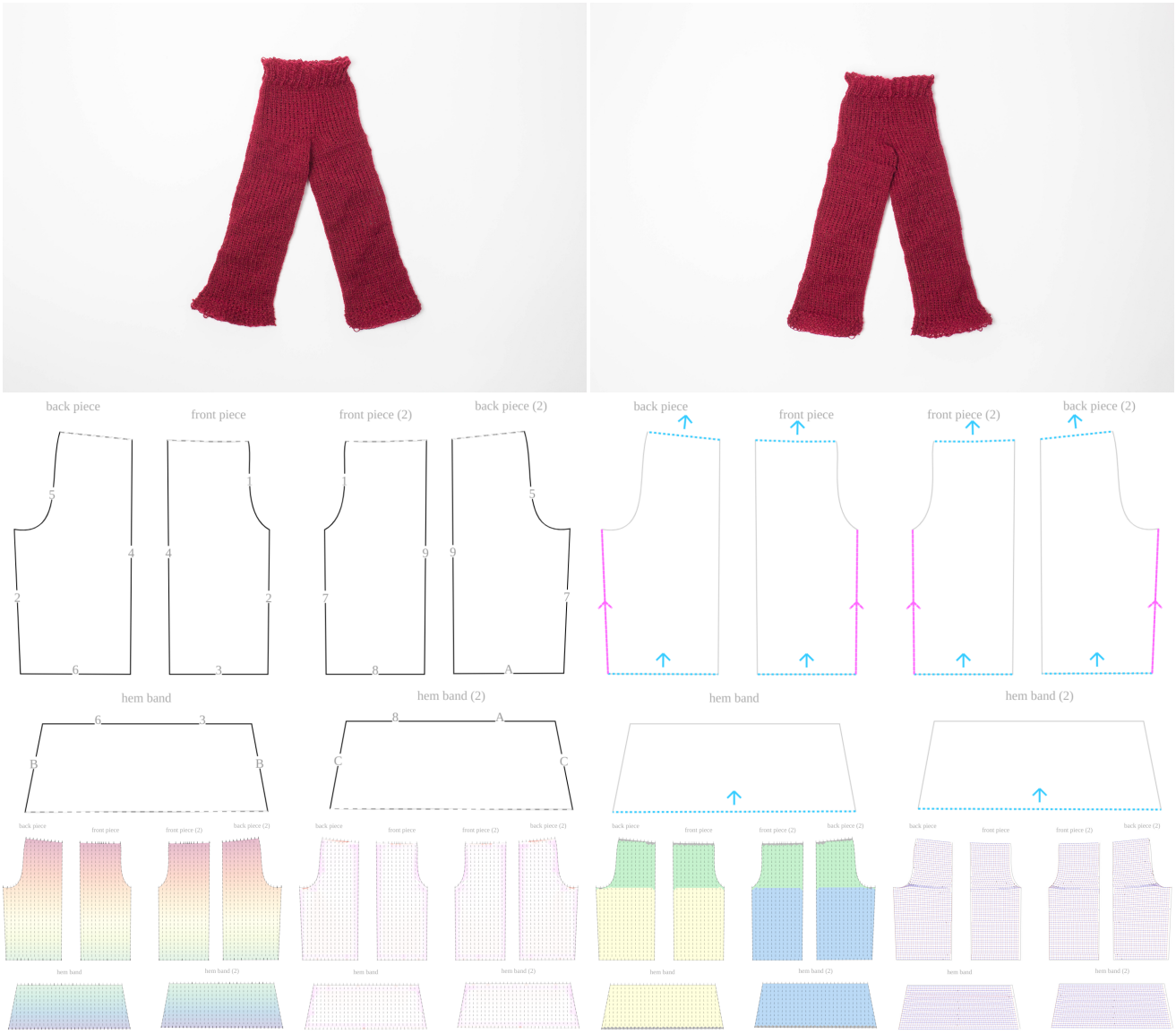


Fig. 28. *Wide trousers*

```
const purl = Action.register({
  pre: ({ k, n, rn }) => k.xfer(n, rn),
  main: ({ k, d, rn, cs }) => k.knit(d, rn, cs),
  post: ({ k, rn, n }) => k.xfer(rn, n),
  splitBySide: true
});
const ribLen = prog.lengthToWaleStitches('1 in');
prog.filter(s => s.countNextWales() === 0).waleGrid(0:end, -
    ribLen).tile(0b10, 2).prog(purl)
prog.filter(s => s.countPrevWales() === 0).waleGrid(0:end, 8).
    tile(0b1001, 2).prog(purl)
```

Listing 10. *wide trousers.js*

Fig. 29. *Stitch program of the wide trousers*

Fig. 30. *Sample evolution*