Garment Design Workflows for On-Demand Machine Knitting

by

Alexandre Kaspar

B.Sc., École Polytechnique Fédérale de Lausanne (2011)
M.Sc., École Polytechnique Fédérale de Lausanne (2014)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

Author .....
Department of Electrical Engineering and Computer Science
September 30, 2021

Certified by .....
Wojciech Matusik
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Garment Design Workflows for On-Demand Machine Knitting

by

Alexandre Kaspar

Submitted to the Department of Electrical Engineering and Computer Science
on September 30, 2021, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Modern computerized weft knitting machines enable on-demand production of custom, whole garments at once. They reduce the need for manual post-processing and generate minimal waste. Yet, their programming is still hardly accessible and is effectively done manually by few skillful knitting technicians. The programming of knitted garments typically involves scheduling hundreds of thousands of stitches. While every individual stitch created on such machines can, in theory, be controlled digitally, the ability to effectively do so depends heavily on the programming software being sufficiently accessible to the user. Unfortunately, current knitting software is typically closed and relies mostly on low-level programming. The lack of standardization and more accessible, higher-level user design tools effectively hinder the possibility of a digital, on-demand production of garments for all.

In this thesis, I explore the design space that flat-bed, weft knitting machines span and propose novel design workflows to enable accessible, digital customization of garments created on these machines. First, I introduce the inverse design problem of automatic knitting program generation from a single image, together with a machine learning framework that enables it. Second, I describe a parametric, primitive-based design tool that merges inspirations from both computer-aided design and pixel-based image editing. Finally, I propose a novel workflow to translate traditional, sketch-based garment patterns into knitting programs. The resulting system allows anyone to harness the plethora of existing garment designs while providing knitting-specific customization capabilities.

Thesis Supervisor: Wojciech Matusik
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

My journey through graduate school at MIT was initially chaotic as I explored diverse fields within computer science and digital fabrication. While this chaos may have been painful in the earlier years, I certainly learned a lot from it and it made the later, more focused part of my journey – on machine knitting – all the more enjoyable.

I am very thankful to my supervisor – Professor Wojciech Matusik – for the large academic freedom and support he gave me throughout my time at MIT. I am also thankful to my thesis committee for their complete availability as well as their respective research groups and students. Several of the first friends I made at MIT come from Professor Frédo Durand's computer graphics group. This includes notably my officemates in D410 – Lukas Murmann, Tzu-Mao Li and Zoya Bylinskii. Thank you all for the great company, the many discussions and the fun throughout the years. I met Professor Stefanie Mueller as she was still a PhD student and interviewing for her future position at MIT. Her job talk at MIT was an inspiration for me to stay on the digital fabrication side while I was slowly moving away toward crowdsourcing for machine learning in computer graphics and vision.

I am very thankful to Professor Neil Gershenfeld from the Center of Bits and Atoms (CBA) and all the students in his group including especially Jake Read and Sam Calisch. As I slowly progressed toward figuring out my main focus on machine knitting, I took two classes of his – "How to Make (almost) Anything", and its follow-up "How to Make Something that Makes (almost) Anything". Both classes solidified my interest in digital fabrication and tangible, physical artifacts. I then spent time as a teaching assistant for the first class, which brought me to Alex Zimmer and Carmel Snow through the textile recitation. Both have taught me a lot about textile and my own general lack of knowledge therein. Some students of the class – notably Molly Mason – brought key ideas that enabled the dataset acquisition for the first project of this thesis. Thank you for the unexpectedness of this part of my journey!

Some of my key collaborators actively worked with me on software development and taught me a lot through it: Kiril Vidimče for my initial work at MIT – the

Foundry project; and Tae-Hyun Oh for a large part of the first learning project of this thesis. Especially, thank you, Tae-Hyun, for your vision, mathematical background and help to make it happen when it was unclear whether it was even possible to start with. Then, I want to thank the other collaborators who made the knitting projects possible: Liane Makatura for her English prowess and her skillful figures spanning all my knitting projects; Yiyue "Alyssa" Luo for her help with machine knitting at critical times during a pandemic; Petr Kellnhofer for his complementary machine learning expertise and help; and Kui Wu for his textile simulation expertise and complementary knitting experience. Also, thank you to Mike Foshey for the general hardware help in the lab, and all my other labmates within the Computational Design and Fabrication Group at MIT.

I am very thankful to Jim McCann and his textiles lab at CMU. Without their precursor work on machine knitting, it is unclear where my journey would have ended. He and his students have all been very open to help and provided us with tools that made my transition to knitting much simpler.

I am very thankful to the MIT administrators who helped bring some form of order throughout the years: Bryt Bradley, Kshama Ananthapura, Mieke Moran and Roger White.

Thank you to my longest MIT roommate, Subra Sundaram, for being there throughout all the storms and sharing a passion about cats, especially Muschi.

Finally, I am very grateful to all the members of my family – both the Kaspar and Prior sides – and their unconditional support throughout all these years, especially the chaotic pandemic end. Thank you to Tom, Coleen, Jillian, Alissa, Jack and Clark. Thank you to my dad Charles-Henri, my mom Živka and my sister Laure for always being there, with the mountains, across the ocean, and bringing me calmness, lots of chocolate and lots of love.

Thank you foremost to my beloved wife Devin Prior and her support, loving presence through all the joy and storms of this long journey, and for the one that starts beyond. I would not have enjoyed this journey nearly as much without you.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Garments are so present in our daily lives that we often do not realize their underlying complexity and the labor necessary for their production. Garment manufacturing has gone through tremendous changes during the industrial revolution, and it has entered a new stage started with the digital revolution. The digitalization of manufacturing is leading a general shift from mass production models to on-demand production ones, with both aims of supporting personal customization and reducing waste. A notable example is the manufacturing revolution from 3D printing [134]. On-demand production of parts is now as easy as pressing on a button from our computer, and having them delivered to one's home within a few days [160].

Interestingly, industrial machines for additive manufacturing of whole garments already exist [150, 159], but they have yet to become more accessible for any similar revolution to happen (e.g., for local, customizable, on-demand garment production). While existing hardware [150, 159] theoretically enables such revolution already, current Computer-Aided Design (CAD) software and its corresponding Computer-Aided Manufacturing (CAM) counterparts are still underdeveloped and rely on primitive low-level representations that prevent fast iteration cycles between design, prototype and final product.

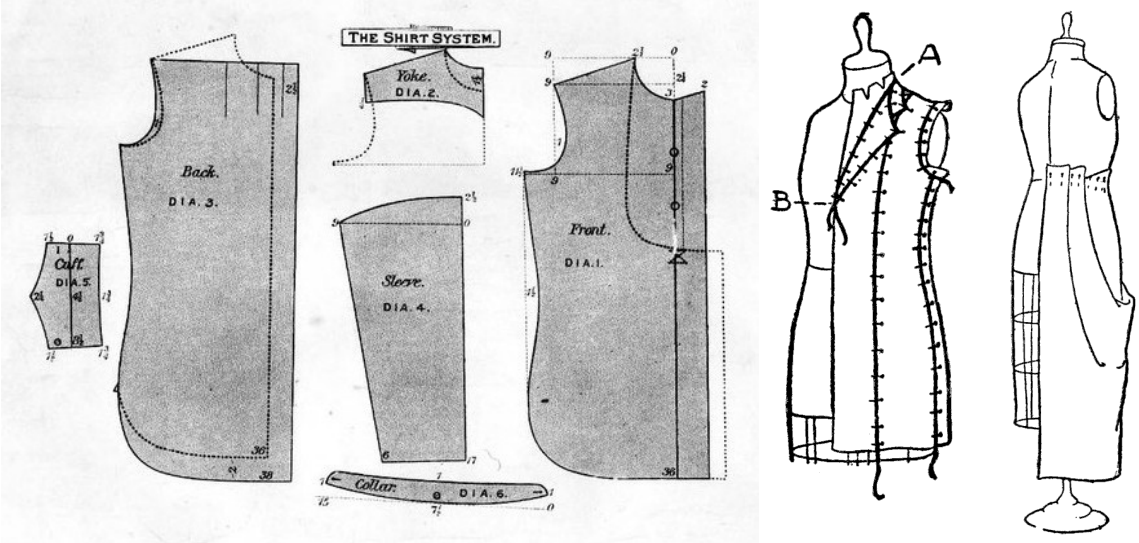


Figure 1-1: *Left*: a garment pattern from "The Cutter's Practical Guide" [175]. *Right*: illustration of two garments being draped with muslin fabric – originally figures 94 and 104 from the work of Conover [40].

1.1 Digital Garment Design

How we *design* something is essentially bound to how we *make* it. In the case of garment making, the traditional design workflow has long been tuned for woven garment production with the two major techniques being *pattern drafting* [67] and *draping* [99]. Pattern drafting refers to the art and technique that composes garment building blocks on paper to then transfer them to fabric panels that are cut and bound together (e.g., with sewing) to create the garment. Draping consists in the manipulation and adjustment of fabric pieces onto a 3D body shape such as a mannequin, folding and connecting pieces together with pins or other temporary binding tools. Both strategies – illustrated in Figure 1-1 – complement each other as modifications in one can be transferred to the other and vice versa. In practice, both manual and large-scale garment production similarly use garment blueprints to cut fabric panels and then bind them with sewing machines – the so-called cut & sew process. In terms of garment customization, while garment tailoring is still primarily relying on custom-made manual production, pattern grading [119, 148] is typically used to produce different sizes and fit variations to mass-manufactured garments.

The digital revolution has brought the advents of digital draping [147, 169, 176]. Digital garment authoring tools are now common to simplify garment design [25, 39, 113] and allow for a seamless transition between the digital space (digital customization, virtual showrooms and try-ons) and a standardized production.

Unfortunately, cut & sew production is hard to fully automate, because of the sewing component. Sewing machines already provide a layer of automation over manual sewing, but they still involve skilled workers. As a consequence, the production of garment has stayed geared towards mass-production to achieve lower production cost. This results in large waste and has led the textile industry to become the second largest polluting industry [9, 21, 145]. While the manufacturing of garments has become more eco-aware and large improvements have already happened over the past decades, the mass-production model is at the root of the problem and it is still the main active business model for garment production [125].

1.2 Computerized Machine Knitting

The focus of this thesis on machine knitting is motivated by the digital manufacturing capabilities of computerized weft knitting machines. In contrast to cut & sew that relies on human sewing panels of textiles together to produce garments, weft knitting machines (see Figure 1-2) can produce whole garments mostly automatically [150, 159]. Business opportunities have started to show up and companies [117] are trying to leverage computerized knitting for customization given new production models [129]. In parallel, lower-cost machines have been developed [142] and commercialized [146], opening the path to more accessibility.

Yet, while the hardware potential is there, knitted garment production is still essentially done through mass production. CAD and CAM for weft knitting has stayed focused on low-level, primitive design that results in slow iteration cycles between design and production, and proper digital customization is not quite possible yet. We argue that one of the missing pieces to harness weft knitting machines as tools for on-demand, additive manufacturing of textile is the existence of higher-level repre-

Figure 1-2: The "whole-garment" knitting machine used for the physical fabrication within this thesis – a 15-gauge model SWG091N02 from Shima Seiki [150].

sentations that enable faster design cycles and fully digital, accessible customization.

The two main challenges that lay ahead are about finding the proper design representation (CAD) and its corresponding translation mechanism to low-level machine code (CAM). Thus, this thesis attempts to answer the two following questions:

1. What should the high-level design representation for weft knitting be?
2. How do we translate the designs into knitting machine instructions?

1.3 Thesis Overview

This thesis tries to answer both questions by considering different design directions that encapsulate important, desirable capabilities of digital CAD software, while demonstrating each within a system that implements the translation from high-level design to low-level machine code (CAM), as summarized in Figure 1-3.

The first part of this thesis provides an overview of general textiles (Chapter 2), and then focuses on weft knitting machines (Chapter 3), their low-level programming and the common computer representations of knitted topology.

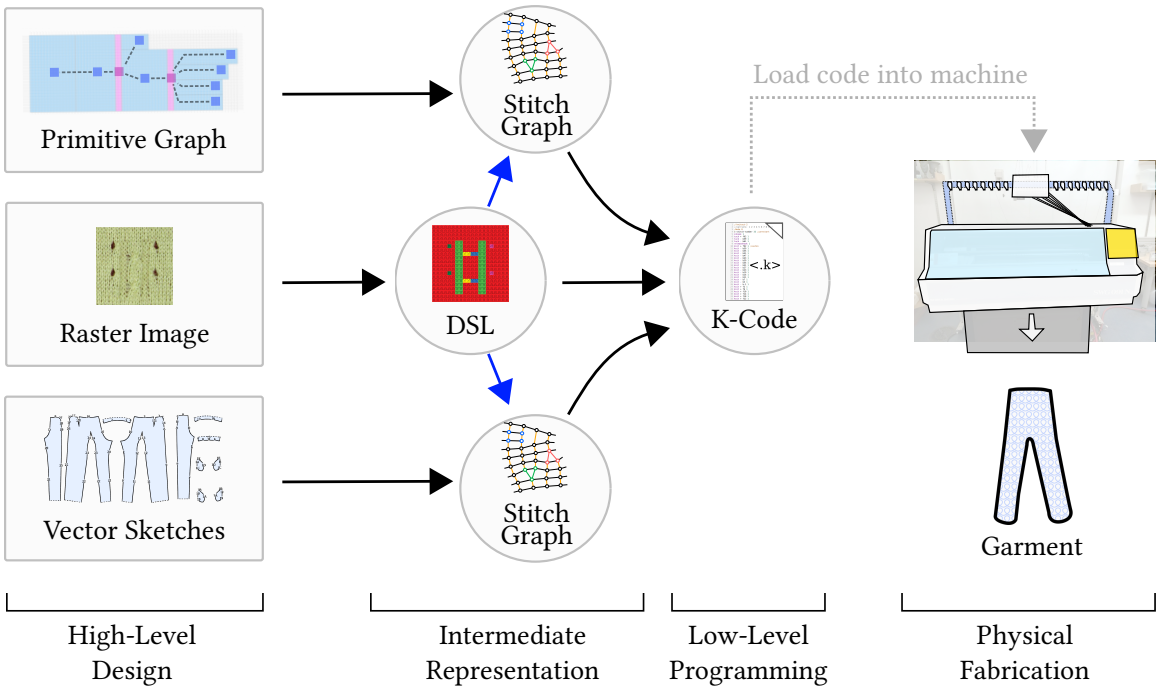


Figure 1-3: Overview of the space covered by this thesis: we propose three high-level CAD systems (left) that translate high-level designs into intermediate representations for knitting (center) and then generate low-level assembly code for knitting (right). That code is eventually loaded on the knitting machine to produce the corresponding physical artifact. The solid, **black arrows** are parts of the CAM components. The solid, **blue arrows** denote that a representation can be used by other intermediate representations. The dotted line is the transfer of low-level code to the machine.

The second part proceeds with the description of three complementary systems and their corresponding design spaces, illustrated in Figure 1-3. Chapter 4 start by introducing the inverse design problem with the intent of uncovering regularities in the knitting program space. Since the general inverse problem is beyond the scope of this work, we instead focus on the sub-problem of stitch pattern recovery given a single image as input. A novel learning framework to harness mixed-data is mathematically formulated together with the necessary data to instantiate it successfully. Chapter 5 considers the forward design problem in a bottom-up approach. We describe a set of simple parametric shaping primitives and the space of garments that they span through composition. We further introduce a layer mechanism to use the knitting patterns generated from the previous chapter and evaluate the customization capabilities with non-expert users. In Chapter 6, we propose a means to translate

the traditional cut & sew workflow for weft knitting machine programming. We introduce new knitting-specific annotations and optimize for the stitch topology while considering the inherent tradeoff between accuracy and simplicity. We conclude in Chapter 7 by summarizing the main insights from the development and usage of each of those systems, and highlight the remaining areas where user customization is still critically needed for accessible, on-demand production to flourish.

Thesis Contributions

The general contributions of this thesis are:

- High-level design concepts that enable simpler and faster design,
- Customization through parametric design,
- A simplified, initial work for knit technicians, and
- Enabling designers to be a more integral part of the full knit design process.

This thesis makes the following contributions, by system:

- Learning-based design (Chapter 4):
  - A set of unambiguous, high-level knitting pattern instructions for machine learning, together with an efficient data acquisition strategy.
  - A learning framework for harnessing both real and simulated data.
  - A system implementation that showcases successful knitting pattern program recovery from single images.
- Primitive-based design (Chapter 5):
  - A set of parametric knitting primitives for customizable composition of knitted garments.
  - A layer-based mechanism for stitch pattern customization.

  - A system implementation with a non-expert user demonstration of its customization capabilities.

- Sketch-based design (Chapter 6):
  - An optimization to interactively specify the knitting time process on an implicit garment manifold.
  - An algorithm for decomposing such time function into simple regions for knitting.
  - A hierarchical optimization that samples the stitch topology while considering the tradeoff between accuracy and simplicity.
  - A set of layer mechanisms for customizing garment patterns.
  - A subdivision mechanism to speed up computations based on stitch graphs.

Part I

Background:
From Fiber to Weft Knitting

Chapter 2

Textiles Background

In this chapter, we give an overview of textiles in general. While the rest of the thesis focuses specifically on weft machine knitting for the production of garments, being aware of the broader world of textiles is important. As we will see, a subset of textiles has received a lot more attention from the manufacturing revolution, mainly because it could afford automation more easily. Yet, the remaining more general forms of textiles have led to the development of specific branches of mathematics that are broadly useful, including for the automation and verification of textile production.

This overview starts by introducing the world of textiles, its complex historical context and highlighting the breadth of its applications (Section 2.1). We then consider the journey from fiber (Section 2.2) to textile and fabric (Section 2.3).

The structure and ideas presented in this chapter were initially developed as a recitation about textiles for the "How to Make (almost) Anything" class at the Center for Bits and Atoms at MIT[1]. The recitation was created together with two colleagues: Alex Zimmer and Carmel Snow.

[1] https://cba.mit.edu/

2.1 Context and Applications of Textiles

2.1.1 Textile, Fabric or Cloth

The word textile typically refers to a form of material that is created by interlacing yarn, thread or fiber. *Yarn* is a long intertwining of fibers, whereas *thread* is a type of yarn typically used for sewing. *Fiber* is a natural or man-made material that is longer than it is wide, giving it advantageous mechanical properties in a specific direction. In practice, the word "textile" is often used interchangeably with "fabric" or "cloth", although they tend to be used in slightly different contexts. Notably, *fabric* is usually not used as-is, but serves as a constituent of a larger piece – e.g., a garment –, whereas textile can be used as-is such as carpets and rugs.

The etymology of the words brings a more intuitive take on their differences: "textile" is borrowed from latin *textilis* ("woven"), derived from *texere* ("to weave") [16], whereas "fabric" is borrowed from French *fabrique* or latin *fabrica* ("the framework or basic structure of anything") [5].

2.1.2 Historical Context and Importance

Textile has a long history, believed to have started more than 30000 years BCE [38, 96]. It is highly tangled with the socio-economical context of the region where it is developed as it involves many layers of the society for its production, its trade and processing [131, 182]. As one of the main economic driving forces during the industrial revolution, textile production further plays an important role in the integration of women as part of the modern workforce [50, 76].

2.1.3 Applications Areas

Textiles are everywhere in our lives. It is the main form of material used for all of our garments. It also covers many of our home floors, furniture, beds and even the interior surface of our cars (see Figure 2-1). As a simple thought experiment, we consider the Materials in Context Database [18]. Out of their 23 material categories, two are

Figure 2-1: Example of home interior from Bell et al. [18] including various forms of textiles: carpet, rug, cloth on sofa and pillows, table cloth, tissue, window curtains, teddy bear and elephant plush toy. Credits: Dana Moos, CC-BY-NC 2.0.

clearly related to textile: fabric and carpet. Their cumulative coverage in terms of patches accounts for more than 16% of the 2,996,674 material patches they compiled out of 436,749 images. Fabric itself gathers the third most prominent number of patches ($\approx 12\%$) behind wood ($\approx 19\%$) and painted material ($\approx 16\%$).

Beyond garments, fashion, interior design or automobile, textile also intervenes in often unexpected ways. In aeronautics, textile is responsible for the sails and spinnakers of boats [29], the envelope of hot air balloons, the wings of paragliders and deltas, as well as parachutes. In architecture, textile can be used as a scaffold for casting complex 3D shapes with concrete [173].

A growing part of textile is dedicated to *smart textiles* [171] that can enhance human performance or safety during critical operations, including fire safety [93], health monitoring [183], medical tissue engineering [184] or even spacesuit applications [126].

Finally, smart textiles have also started tackling simpler user applications and tangible interfaces by integrating function in the fabric [132] or within the fiber itself [61, 88, 109, 110, 136, 163].

Figure 2-2: Examples of natural fibers, from left to right: sheep wool (animal), cotton (plant) and absestos with muscovite (mineral). In the public domain, respectively from: Bernard Spragg, the US Department of Agriculture, and Aram Dulyan.

2.2 From Fiber to Yarn

Fiber is the base unit of all textiles, and notably the yarn typically used for producing fabric and garments. However, not all fibers are necessarily used for textile. A notable example is *paper* that is typically made by processing fiber in water, draining the water and then pressing and drying the resulting material [75]. In this section, we briefly cover the types of fibers and their differences. For a more extensive look at the broad world of fibers, see the work of Kadolph [85].

2.2.1 Types of Fibers

Fiber can be natural or man-made. Most *natural fibers* come from animals or plants, although some forms are created from geological processes, each of which is illustrated in Figure 2-2. Common examples include wool, silk, cotton or flax. On the other side, *man-made fibers* are fibers that undergo a significant modification during production. They are typically subdivided into *semi-synthetic* – when starting from a fiber-like raw material that is only partially modified –, and *synthetic* – when starting from synthetic material. The majority of *semi-synthetic* fibers are based on cellulose [180], which includes those used for paper making. Common examples of *synthetic* fibers include nylon, acrylic and elastane. Notable examples include carbon fiber [37], optical fiber [36], fiberglass [177], as well as metallic fibers typically used in electric cables. Figure 2-3 shows examples of both nylon and carbon fiber.

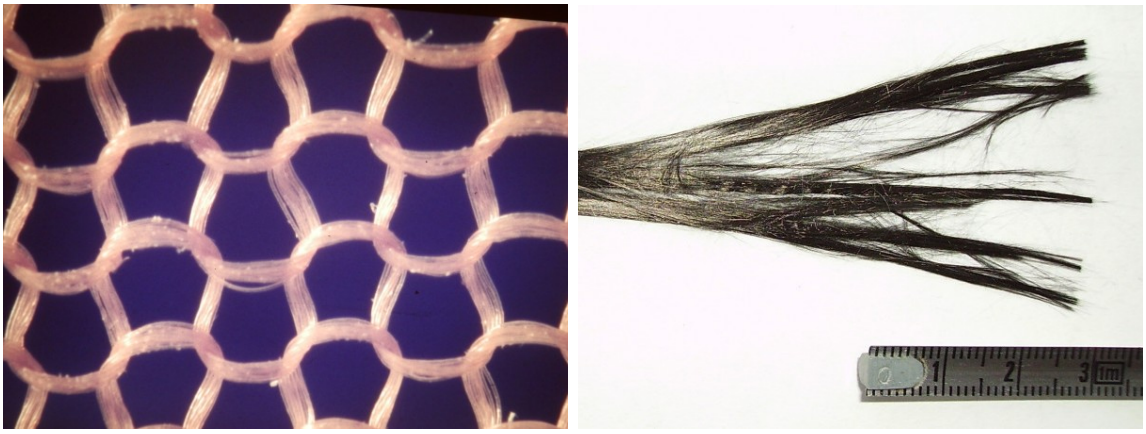


Figure 2-3: Examples of synthetic fibers: nylon (left) and carbon fiber (right). Credits: Vigorini, CC-BY 4.0 (left); and in the public domain via cjp24 (right).

Figure 2-4: *Left*: the two different twisting directions, often called S and Z twists for the patterns they produces. *Right*: skeins of yarn and a close-up of their plies.

2.2.2 Fiber Processing

The transformation from fiber to yarn can involve many different steps. As a common example, cotton processing includes a large sequence of operations [59]. Raw cotton balls undergo several pre-processing steps including the extraction of its fiber components (*ginning*), and their cleaning, disentanglement and intermixing (*carding*). Then, the fiber bundles undergo *spinning* during which the strands of fibers are twisted and eventually wound onto a bobbin to form the yarn. Strands of yarn are typically called *plies* and they are often combined by twisting them together. This twisting is typically done in the opposite orientation from that of the fiber in the individual plies so as to create a *balanced yarn* that does not twist upon itself. Figure 2-4 illustrates the twisting of yarn plies.

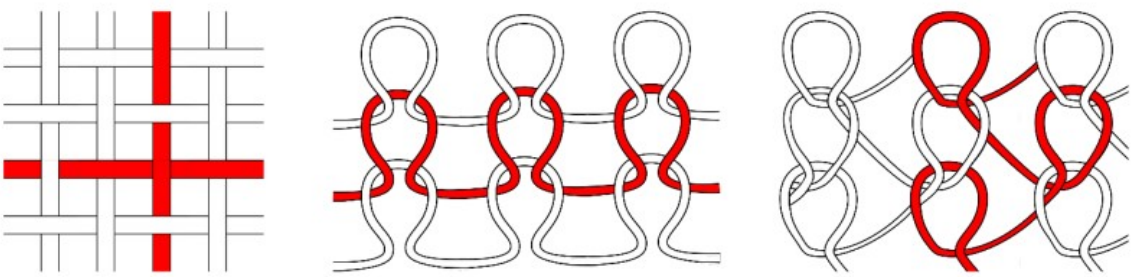
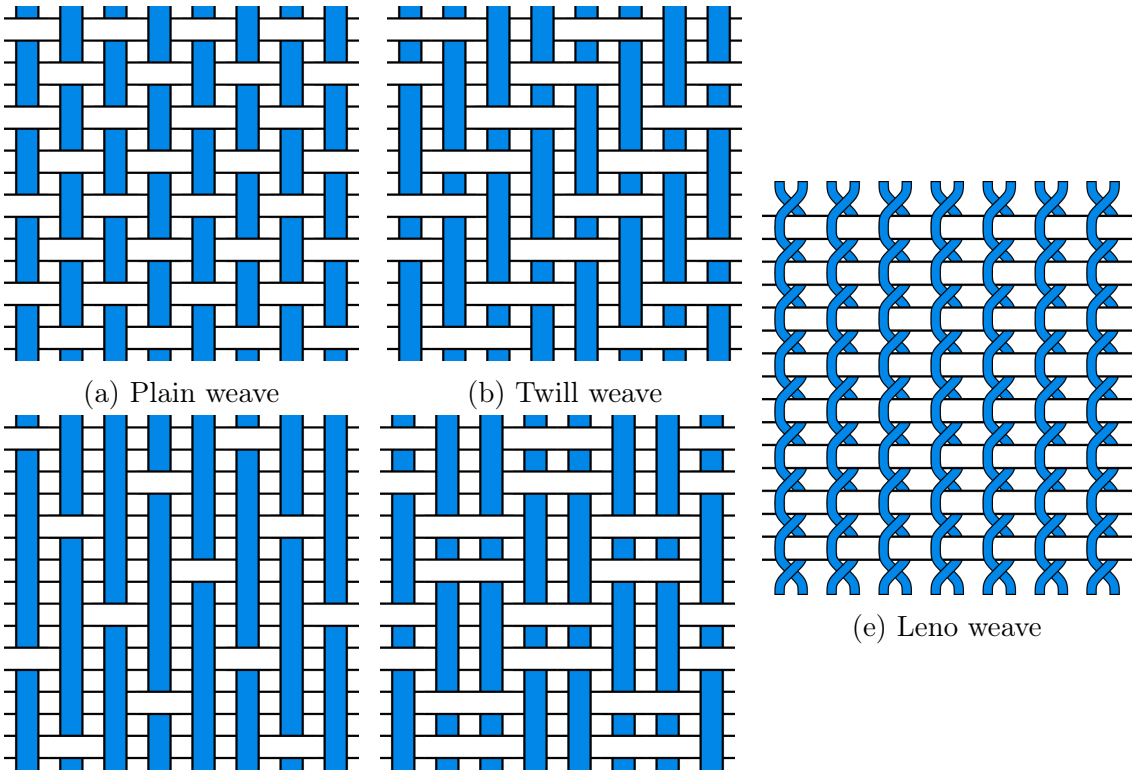


Figure 2-5: Three of the most common textile topologies: woven (left), weft knitted (center) and warp knitted (right).

At the other end of the spectrum, a group of synthetic fibers that is gaining interest in smart textiles are *monofilament* fibers [65]. Two fabrication methods include extrusion of melted material (e.g., nylon, PLA [31] and recycled PET [12]), and pulling on a material preform that is slowly melted (e.g. optical fiber). More recently, those fibers are starting to integrate additional components inside of the filament structure including electrodes [64], diodes [136] and even full micro-controllers [107].

2.3 From Fiber to Textile and Fabric

The two most common forms of fabric in garment-making are *woven* (Section 2.3.1) and *knitted* (Section 2.3.2), illustrated in Figure 2-5. Their prevalence is in part due to their regular structures that afford large-scale automation. Correspondingly, they form the majority of mass-manufactured textiles for garment production. More general textile categories tend to have larger degrees of freedom that make them less amenable for automation, notably: *crochet* (Section 2.3.3) that uses similar constructions as knitting, yet with very different mechanical properties; *braiding* (Section 2.3.4) that encompasses woven structures; and *knotting* (Section 2.3.5) that encompasses both crochet and knitting. Fabric binding with *sewing* (Section 2.3.6) is then discussed given its importance to garment production, followed by *tufting* (Section 2.3.7) that works with a similar, yet simpler thread insertion principle. We briefly mention so-called *non-woven* textiles (Section 2.3.8) and conclude with the notions of *nap* and *pile fabric* (Section 2.3.9).



(a) Plain weave (b) Twill weave (e) Leno weave

(c) Satin weave (d) Basketweave

Figure 2-6: Examples of common woven patterns.

2.3.1 Weaving

Woven fabric is composed of two sets of orthogonal yarns – the weft and warp yarns – that interlace to form a sheet of fabric [3, 15], as illustrated in Figures 2-5 and 2-6. Although weft and warp yarns look similar locally, they play distinct roles. The *warp* yarns form parallel, independent tracks that typically do not directly interact, whereas the *weft* yarn goes back and forth, orthogonal to the warp yarns.

Different types of woven patterns – also known as *weaves* – have received specific names over time. The most common weaves are illustrated in Figure 2-6: the *plain* weave that alternates under-over as a checkerboard; the *twill* weave that forms a diagonal pattern and is used for denim fabric typical of jeans; and the *satin* weave whose warp threads float over four or more passes of weft thread, producing a glossy and smooth material. The *basketweave* is a common variation of the plain weave that creates a criss-cross pattern by using larger checkerboard tiles. An important

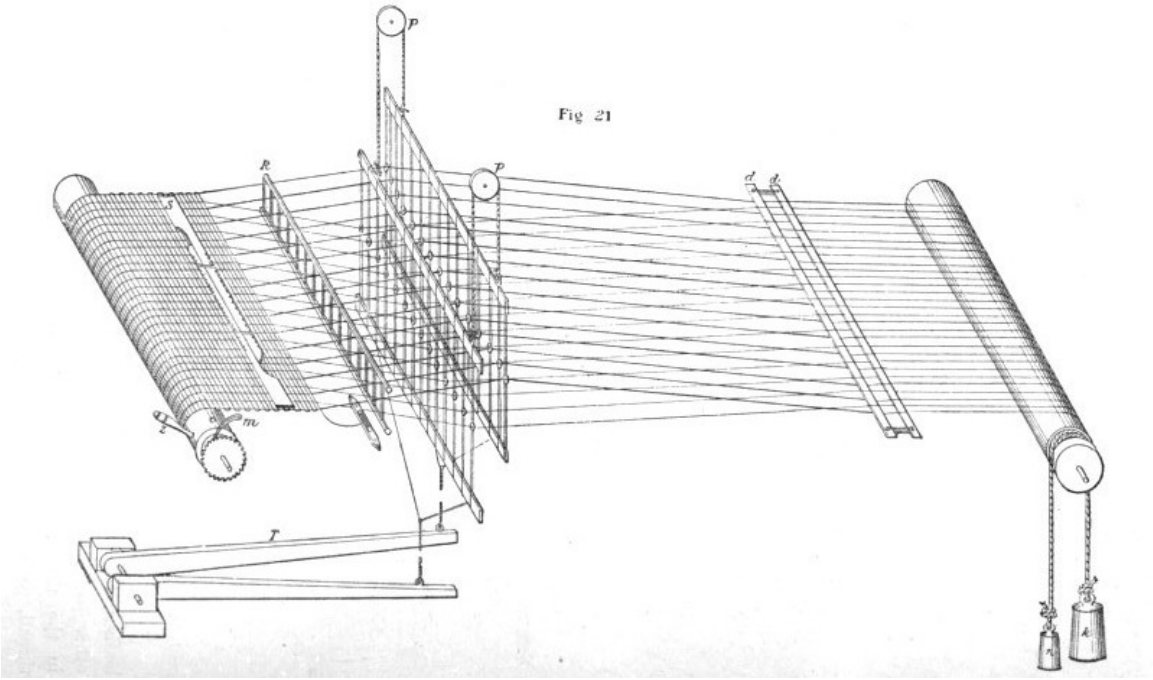


Figure 2-7: Illustration of common loom mechanisms. Originally figure 21, page 78 of the work of Barlow [15].

specialized weave is the *Leno* weave – also known as *cross* weave – that intertwines pairs of adjacent warp threads around the weft yarn to produce a strong yet airy fabric [57]. It is often used for creating sturdy bags, nets and medical gauze.

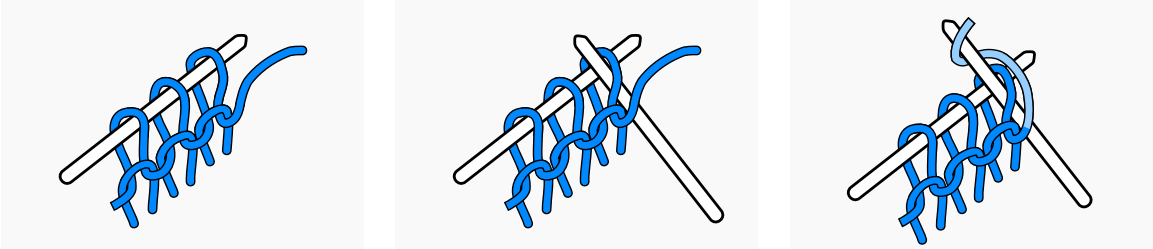
Looms are devices used to create woven textile, whose general mechanisms are illustrated in Figures 2-7. They keep the warp thread under tension and often allow some form of control over the warp yarn selection to simplify the path of the weft yarn. Important mechanisms in looms include: (1) the *shedding* that raises a selection of warp yarns, forming a *shed* that allows the passage of the weft yarn; (2) the *weft insertion* that takes care of transferring the weft yarn across the width of the fabric, historically done with a *shuttle* that carries it both ways; and (3) the *beating-up* that compacts the weft yarn after each of its passes.

One of the critical components of looms for automation is the mechanism behind the selection of the warp yarns, which may allow the user to program different woven patterns. The common mechanism for shedding in Figure 2-7 relies on *heddles* – small eyelets that let the warp yarn through, and can be raised or lowered mechanically,

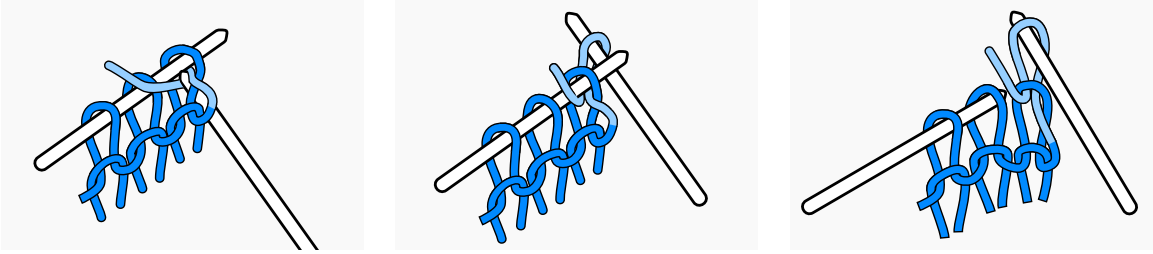
or made to twist around each other (e.g., for Leno fabric). How the machine selects the heddles to be raised defines the pattern programming capabilities. In 1804, Joseph Marie Jacquard integrated ideas from Basile Bouchon, Jean-Baptiste Falcon and Jacques de Vaucanson into a loom attachment that allows programmatic selection of heddle groups with punch cards [15]. As a chain of punch cards advances with the weaving process, only the heddle groups for which holes exist in the current punch card get selected (or not). Programming the weaving pattern became thus as simple as creating portable punch cards. Beyond looms, this automation work notably led to the *Analytical Engine* of Charles Babbage, and with it, the rest of the modern computers [51].

The first powered looms used a *shuttle* that carries the weft yarn spool across the fabric's width in a *continuous* back-and-forth manner. Modern looms use various mechanisms that can achieve much higher throughput – i.e., a larger number of *picks* per minute of the weft yarn across the fabric's width. This includes notably: *airjet* and *waterjet* looms that propel the weft through with compressed air (or water) bursts, *rapier* looms that mechanically grasp and carry the weft yarn across the shed before retracting without it, and *projectile* looms that propel an object bound to the weft, then separated and carried back mechanically. All these mechanisms have in common that the weft thread is *discontinuous*, and typically cut to be slightly longer than the fabric width. One modern exception consists of so-called *narrow fabric looms* such as *needle looms* used for making ribbons, belts and various types of tapes [164].

An important part of the woven fabric is the *selvage* that corresponds to the most lateral edge of the fabric which prevents the fabric from fraying. In *shuttle weaving*, the selvage can be as simple as the ends of the weft thread past the most lateral warps – i.e., by alternating the selection of the end warps so they are caught by the weft yarn as it goes back and forth. In modern *shuttleless weaving*, dedicated mechanisms are employed such as *fused selvage* that uses temperature to bind the fabric (notably with termoplastic fibers), *leno selvage* that binds the weft with additional small threads and twisting, or *tucked-in selvage* that tucks the extremities of the discontinuous weft edges back into the fabric (producing a double weft density near the edge).



(a) Initial state (b) Active needle insertion (c) Formation of new loop



(d) Active needle retraction (e) Extraction of new loop (f) Dropping of old loop

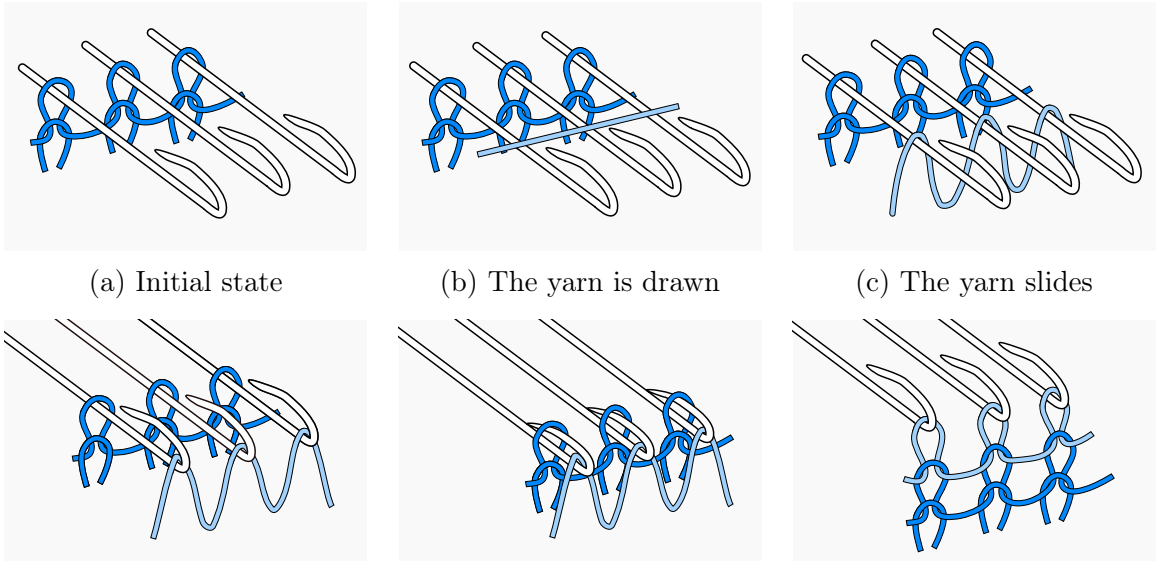
Figure 2-8: Creation of a knit stitch with two knitting needles. When the active needle retracts – from (d) to (e) –, its endpoint slides closely around the other needle to pull the new loop through the old one.

2.3.2 Knitting

Knit fabric is formed by pulling loops of yarns through previous existing loops, eventually forming rows and columns of stitches that are interconnected with each others. Figure 2-5 illustrates two forms of regular knitted topologies: *weft* knitted fabric forms rows of loops with a single yarn thread, whereas *warp* knitted fabric forms columns of loops with parallel yarn threads. The terms weft and warp naturally match the thread directions of weaving. For a complete review of knitting technologies, see the work of Spencer [158].

Hand Knitting

Knitting by hand is typically done with two ore more needles that stack sequences of stitches – i.e., the individual loop units in knitting. Two needles interact to create a new loop that is pulled through a pre-existing loop that drops from the holding needle while keeping the new one on the active needle, as illustrated in Figure 2-8. A variety of *cast-on* techniques exist for creating stitches that do not depend on previous



(a) Initial state (b) The yarn is drawn (c) The yarn slides

(d) The hook catches the yarn as the needle retracts (e) The hook is closed to knock the old loop over (f) The new loop is pulled through the old one

Figure 2-9: Beard needles and their actuation to form new knit stitches

stitches and are necessary to start the knitting process. Similarly, techniques exist to close the knitted structure – i.e., known as *bind-off* or *cast-off* procedures. By using either circular needles with double ends, or relying on more than two double-ended needles, then one can *knit in the round* – i.e., knit tubular structures. The individual rows are then replaced by a spiral-like structure.

Flat Weft Knitting

One of the first step to the textile industrialization was the invention of the *Stocking Frame* – the first form of knitting machine – by William Lee in 1589. It used a flat bed containing a parallel set of *beard needles* laid out to hold stitches. The basic weft knitting process goes as follows: (1) a yarn is carried over the bed and gets caught in the hooks as the needles get actuated; (2) the old stitch loops get knocked over their needle hooks as these are closed during the needle retraction; (3) this forms new stitches by pulling the new loops through the old ones as they drop from the needles. By repeating the process, large sheets of weft knitted fabric can be created quickly. This stitch creation process is illustrated in Figure 2-9.

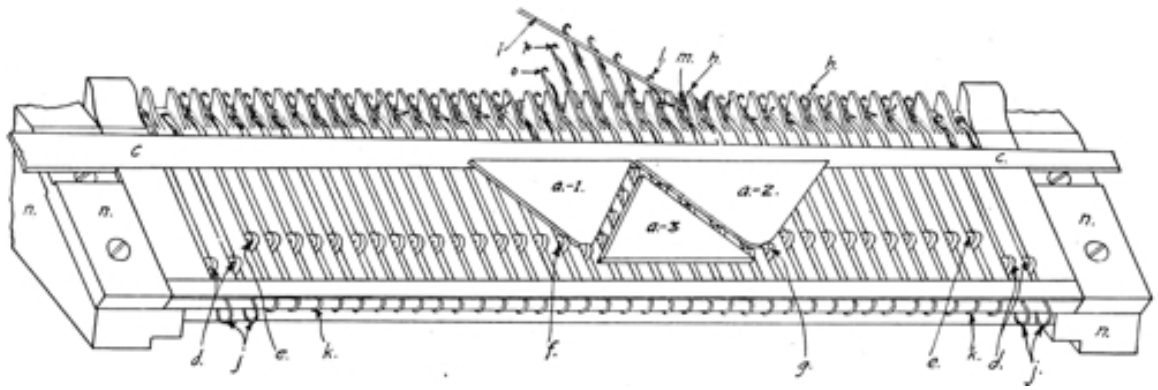


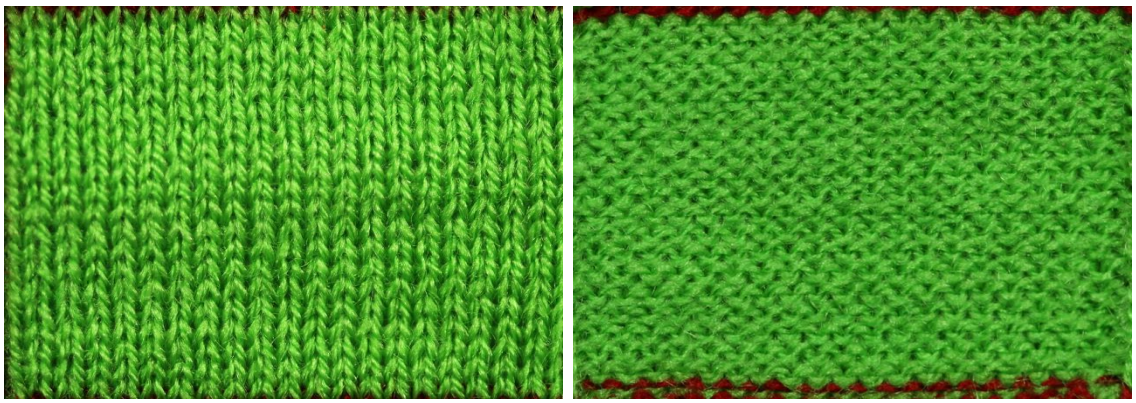
Figure 2-10: Schematics of the actuation of a flat bed machine with a mechanical cam. Originally figure 16 from the work of Buck [26].

Mechanically, beard needles have a hook that can partially flex and close under mechanical pressure. Other more recent needles mainly change the actuation of the needle and its hook closing mechanism. Notable ones include the *latch needle* that closes the hook with a mechanical latch, the *compound needle* and the *slide needle* that both use an additional linear sliding component that acts as a hook closure.

Domestic flat bed machines typically have a manual carriage that actuates the needles through a set of mechanical cams as illustrated in Figure 2-10. The carriage may be programmed to choose a sequence of needles to select or their respective actions. More complex flat bed machines may include additional beds. A secondary bed allows for complex stitch patterns including *purl* stitches – i.e., the back of a knit stitch, which looks very different, as illustrated in Figure 2-11. Industrial flat bed machines typically have at least two beds facing each others and can knit tubular structures by forming cycles that cover both beds.

Circular Knitting

Circular knitting machines have a fixed circumference that is packed with needles. Specific needles can be selected and actuated back and forth to produce notably the heel of a sock. Such machines typically target socks, although sleeves can also be made. While flat bed machines with two beds can also knit tubular structures, they tend to be slow. The speed limitation is due to their acceleration profile: since the



(a) Front of jersey fabric (b) Back of jersey fabric

Figure 2-11: The front (left) and back (right) of a basic jersey fabric highlighting the distinct appearance of both *knit* stitches and *purl* stitches respectively.

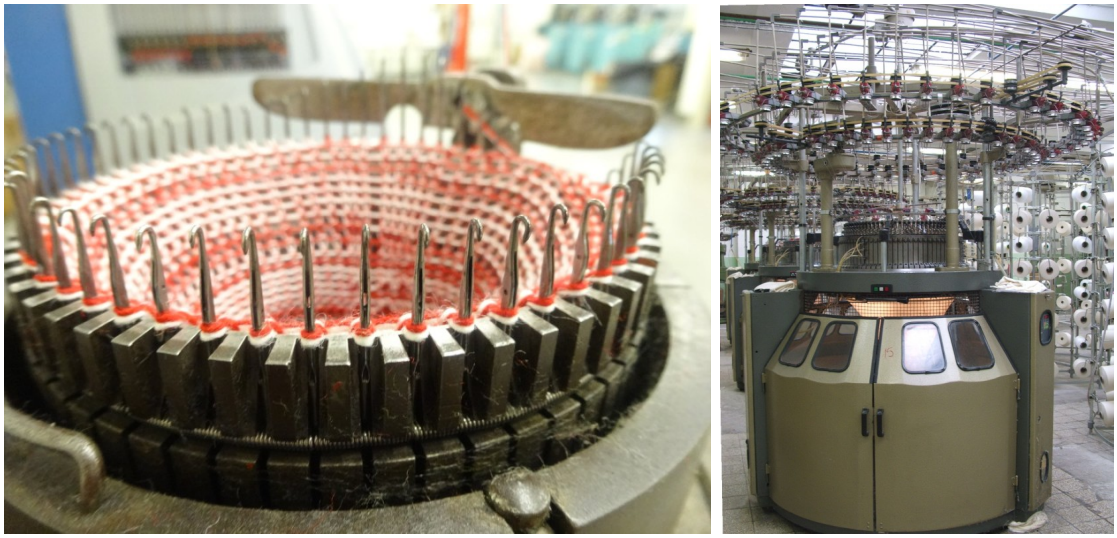


Figure 2-12: Example of hobbyist circular knitting machine that is manually actuated by rotating a shaft (left), and a large industrial circular knitting machine (right). In the public domain thanks to Elkagye.

carriage alternates between going left and right, it keeps accelerating and decelerating. In contrast, circular knitting machines do not have to decelerate and can knit at a constant rate when their yarn continuously rotates in the same direction. This makes them the machine of choice for high-throughput knitted fabric production as they achieve the highest throughput. They typically produce the knitted fabric used for making t-shirts and other knitted garments. Figure 2-12 illustrates both an older design for sock knitting, and an industrial high-throughput circular machine.

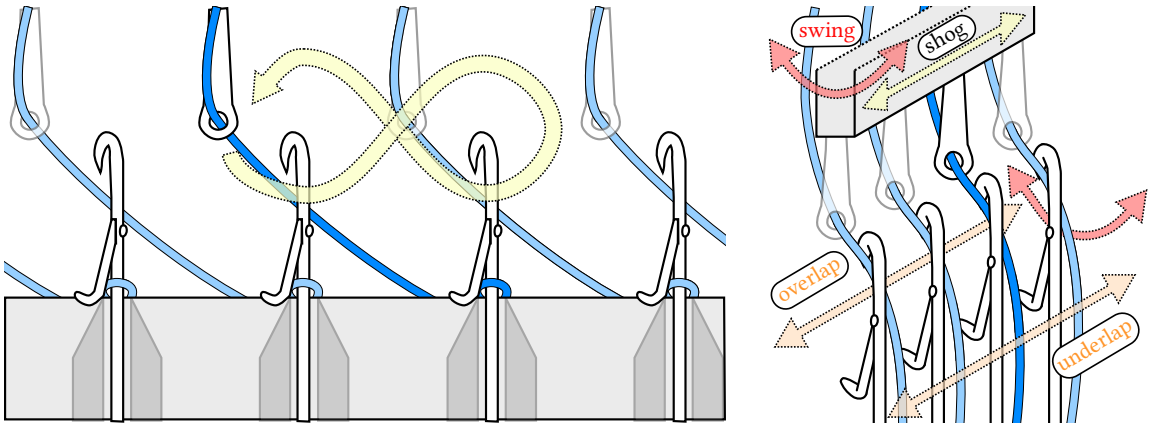
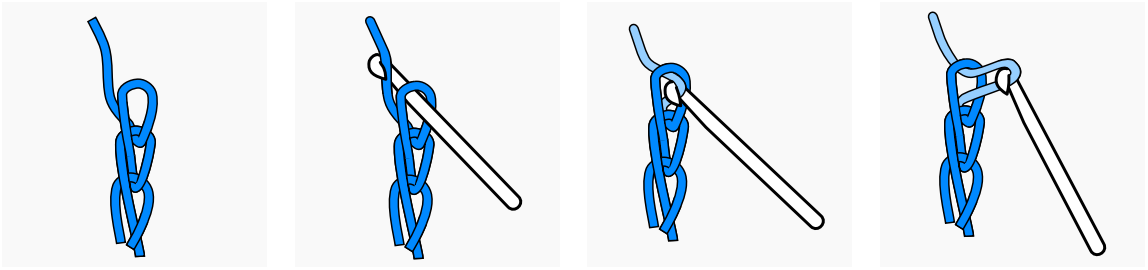


Figure 2-13: An example of swaying illustrating the movement of the yarn guides in a Raschel warp knitting machine: the general movement from the front of the needles (left), and the movement decomposition from the side, behind the needles (right). The general *lapping* movement is decomposed into *swing* and *shog* axes, and its front and back passes relative to the needles are called *overlap* and *underlap*.

Warp Knitting

The last category of knitting machines are *warp knitting* machines that work with many warp threads in parallel. Mechanical guides bring the yarn into the needles by swaying laterally in a *lapping* movement that can be decomposed into both a lateral motion parallel to the needle bed – known as *shogging* or *shog* – and a back and forth motion between the front and back of the needles – the *swing*. As the needles retract, their hooks are closed to lets the old stitches get knocked over, before extending the needles and opening them again, to repeat the process. Note that the shogging of the guides must allow for warp threads to reach more than a single needle, otherwise we would end up with individual, separate stitch columns. This is visible in Figure 2-5 where pairs of adjacent wales are connected and Figure 2-13 that illustrates the swaying of the yarn guides in a Raschel warp knitting machine. The swaying itself is programmable over time and allows for the creation of various knitting patterns.

Typical warp-knitted fabrics include lace, tricot as well as stretchy fabric used in athletic wear. An important functional difference with weft-knitted fabric is that yarn damage in a warp-knitted fabric does not trigger large-scale unravelling. In terms of manufacturing, all the needles work in parallel which allows for very fast production. The supply of yarn is very similar to that of warp threads in a weaving loom.



(a) Initial loops (b) Catching (c) Pulling (d) New loop

Figure 2-14: The formation of a stitch in crochet: the hook needle is inserted through a loop to catch the yarn and then pulled back through the loop to form a new loop.

2.3.3 Crochet

Crochet is a form of loop building that works with a hooked needle [28], one stitch at a time, without having to keep previous stitches on another needle. Compared to knitting, the yarn can be pulled through any previous stitch easily, which enables less structured forms of knitted fabric, and is potentially more accessible. Correspondingly, stitch loops are typically not kept open for long, but instead closed directly. From a mechanical perspective, crocheted fabric tends to stretch less than knitted fabric, and it does not necessarily unravel widely upon local yarn damage. As a less structured fabrication process, it affords less automation and is still mainly manual. On the other side, the wide degrees of freedom allow for the creation of very complex 3D shapes such as with hyperbolic crochet [69].

The simplest stitch of crochet is the *chain stitch* that catches yarn through one loop to create a new loop, as illustrated in Figure 2-14. The action of catching the yarn by moving it over the hook is called a *yarn over* and is a component of more complex stitches together with the action of *drawing the loop* by pulling it through another loop.

Tunisian Crochet

A special form of needles used in Tunisian crochet – also known as Afghan crochet – works by stacking loops on the hook needle. The corresponding hook needle is typically longer and has an end that prevents stitches from going through.

Figure 2-15: Illustrations of braids: braided Swiss bread (left) and USB cable (right).

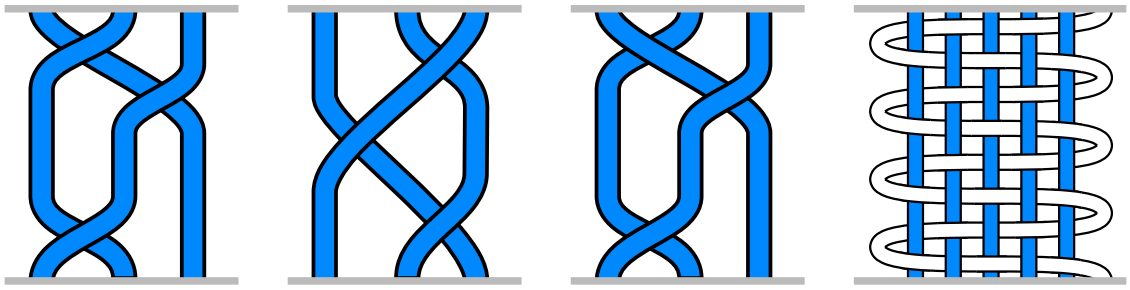
2.3.4 Braiding

A braid is the interlacing of two ore more strands of flexible materials. Common examples include: hair braids, ropes that braid multiple yarns together to prevent twisting under load, and various types of bread such as the *Zopf* (known as "Tresse" in French, which means *braid*) or the Jewish *challah* bread – as shown in Figure 2-15.

In the industrial setting, various forms of metallic braiding are often placed around electronic cables to shield them from electromagnetic interference. Material composites use braiding to increase mechanical properties and sometimes form the composite itself [13], whereas braiding serves for the formation of complex freeform materials in architecture [174]. Braiding machines typically use cyclic motions of bobbins to intertwine yarn or other composite materials together. A notable example of industrial braiding is with carbon fiber and other composites in the aerospace industry.

Braid Theory

In mathematics, braids play an important role in group theory [34, 120] with the Artin braid groups. An important underlying problem is whether two braids of N strings are topologically equivalent – i.e., if they represent the same interweaving, modulo some free movement of the N threads while keeping their ends fixed. The theory behind Artin braid groups has recently been used by Li and McCann [100] to verify the tangling properties of knitted transfer operations on weft knitting machines. They provide a means to verify the validity of a transfer sequence given the source



(a) 5-braid (b) Equivalent (c) Distinct (d) Plain weave

Figure 2-16: Illustration of geometric braids: the two on the left are topologically equivalent, whereas the third from the left is distinct due to its different ordering from the leftmost one. The rightmost example showcases a plain weave as a braid.

and target stitch locations on the needle bed. Finally, from the geometric perspective, traditional weaving can be considered as a specific case of braiding with one braid acting as the weft thread[2], as illustrated in Figure 2-16.

2.3.5 Knotting

Knots are loop structures that exhibit a form of tangling that cannot be undone without passing one or both ends of the material backward through its loops – i.e., effectively undoing the knot. From a structural perspective, knots stabilize yarns or strings and are typically used for binding things together such as with cordage on sailboats. Knotted fabric is created by forming webs of knots, such as in net making.

A common form of knotted textile is *macramé*, in which multiple parallel yarns are braided and knots are formed locally to rigidify the structure. Macramé textile is self-supporting when put under tension so that it can hold objects tightly (e.g., the plant pot in Figure 2-17). The main knots of macramé are *square* knots (also known as *reef* knots) and various forms of *hitch* knots that connect different yarns locally.

Knot Theory

The mathematical study of knots deals with their topological aspects [2] and is closely related to *braid theory*. While a mathematical knot represent a closed curve, *links*

[2]This assumes a continuous weft thread, which is not common in modern weaving looms.

Figure 2-17: Macrame examples: as a flat sheet (left) and a net wrapping around a plant pot (right).

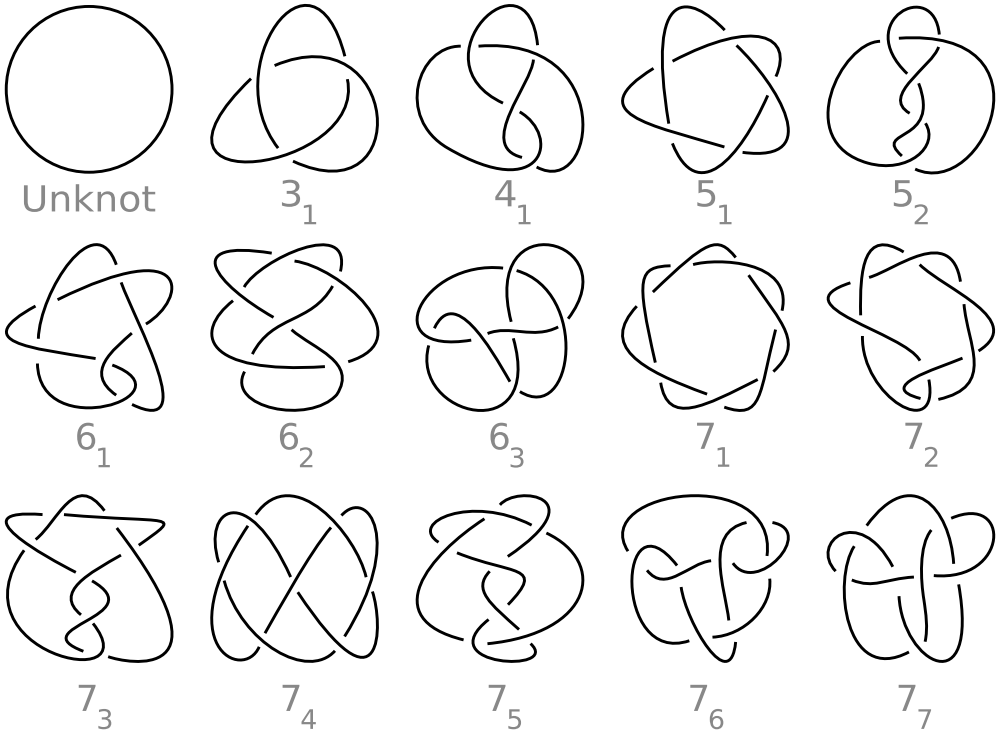


Figure 2-18: Mathematical prime knots up to 7 crossings with their Alexander–Briggs notation, excluding mirrored versions. In the public domain, created by Jkasd.

represent collections of knots that may be tangled together, and *braids* can be transformed into links by binding their ends. Some mathematical operations are used similarly on knots, links or braids such as Reidemeister moves [165] that transform a knot

Figure 2-19: A Brother machine for both sewing and embroidery – note the linear gantry (left) – and a close-up looking at the second bobbin that provides the thread below the fabric (right). The main thread comes from above, through the needle.

(or braid) into another, topologically equivalent form of it. Figure 2-18 visualizes some prime knots (i.e., that cannot be decomposed under the *knot sum* operation [114]).

In contrast to practical knots, mathematical knots are closed. By cutting them topologically, one can represent knotted textiles. Knitting can be represented as the composition of an interlocking series of slip knots [112].

2.3.6 Sewing

Sewing is mainly used to bind objects together using stitches made with a sewing needle and thread. While it is extensively use for binding fabric in garment production, it can be used to bind other materials such as leather, or even books.

Sewing machines speed up the binding process by taking care of the stitch creation process automatically, leaving to the user the work of guiding the machine path and fabric tension [6]. Modern machines can often use a collection of different stitch types that depends on the number of threads and needles used by the machine. Common hobbyist sewing machines typically use two threads: one passing through the needle, and one stored in the bobbin case below the feed dogs (shown on the right of Figure 2-19). Common stitches include the *lockstitch*, *zigzag stitch* (for preventing fabric unraveling) and the *overlock* or *serger stitch* (for bindings at the edge of the

Figure 2-20: Different types of seams in the inside of a night robe (left), and border seams binding two flat pieces of fabric as a table mat (right).

fabric [81]). Figure 2-20 illustrates different types of seams.

Embroidery

Beside binding objects and fabric together, sewing can be used for *embroidery*, i.e. using the sewn thread as an embellishment, or means to change the fabric appearance and draw motifs as shown in Figure 2-21. Advanced modern sewing machines can be extended with a gantry for automatic embroidery given an input image to the machine [24] as illustrated in Figure 2-19. *Visible mending* makes use of embroidery to transform garment defects into decorative patterns.

Quilting

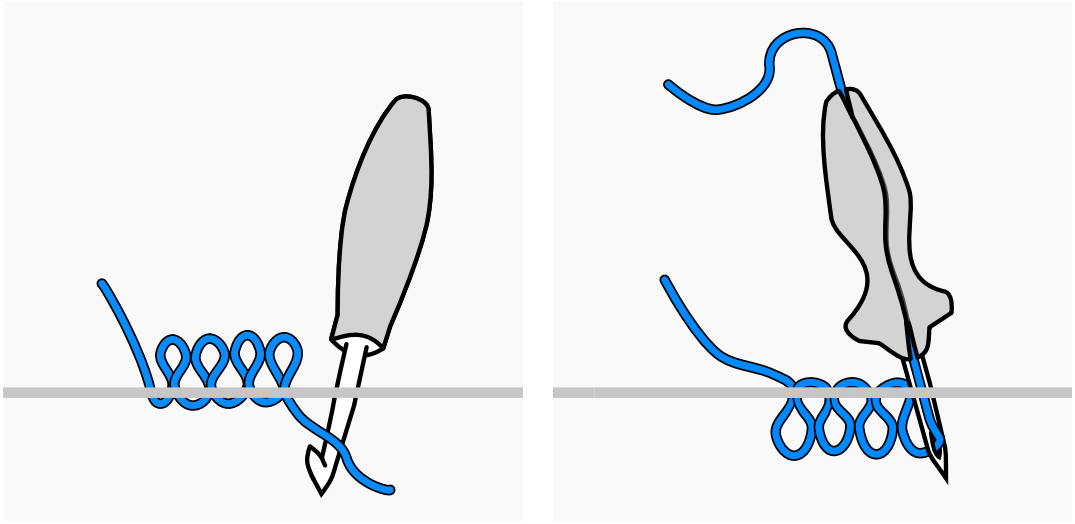
Quilting is a form of sewing that integrates several layers of fabrics. Similarly to embroidery, it is typically used as an embellishing of the fabric, and often mixes different types of fabrics or colors.

Figure 2-21: Fine embroidery on a shirt (left and top), and coarse yarn embroidery on a drawing (right and bottom) with its mirrored back (right inset).

2.3.7 Tufting

Tufting is primarily used to create rugs and carpets, and consists in inserting yarn loops through an existing structure (e.g., another textile, typically woven) with some form of needle. Compared to sewing, it only requires a single thread and does not need to completely cross the base material [106] – although the most common forms typically go through. It has notable uses in composite reinforcement [30, 68, 106].

Manual tufting can be done with a *hook needle* by pulling loops of yarn through the base material, or with a *punch needle* by simply going through the material and retracting to leave a loop on the other side. Both are illustrated in Figure 2-22. For larger-scale projects, *tufting guns* provide a semi-automated variant of the punch needle that typically includes an automatic loop cutting mechanism to allow for a felted finish. Industrial tufting machines for carpets and rugs basically proceed in the same way with many needles actuated in parallel.



(a) Hook needle (b) Punch needle

Figure 2-22: Two different tools for tufting: a hook needle pulls the yarn back through the material to form a loop (left), whereas a punch needle pushes the yarn through and retracts while letting the yarn slide and stay as a loop (right).

2.3.8 Non-Woven

Non-woven textiles bundle fiber together with a limited structure and typically shorter macroscopic continuity of the fibers. They notably do not require the fiber to be transformed into yarn for production. They span various application domains from garments to the health industry and other technical textiles [8].

Felting

Felt is made by explicitly tangling fibers together locally. It seems to have appeared in human history much before knitting and weaving [95]. The resulting fabric has interesting physical properties including water absorption, permeability, fire resistance and insulation capabilities [52].

Manual felt making is done either *wet*, by entangling fiber in hot water with friction, or *dry* by using barbed needles to poke the fiber and increase its internal entanglement. Hobbyist felting machines look similar to sewing machines, although they typically do not introduce any yarn. The fiber is locally added manually, and punched successively until it binds to the existing felt fabric. Figure 2-23 shows

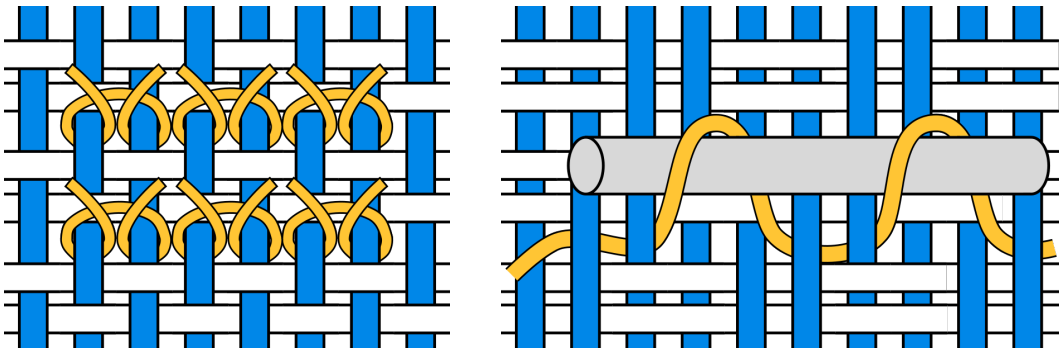
Figure 2-23: Felt examples: raw sheets of felt (top-left), a fox created by needle felting (bottom-left), and rugs (right). Credits to Sarah Stierch for the Kyrgyz felt rugs – CC-BY 4.0 – and to Amanda Adebisi for the fox – CC-BY-ND 2.0.

different examples. Recently, low-cost 3D printing systems were used to create 3D felted fabrics using yarn [74], or by binding layers of felted fabric [127, 128].

The main industrial production of felt is based on the dry mechanism: fiber is distributed and pressed between two panels, before being repeatedly poked through to entangle the fiber structure with many needles in parallel.

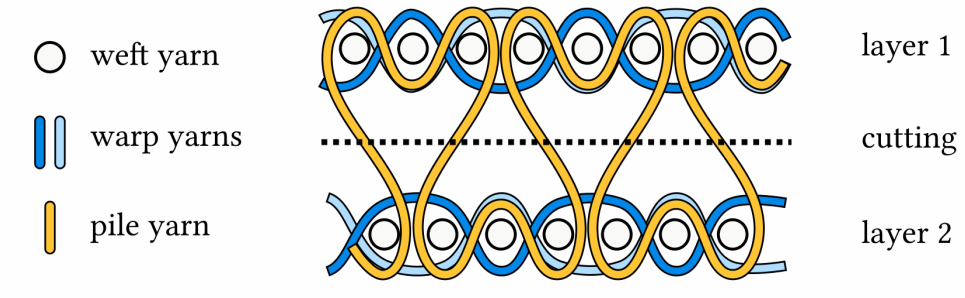
2.3.9 Napped and Pile Fabric

In general, the *nap* refers to the fuzzy surface of fabrics such as felt. The *nap* originally referred to the rough surface of woolen fabric before it was *sheared* to improve its smoothness – effectively removing the nap. It then later referred to raised fibers introduced explicitly as part of the fabric – also known as *pile*. In both cases, the fabric is said to be *napped* if it was processed to get a smooth finish – typically by *raising* the nap, and then *trimming* it.



(a) Knotted pile

(b) Rod on hand loom



(c) Face-to-face weaving

Figure 2-24: Different mechanisms to introduce pile in woven fabric.

Tufting directly creates pile by inserting loops through a primary material. Knitting can introduce pile either through dedicated mechanisms (e.g., in warp knitting), or by using specific knitting structures such as floating yarn or spacer fabric. Weaving relies on specialized machines or mechanisms.

Woven Pile Fabric

Some of the oldest woven pile formations are based on manually inserting knots in the woven structure – the *knotted pile*. Other traditional methods form either *weft pile* or *warp pile* by manipulating the corresponding yarns (weft or warps) so as to create local loops. Figure 2-24 illustrates some of them.

One manual method used on hand looms consists in inserting *rods* and twisting the weft (or a dedicated pile yarn) around these rods so that their later removal forms pile loops. Power looms use various specialized mechanisms or weave structures that make use of floats which are eventually raised, either during the weaving process,

Figure 2-25: Towels, blankets and rugs are common examples of pile fabric. The loops can be kept as-is (top-left) or cut and processed for a softer finish (right). Furniture also commonly uses napped fabric such as on this velvety box chest (bottom-left).

or a posteriori. One specialized mechanism is *face-to-face* weaving that creates two distinct woven fabrics which are bound together with the pile yarn – typically along the warp direction. The pile yarns are then eventually cut to separate the two fabrics, resulting in a cut pile. More complex mechanisms such as those in *Axminster looms* use dedicated pile warps that can be programmatically inserted in the main fabric.

Common Pile Fabrics

One of the most notable pile fabrics is *velvet* – a cut-pile fabric with even, short pile heights. It has a distinctive soft feeling and a strong sheen [10, 124]. While velvet is typically a *warp-pile* fabric, *velveteen* is a *weft-pile* fabric that looks very similar. Another common pile fabric is *plush* that differs from velvet by its longer cut pile and a typically lower density. One of its main uses is for the fabrication of stuffed toys such as teddy bears, typically called *plushies*. *Velour* is a type of plush fabric that is often used in clothing. *Terrycloth* is a loop-pile fabric – i.e., the loops are kept uncut – commonly used for towels and bath robes given its high absorption capabilities [130]. Some of those fabrics are illustrated in Figures 2-25 and 2-26.

Figure 2-26: Examples of napped fabric used in garments: as an inner layer of a knitted sweatshirt (top) and as an outer layer fleece (bottom).

Chapter 3

Computerized Machine Knitting

In this chapter, we first describe the machinery involved in flat knitting machines (Section 3.1). We then consider different low-level programming options for those machines (Section 3.2). Finally, we describe some of the common topology representations that are behind recent, higher-level design tools (Section 3.3).

3.1 Flat Knitting Machinery

The machine used in this thesis is a "whole-garment" knitting machine from Shima Seiki [150] (model SWG091N2, needle gauge 15), illustrated in Figures 3-1 and 3-2. While we try to mention other types of mechanisms and components to provide a more general overview, the focus is explicitly on the aforementioned machine model. See the brief review of Choi and Powell [35] for a description of other existing flat bed machines, or the book of Spencer [158] for a complete review of knitting machines.

Computerized flat knitting machines all typically possess a few fundamental mechanical components: one or more *needle beds* that hold the needles being actuated, one or more *carriages* containing electronically-programmable cam systems that interact with the needles of the machine, one or more *yarn carriers* that bring the yarn to a level where it can be caught properly by the needles, and multiple *rollers* that can catch the fabric as it is knitted to condition the yarn tension, direct the fabric out and prevent stitches from piling up.

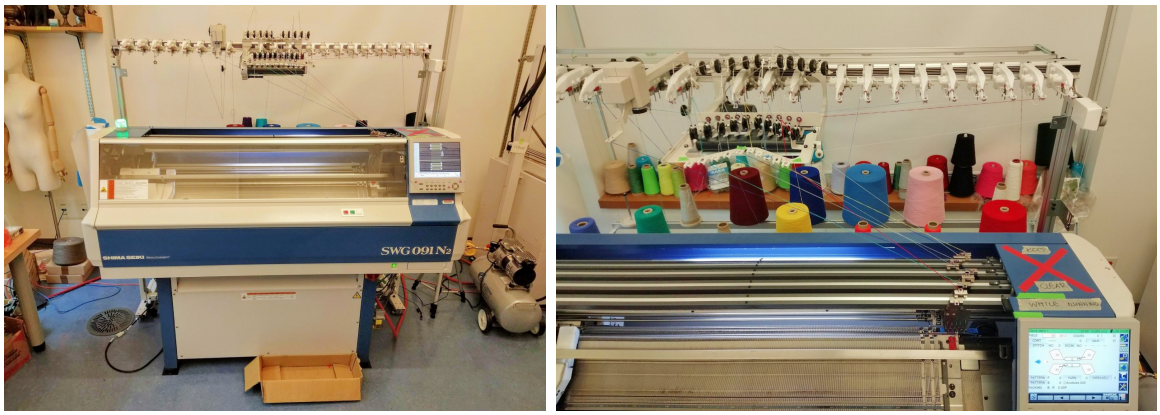


Figure 3-1: The machine used within this thesis (left) and a close-up from above, highlighting the yarn setup, with the protective cover opened.

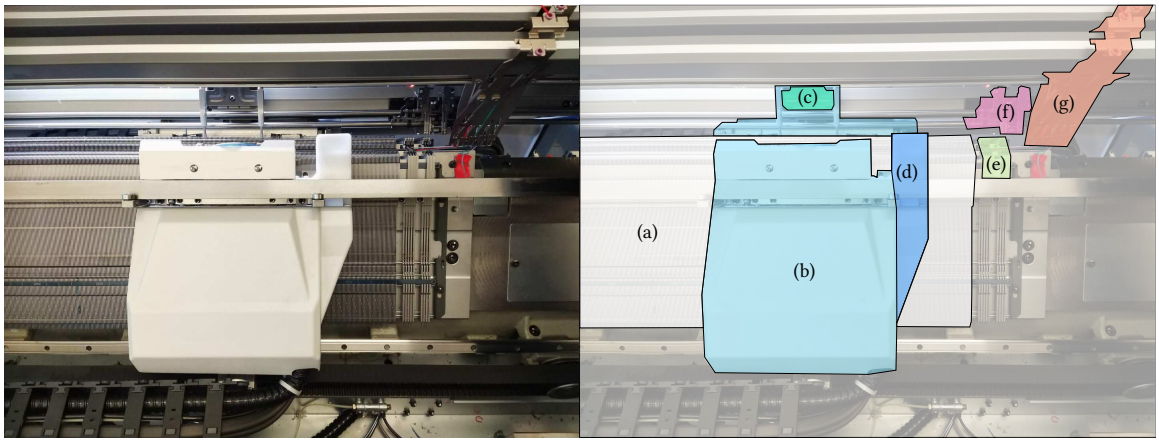
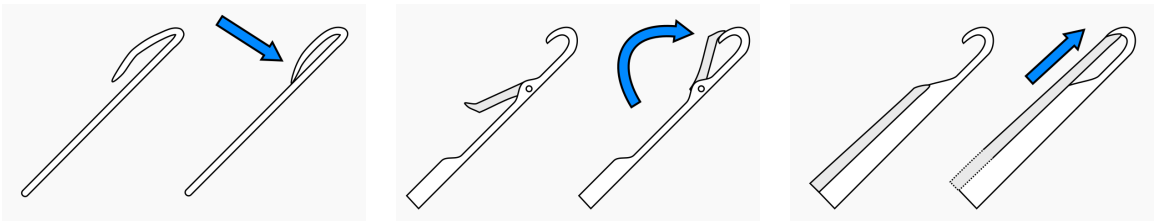


Figure 3-2: Inside view of the machine with component overlays: (a) needle bed, (b) carriage, (c) presser plate, (d) vacuum vent, (e) yarn holding hooks, (f) yarn insertion unit, (g) multiple yarn carriers.

3.1.1 Needles and Needle Beds

The needles hold the loops of yarn and are thus one of the key components of the knitting machine. The two most common, modern knitting needle types are the *latch* needle and the *compound* (or *slide*) needle, illustrated in Figure 3-3.

Latch needles rely on a latch at the base of the needle hook that can rotate to close/open it. The opening and closing of the latch happens naturally with the stitch loop motion as needles go through their up-and-down actuation, whereas brushes are typically used for opening it in other scenarios.



(a) Bearded needle (b) Latch needle (c) Slide needle

Figure 3-3: The three most common types of machine knitting needles.

Compound / slide needles combine an open hook needle together with a *slider* that "slides" on top of the needle component, effectively allowing the closing/opening of the hook section via a simple linear action (i.e., translating the slider component). This thesis uses a machine with slide needles.

Needle Beds

A needle bed is a panel that contains needles laid out parallel at a regular interval. The linear density of needles is called the *gauge* and is measured in *needles per inch*. The two most common types of beds are called *V-bed* and *X-bed*. The former consists of two beds positioned in an inverted V form, whereas the latter uses four beds in an X configuration. This work uses a 15-gauge V-bed machine shown in Figure 3-4.

A single bed can produce flat sheets of fabric. Two beds allow for tubular structures and combinations of those with flat sheets of fabric. The purpose of having four

Figure 3-4: Views of the needle beds: from the left side (left), from above (center) and from the right side behind the yarn carriers (right).

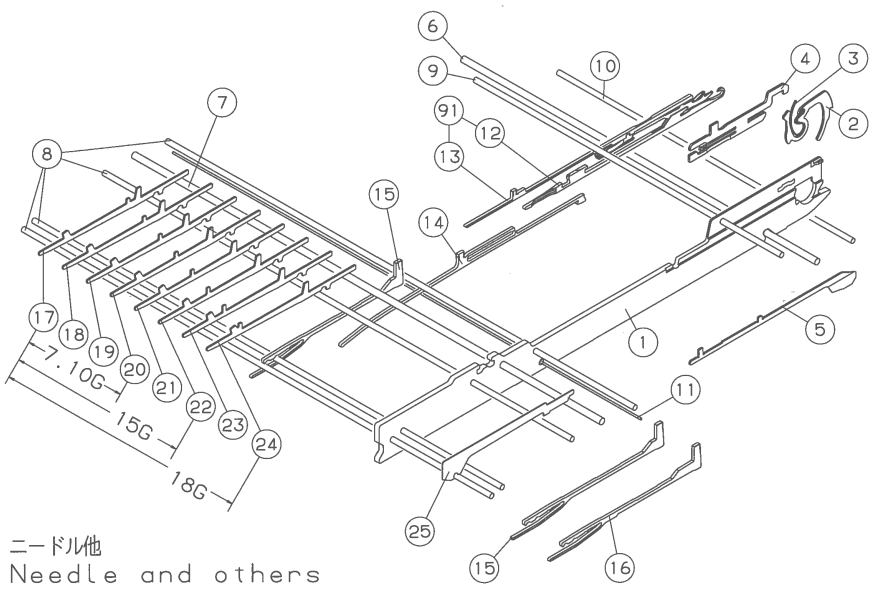


Figure 3-5: Schematics from Shima's manual [150] illustrating the replaceable components on the needle bed: the needle plate for containment (1), the sinker (2) and its spring (3), the sinker jack that engages the sinker through the cam (4), the yarn guide (5), different wires for fixing elements in place (6-11), the slide needle (12), the slider (13), the jack that engages the needle through the cam (14), the select jack that selects which mode of the cam to go through (15-16), different versions of selectors that get triggered by solenoids (17-24), the select spacer (25).

beds – in the *X-bed* setting – is not to increase the span of complex surfaces that can be created. Instead, it is so that stitches on a *lower* bed can be hold temporarily on the corresponding *upper* bed. This temporary displacement frees the lower needle and allows stitches from the other lower bed side to knit on their opposite lower bed – the one we moved our stitches to the upper bed from –, before moving stitches back to their corresponding original beds. This is notably useful for creating complex stitch patterns when knitting tubular structures.

Needle, Jacks, Selectors and Sinkers

Figure 3-5 illustrates the different replaceable components that are involved with each of the individual compound needles, whereas Figure 3-6 shows the combination of the slide needle, slider and needle jack.

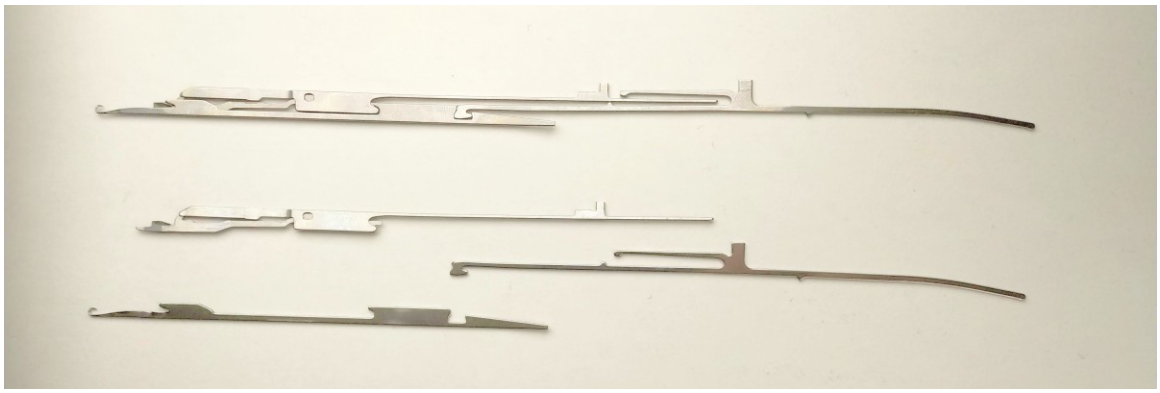


Figure 3-6: The typical group (top) of components that get replaced, separated from bottom to top: the slide needle, the needle jack and the slider.

Jack components have a pin sticking out that gets engaged through the cam depending on the pin location. Simpler needle designs have a pin integrated as part of the needle itself. The reason for wanting a separation is due to both the needle hook and the pins being typical sources of breakage: replacing only one allows for some cost reduction, to the detriment of a more complex design.

Select components are related to the different ways that jacks can go trough the cam. By default they do not engage. Their designs are typically differing slightly across neighboring needle lanes to allow for high-speed solenoid selection without conflicts. Figure 3-5 shows many variants (pieces 17 to 24).

Sinkers primarily serve to hold down old stitch loops as the needles are actuated to form newer stitch loops. In some knitting machines, they also help with the loop formation process.

3.1.2 Carriage

The carriage is the component that moves back and forth on the needle bed to actuate the needles locally. Figure 3-7 illustrates the internal of one side of such carriage – the other bed side having a similar system. Some of the important components include:

- *cam systems* [56, 139] that engage the needle components through their associated jack pins – typically, one for the needle and one for the sinker,

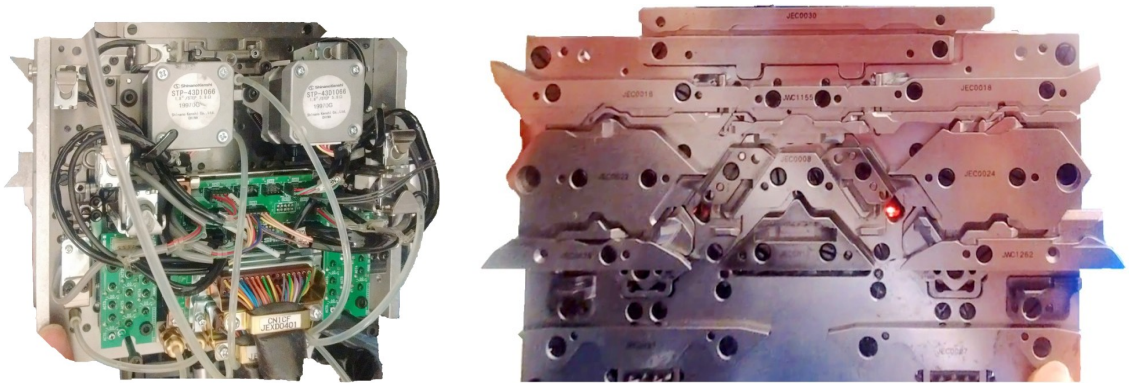


Figure 3-7: Internal views of the carriage: its front (left) and back (right).

- different sets of *solenoids* (e.g., for selecting the needles and the presser plates),
- a *presser plate* that effectively presses on the yarn to prevent its rebound when knitting complex short-rows locally,
- a *vent* connected to a vacuum for clearing fibers and cut yarn pieces.

By selecting needles with solenoids and electronically controlling the cam path, the machine can specify the actions of each individual needle in the path of the carriage.

3.1.3 Yarn Carriers

Yarn carriers are responsible for bringing the yarn locally near the location where needles are being actuated. Since the needles are actuated by the carriage's cam, the yarn carriers tend to follow it. Our machine allows for two types of yarn carrier movements: *synchronous* in which case the carrier is moved together with the carriage (this is typically done on other machines by selecting the carrier with a solenoid), or *asynchronous* in which case the yarn carrier moves separately from the carriage. The latter is possible thanks to the yarn carriers being driven independently by stepper motors (one for each carrier).

Manual Tensioning Mechanisms

On our machine, the yarn typically (1) starts at a cone, (2) gets caught by a magnetic bottleneck with tunable friction, (3) goes through small eyelets to detect accidental

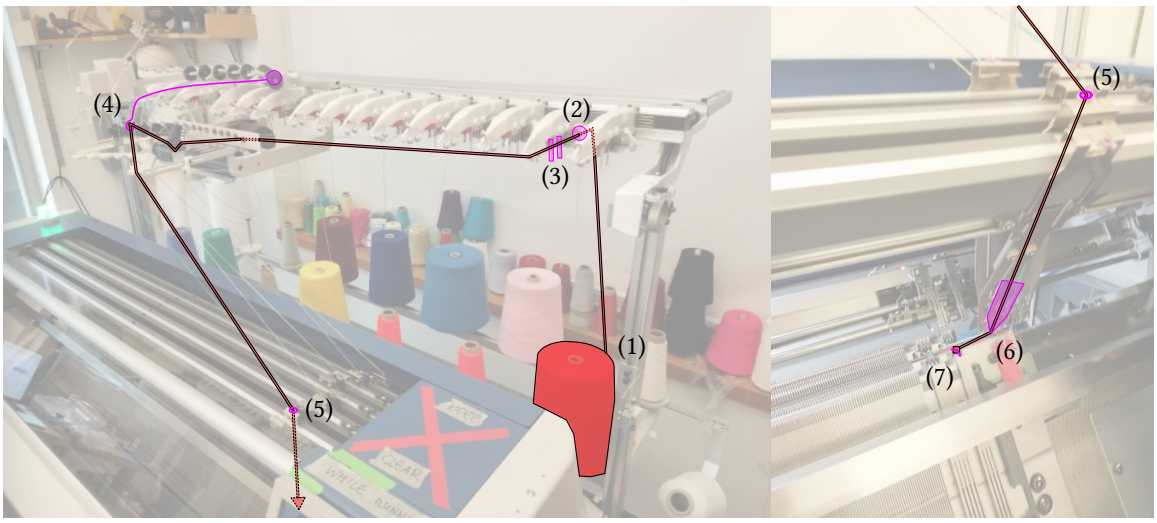


Figure 3-8: Path of the red yarn from its cone to the yarn carrier and kept locked in the yarn holding hook.

knots of tunable width, (4) is put under tension by a tunable springy arm, (5) is routed to the top of a specific yarn carrier, (6) exits the base of the carrier to be caught somewhere, (7) specifically in the yarn holding hook when not used. These steps, illustrated in Figure 3-8, serve as a way to condition the tension of the yarn so that sensors can detect when things go wrong, and the yarn ends up getting pulled automatically when the yarn carriers move (so that it is always under proper tension). Note that the tunable controls are all mechanic and thus their tuning is done by manually adjusting knobs.

Typically the yarn can be caught (i.e., fixed in a stable manner) at a few locations: (i) by a *needle hook*, (ii) by the *yarn holding hook* that holds the yarn when not used, (iii) by the *yarn insertion unit* that grabs the yarn and inserts it to start knitting.

Finally, a small blade / scissor complements the insertion unit to ensure that the automatic insertion has calibrated length of dangling yarn (at start). The yarn that is cut gets vacuumed through a vent on the carriage.

Digital Stitch Control System

A different way to condition the yarn is through a system that controls the yarn tension with an electronic feedback loop, shown on the left of Figure 3-9. The base

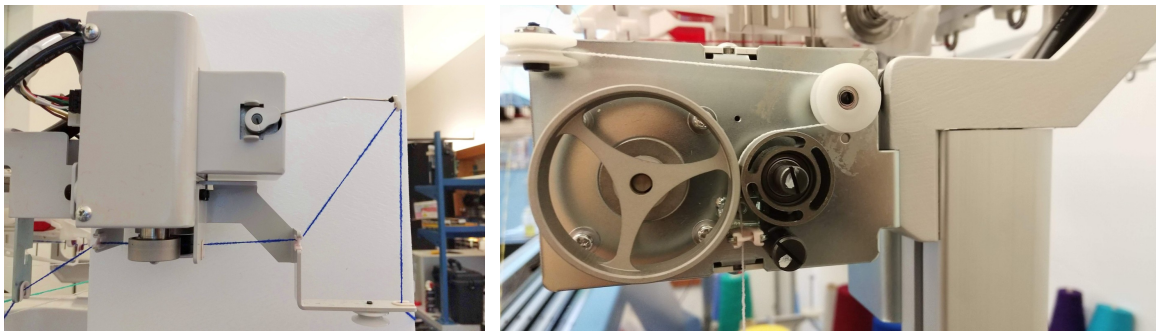


Figure 3-9: Common addon devices: digital stitch control system (left) and elastic system (right).

has a motor that catches the yarn against a rotating cylinder to control the length of yarn already engaged. The arm is connected to a servo that assumes a specific force feedback to be calibrated. Its purpose is to get information about the tension of the yarn and allow the system to react to any changes. In practice, it tries to stay at a horizontal angle.

The calibration is done in two steps: (1) the length of the yarn from the catching base to a specific needle on the needle bed is measured, (2) specific weights are hanged onto the servo arm instead of the yarn to get force measurements.

3.1.4 Rollers

Rollers – shown in Figure 3-10 – are used to catch the yarn and put it under tension (due to pulling) as it exits the machine. Having the yarn under tension can become important when knitting complex 3D shapes with large local curvature (e.g., short-

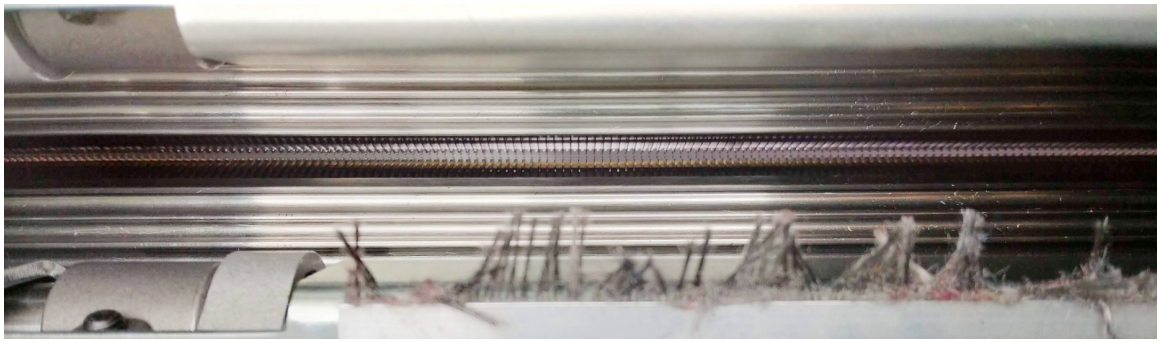


Figure 3-10: View of the rollers from below the needle beds. The horizontal slit at the center is the space between the needle beds. The rollers are currently *open*.

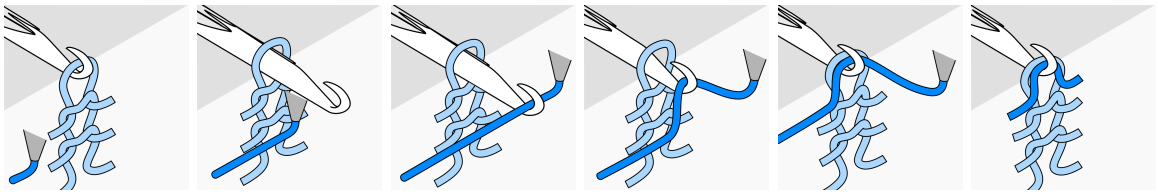


Figure 3-11: The tuck operation that adds a loop to the needle hook.

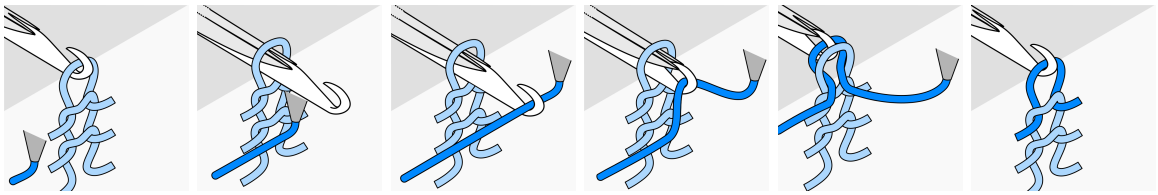


Figure 3-12: The knit operation as generated with a slide needle.

rows) as the yarn tends to bounce back up otherwise. In such scenarios, *waste yarn* is first knitted so that the initial knit structure gets caught by the roller before proceeding with the real knit portion. For all the designs in this thesis, the action of the sinkers and presser plates were sufficient to prevent stitch piling – i.e., we kept the rollers open.

3.1.5 Basic Operations

While the presented machinery can appear very complex, the number of distinct base operations the machine can produce is quite limited. We describe each of those here and refer the reader to the mathematical definitions available in the work of McCann et al. [116] and the original visualizations from Narayanan et al. [122].

Tuck

The *tuck* operation consists in having the needle hook catch a loop of yarn, while keeping the preexisting loops as-is. See Figure 3-11.

Knit

The *knit* operation consists in catching a loop of yarn while the previous loops are kept outside of the hook. The hook is then closed and the needle is retracted. This

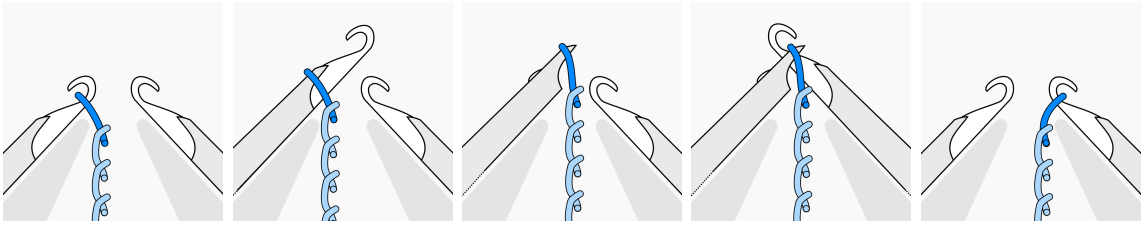


Figure 3-13: The transfer operation from one bed to the other.

knocks the old loops over and lets the new loop in the hook pull through them – effectively forming a knit stitch. See Figure 3-12.

Note that a knit operation without previous loops has topologically the same result as a tuck, although the motions of the needle components are different.

Miss

The *miss* operation consists of a no-operation in terms of needle operation. However, semantically it means that the carriage and yarn carriers went to the corresponding needle, and can thus be different from no operation from a program perspective.

Transfer

The *transfer* operation effectively transfers any loops from a needle to the needle facing it on the opposite bed. See Figure 3-13.

Sliders can be used as location for storing yarn loops temporarily during transfers. They cannot be used for any other needle operation than transfers, and the machine expects that the sliders are empty when starting any other operation.

Split

The *split* operation is a compound form of *knit* and *transfer* in which the previous loops are transferred to the other bed side instead of being dropped after pulling through the new stitch loop.

The *transfer* operation can be viewed as a split with no yarn as this does not create a new stitch but still transfers the previous loops to the other bed side.

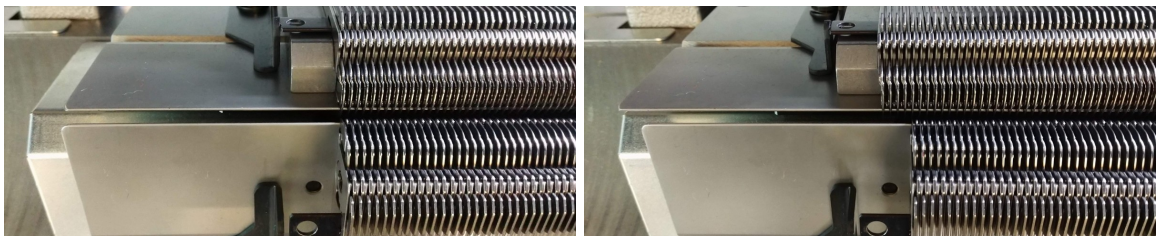


Figure 3-14: Racking example: zero offset and -4 offset of the back bed (in needles).

Racking

Racking is a bed-wise operation that offsets (by some limited number of needles) the two bed sides, as illustrated in Figure 3-14. In conjunction with transfers, this allows the displacement of stitches along each needle bed by (1) transferring to the other side, (2) racking, and (3) transferring back.

Through sequences of transfer and racking operations, a stitch can be moved mostly anywhere in the needle and bed space, notwithstanding constraints due to the existing other stitches and the yarn tension.

Amiss

The *amiss* operation is a special form of carriage movement that pulls down the yarn brought by the yarn carrier(s). This matters in some scenarios when we want to float one yarn and have it caught by stitches that transfer across beds, mainly because the yarn carrier exits are typically some distance above the location at which the stitch do their transfers. By using an *amiss*, we can take the yarn lower without having to catch it in a needle (and then drop it).

3.1.6 Special Carrier Modes

Flat bed machines can have various types of yarn carriers and modes for their actuation on the bed. For our machine, the yarn carriers stay the same, but their programming modes include three important variations: *inlay carrier*, *plating carrier* and *elastic yarn carrier*. Samples using the first two are shown in Figure 3-15.

Figure 3-15: Examples using special carrier modes: white inlay yarn (left), gray-blue plated yarn that controls the color side using purls and ribs (right).

Inlay

Inlay is a yarn that is typically *not* knitted, but is hold in the knitted structure via different means. There are two main forms of programmatic inlay: *automatic* inlay that relies on an dedicated carrier synchronously moving slightly ahead of the base yarn carrier, and *manual* inlay that explicitly draws the inlay yarn separately from the base yarn and relies on transfers of the base yarn to "catch" the inlay yarn in the base structure. This second form of inlay is a case where the *amiss* operation becomes important. The first form typically has no issue with catching the yarn because the offset of the inlay carrier is tuned so that the inlay ends up at a low enough height to be caught properly by the main yarn as it alternates between bed sides.

An important mechanical difference of inlay compared to knit fabric is that it does not stretch in its drawn direction – unless elastic. This produces a mechanical blocking mechanism over the base structure that typically stretches in both directions. Inlay can also be used for cable actuation and other functional purposes [7, 109, 110].

Plating

Plating consists in using multiple yarn carriers that knit together, but with a slight offset of the yarn carriers that drives the ordering of the loop stack. It is typically used for colorwork with two different yarn colors.

Elastic

Elastic yarn (e.g., elastane) can be drawn specifically as inlay that is pre-stretched. As a consequence, the knitted artifact ends up shrinking due to the elastic inlay, but can typically stretch back to the original structure due to the elasticity of the yarn. This is often used for socks or as part of the cuff of gloves. From a programming perspective, elastic yarn provides an additional control which is the frequency at which to release the yarn. By default, the system keeps the elastic yarn from drawing so that it gets pre-stretched. By choosing how often it gets released, we can control the amount of pre-stretching that gets introduced.

3.2 Low-Level Machine Knitting Programming

At the lowest level, programming a knitting machine consists in scheduling the motion of its carriage, yarn carriers and the actuations within the carriage to select needle operations during its motion. This complex scheduling is typically not done by the user. Instead, the user controls are reduced to mainly the sequence of *needle operations*, together with machine and yarn carrier *states*. This effectively focuses the attention on the topology of the knit construction, while providing control over important states that can ease or simplify the manufacturing. Those *states* may typically include: which yarn carriers are active and their locations, in which direction the carriage is moving and at what speed, the speed and functions of other knitting devices (e.g., rollers, presser plate, yarn insertion unit, yarn holding hooks, etc.). The resulting description is then translated into carriage and yarn carrier motions, as well as low-level device actuations that can be dependent on those motions.

We first describe the common representation for most proprietary software based on time-needle images, and then mention the other sequential forms that have been developed in hand knitting and more recently in computational knitting research.

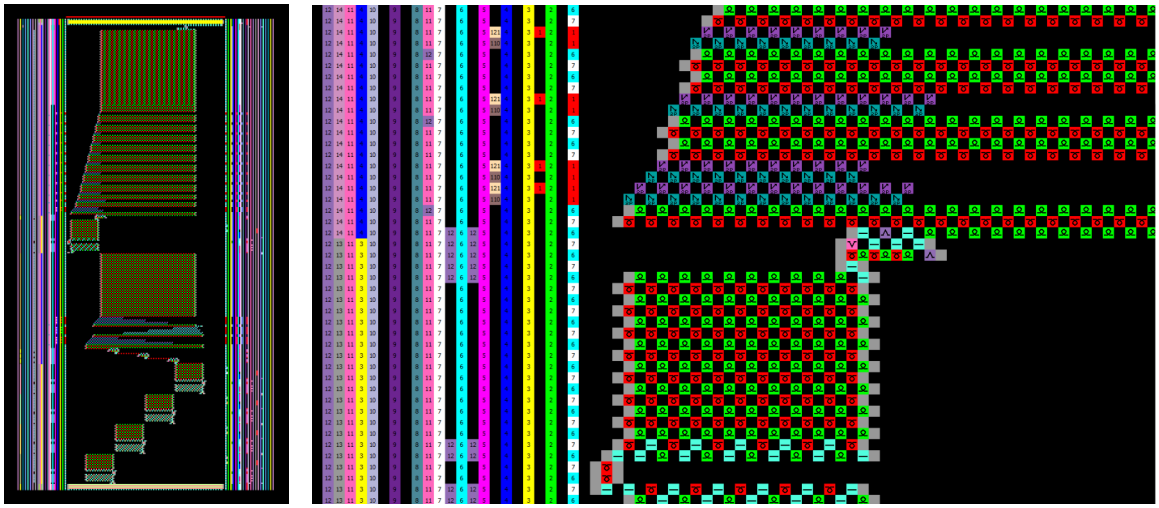


Figure 3-16: Examples of time–needle programs in KnitPaint [150]: the overview of a glove (left) and a close-up near the merging of the thumb with the palm (right). Vertical bars on the sides specify machine states for the corresponding lines.

3.2.1 Time–Needle Images

The most common low-level programming representation consists of *time–needle* images such as the ones shown in Figure 3-16. Typically, the x-axis represents the needle space and the y-axis represents time. Each pixel – which we'll refer to as *stitch code* – encodes a needle operation, a machine state, or a combination thereof. The different machine states are encoded either *explicitly* in sections of the time–needle program dedicated to machine states (e.g., on the left and right of the needle program), or *implicitly* with the needle operations in the stitch codes.

In the case of Shima [150], their stitch code library does a mix of both: some states are explicitly controlled in locations on the left and right of the program, and some are part of the needle operations. Furthermore, they also introduce context sensitive states that depend on the local needle operations. As such, a given stitch code may carry information that is dependent on the stitch codes below it in the time–needle program. The work of Underwood [170] provides a good overview of this design space, and it notably introduces higher-level constructions encoded in user-defined stitch codes.

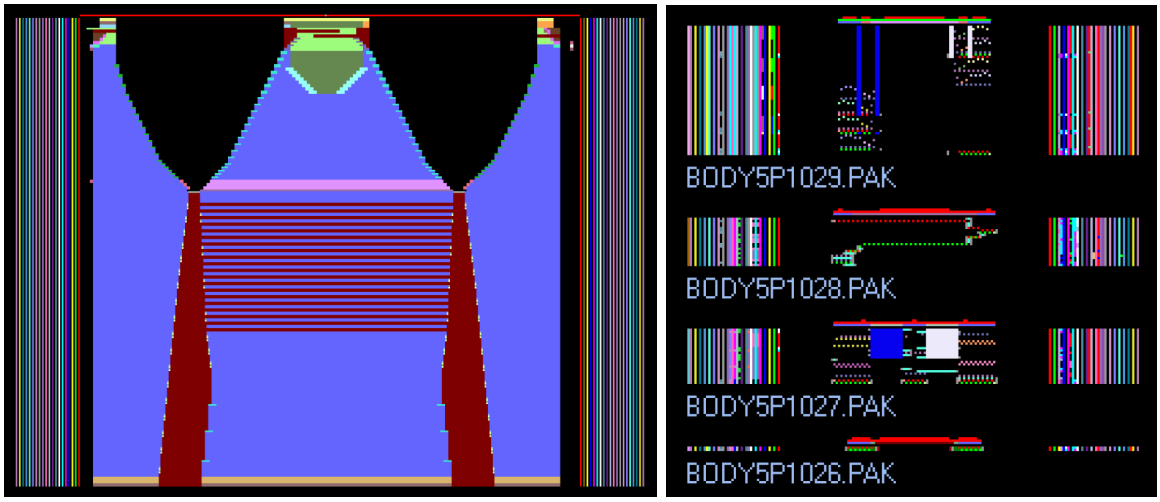


Figure 3-17: Example of high-level stitch code program for a raglan shirt from Shima Seiki (left), and 4 of its 57 accompanying packages (right).

Package Development

One way to modularize time–needle programs consists in developing libraries of higher-level stitch codes. Shima [150] implements this with *packages* that rely on pattern matching: the user associates a high-level stitch code configuration (e.g., which stitch codes are next to which other stitch codes) and the corresponding stitch codes that effectively implement the corresponding lower-level program (using low-level stitch codes, or potentially other high-level stitch codes). The user can then write a knitting program using the corresponding higher-level stitch codes and use a specific set of packages to transform the program into a lower-level one that can be compiled for the machine. This is essentially a form of string rewriting system [23, 135, 141]. An example of complex pattern development is illustrated in Figure 3-17.

One specificity of package development is that high-level stitch codes are not considered valid forms of low-level programs to be compiled for the machine. In practice, this requires the user to describe any meaningful context that can be matched. If an unmatched context is present, then the package development fails as some high-level stitch codes cannot be developed into some low-level versions. From the package development perspective, this is *desirable* as no rule was provided to deal with the unexpected solution. From a user perspective, this can be a bottleneck. Further-

more, the limitations on *how* the higher-level stitch code context is described directly impact the expressiveness of the higher-level stitch programs.

Free Packages form a type of packages that uses simpler rules for pattern matching. It is targeted at local forms of patterns such as with *jacquard knitting* that emulates jacquard patterns from weaving with yarn loops. Figures 3-19 and 3-20 respectively illustrate different packages for jacquard knitting, and corresponding examples of pattern developments.

3.2.2 Instruction Sequences

Another strategy to writing knitting programs consists in explicitly writing the instruction as a list of instructions similar to a traditional computer program written with lines of code in C/C++ [161] or Javascript [43].

An example is KnitSpeak [133] that was developed by hand-knitters to describe their patterns. For machine knitting, McCann et al. [116] developed the knitting assembly language known as Knitout [115]. Its base instructions extract a minimal set of operations that allows the specification of knitting programs for Shima, yet with a much more concise set of operations. Those operations are tightly bound to the needle operations in Section 3.1.5. Examples of both are illustrated in Figure 3-18.

```
Row 1 (RS): Sl1 wyif, k1, yo, k2tog, yo, k1.
Row 2: K1, 1-to-5 inc, k1, yo, k2tog, k1.
Rows 3, 5, and 7: Sl1 wyif, k1, yo, k2tog, k6.
Rows 4 and 6: K7, yo, k2tog, k1.
Row 8: BO 5 sts, k1, yo, k2tog, k1.
```

```
;!knitout-2
;;Carriers: 1 2 3 4 5 6 7 8 9 10
;Node 0
x-stitch-number 33 ;yarnstart
inhook 1
tuck + f6 1
tuck - f2 1
tuck + f4 1
miss + f6 1
releasehook 1
knit + f0 1 ;caston
knit + f4 1
knit + f8 1
knit + f12 1
knit + f16 1
knit + f20 1
knit + f24 1
knit + f28 1
knit + f32 1
```

Figure 3-18: Examples of knitting programs in Knitspeak (top) and Knitout (bottom).

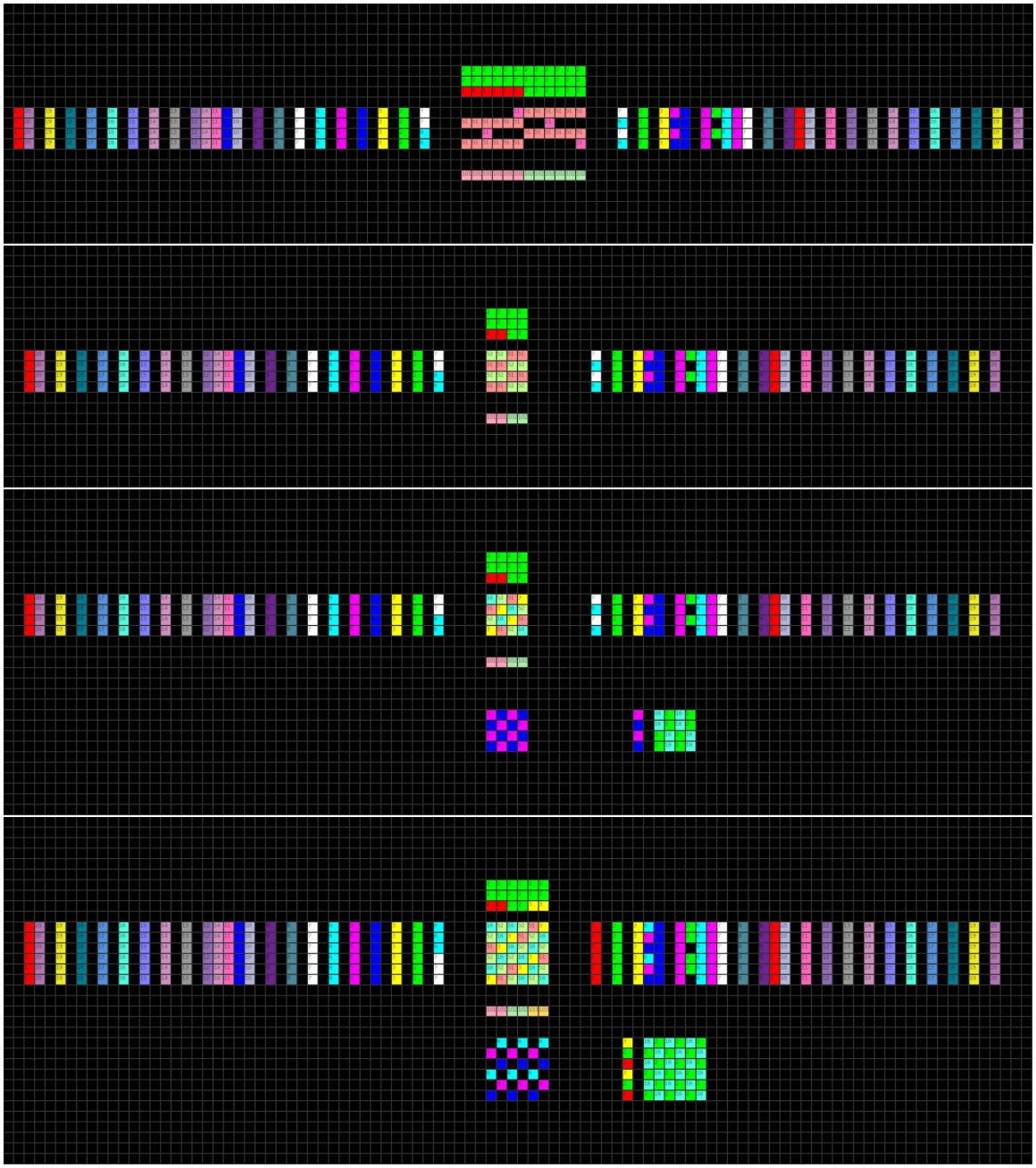


Figure 3-19: Examples of free packages used for jacquard knitting, with N colors and a specific backing strategy, from top to bottom: $N = 2$ floating, $N = 2$ tubular, $N = 2$ "pique" and $N = 3$ alternating.

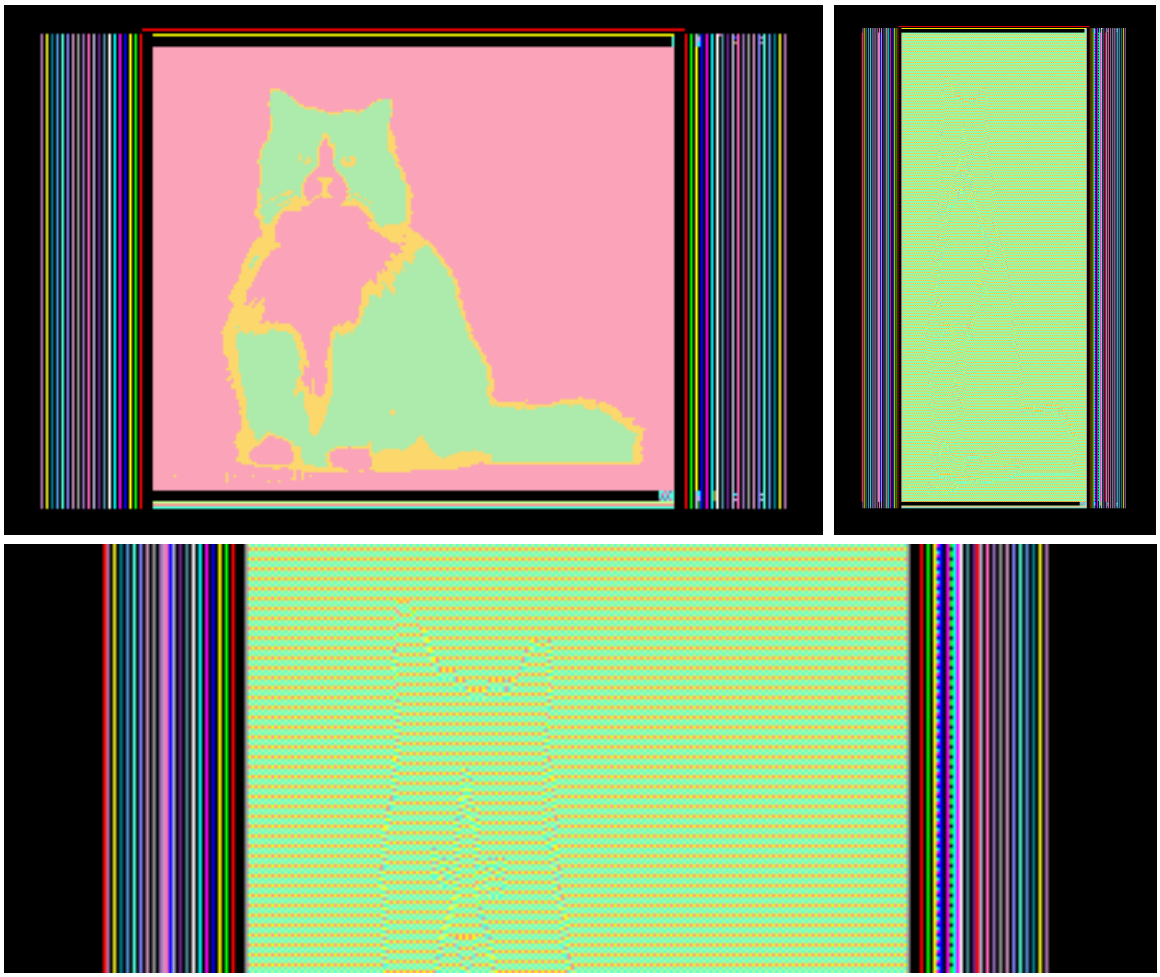


Figure 3-20: Example of jacquard-knit program using free packages: the user input (top-left), its developed result (top-right) and a close-up highlighting the local patterns (bottom).

Figure 3-21: Knitted 3-colors jacquard of cat pattern: its front (top) and its back (bottom). The smaller slices are close-ups around the eyes of the cat for both sides.

3.3 Stitch Representations

Beyond working directly with the action and state spaces of the knitting machine, higher-level programming requires some form of representation of the garment to be knit. The choice of representation is critical as it constrains the design space. We present here some of the most common representations.

3.3.1 Mesh-based Representations

One of the initial representations for high-level knit programming comes from the simulation work of Yuksel et al. [190] who associated different types of stitches with faces of a 3D mesh – the *stitch mesh*. They further label the edges of each face as either *course* or *wale* edges, providing information about the underlying knit topology. The initial work [190] considered invalid knitted tubular topologies (loops instead of helixes) as it was focused on fast, plausible rendering. It was extended with an automated algorithm to convert a 3D mesh into a stitch mesh [186], and then made *hand-knittable* by introducing shift paths and enforcing direction constraints [187]. Narayanan et al. [123] built a design tool to create machine-knittable stitch meshes while storing stitch instructions in the code associated with each face type. Guo et al. [63] proposed to use stitch meshes for representing *crochet* topologies. Finally, Wu et al. [188] looked at the decomposition of 3D structures into flat panels, together with algorithms to schedule those on the knitting bed.

While stitch meshes work especially well for simulating complex 3D geometry, this thesis explicitly avoids using such representation.

Creating a 3D Mesh is a complex and tedious process. Although there are many dedicated tools, they mainly focus on virtual purposes such as character animation, photorealistic renderings and simulation. The perspective of this thesis is geared toward accessibility, and thus we decided to avoid relying on a representation that requires complex engineering before being able to start with the design process itself.

Modeling Tools for Engineering and Fabrication are all essentially parametric, which 3D meshes are not. Instead, meshes in engineering applications are ephemeral outputs that are generated when the user requires integration with tools designed for simulation or photorealistic rendering. Most modern Computer-Aided Design (CAD) tools rely on boundary representations [11, 108] that combine parametric 2D sketches with geometric operations such as boolean operations and face extrusions. A key advantage is that the user description of the shape is simpler (much fewer degrees of freedoms). Furthermore, by using a parametric representation, one can reason not only about the desired physical artifact, but also the space of its parametric variations, thus enabling design exploration which is critical in early iterative design cycles for digital fabrication.

3.3.2 Graph-based Representations

Narayanan et al. [122] introduce the *Knit Graph* as a high-level representation of the stitch topology targeted at machine knitting. It corresponds to the dual of the original stitch mesh representation [190] with *tubular* courses being represented as cycles, and as such, requires an additional *tracing* step to transform the graph into a structure that can be knitted with real yarn. Their strategy relies on *double tracing* – i.e., each node of the Knit Graph is to be knitted twice –, which affords a simple rule-based algorithm, and requires a minimal amount of yarn cuts, to the detriment of a loss in spatial resolution over the wale direction.

This thesis uses similar graph-based representations for the underlying stitch structure. Chapter 5 relies on a graph topology that does not require tracing, while storing the same circular course information for regular pattern application. The graph is referred to as *stitch graph* and illustrated in Figure 3-22. Chapter 6 relies on a graph similar to that of Narayanan et al. [122] – although we follow our initial naming of *stitch graph* –, and we propose modifications of the tracing algorithm to support intarsia.

More recently, Nader et al. [121] introduced a higher-level graph that uses a partition of course-connected stitches as nodes and their wale-connectivity as edges –

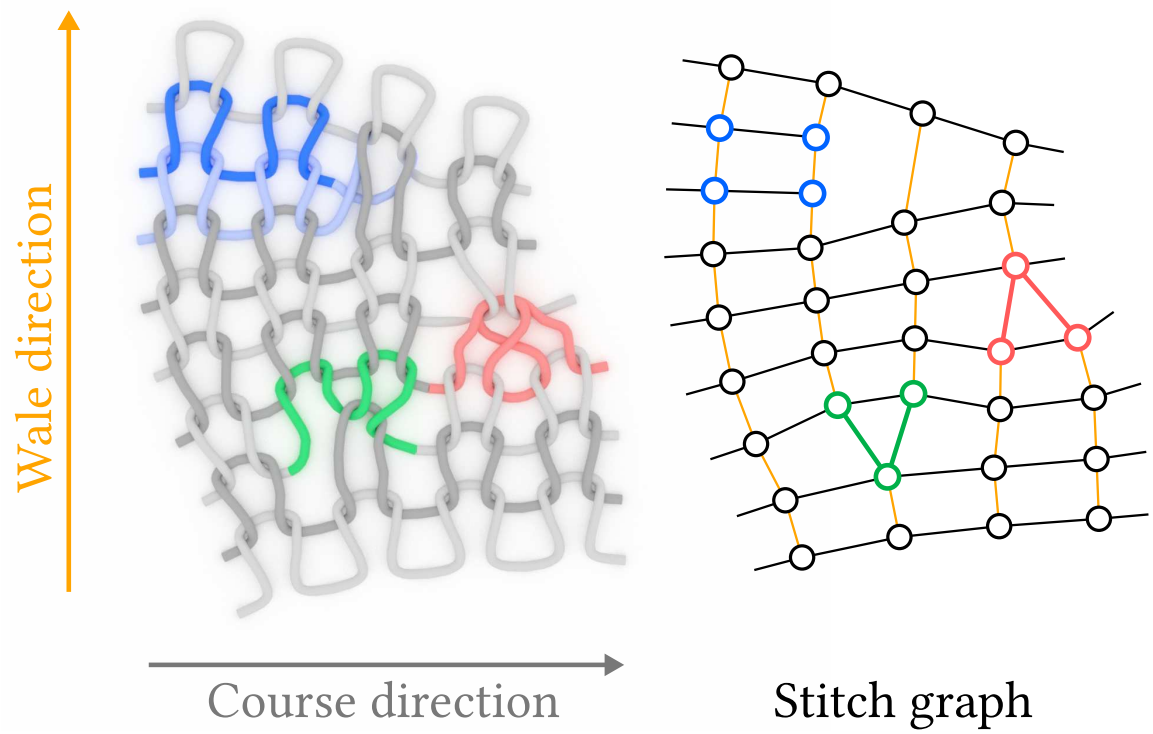


Figure 3-22: Illustration of important stitch topologies for machine knitting (left) and their corresponding stitch graph (right). Important concepts include: *courses* and *wales* as rows and columns of the knit structure; irregular shaping structures with stitch *increase* and *decrease*, and *short-rows*.

the *KnitNet graph*. They form an *Action graph* representing the action scheduling context given the previous graph and recursively rewrite it into a canonical form that can then be translated into machine instruction using a larger-scale form of context. Both Chapters 5 and 6 also rely on some form of higher-level representation based on courses and wale connectivity, but our scheduling is simpler as the focus of this thesis was on the user representation.

Terminology

Figure 3-22 visually highlights some of the important machine-knitting constructions and terms that this thesis extensively makes use of. The nodes of the stitch graph represent stitches, and the edges correspond to the stitch connectivity. *Courses* and *wales* respectively form rows and columns of the knit structure. Irregular structures for shaping include the stitch *increase* with two next wales, the stitch *decrease* with

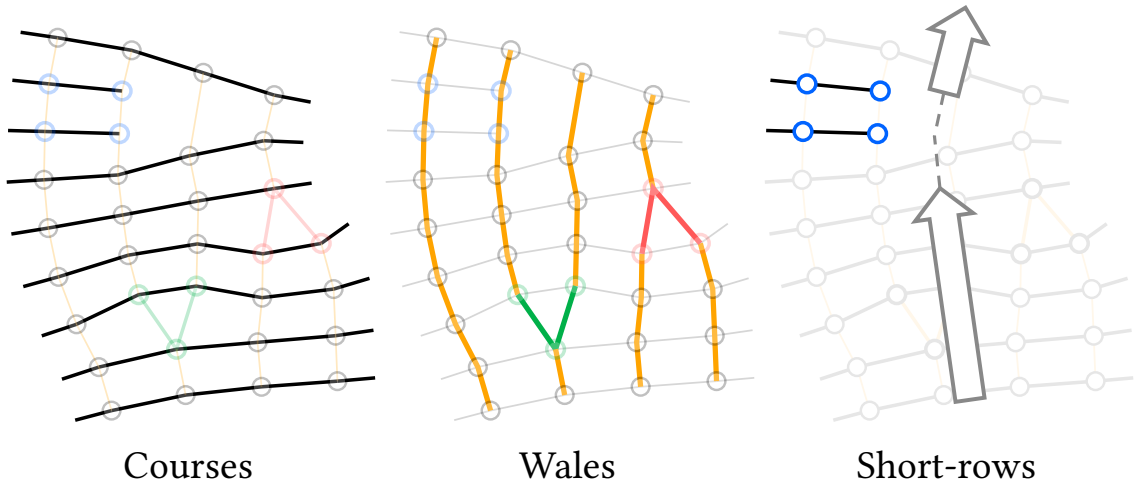


Figure 3-23: Highlight of different parts of the stitch graph: the *courses* that form rows of the knit structure, the *wales* as columns that split with stitch *increase* and merge with stitch *decrease*, and the bending effect due to *short-rows*.

two previous wales, and *short-rows* that only cover a portion of the base course. Each stitch node has one or two adjacent *course* neighbors on the same row. The *wale* neighbors can be separated into two groups – *next wale* neighbors and *previous wale* neighbors – and each stitch can have zero to two neighbors for each group. Stitches are considered as *regular*, if they have exactly one *next* and one *previous* wale neighbor, each corresponding to a (1-1) wale linking, while *increase* and *decrease* are *(1-2)* and *(2-1)* linkings, respectively. Figure 3-23 highlights the structure of courses and wales, as well as the bending impact of short-rows.

Short-rows can bend the course orientation to change the angle of knitting either in the same plane – when appearing at the side of a flat structure –, or out of the plane such as for tubular short-rows that are notably used in the heel of a sock.

Part II

Knitted Garment Design and Programming

Chapter 4

Learning-Based Garment Programming

While programming weft knitting machines requires expertise, knitted garments are ubiquitous around us. As a result, one may wonder whether it is possible to directly reuse those readily accessible designs – e.g., by scanning them, similarly to how one can scan text documents. More specifically, we ask the following question:

> “Can we automatically infer the knitting program corresponding to a piece of knitted fabric given a single image of it?”

The initial motivation that hints to a positive answer comes from low-level proprietary programming environments for weft knitting machines. The time–needle representation they use is very similar to pixel grids in image editing software [4, 98]. In fact, even the name of their software (KnitPaint for Shima [150]) is directly reminiscent of image editing software [44]. Meanwhile, the advances of machine learning have made possible many complex transformation between very different image spaces, notably with works on Conditional Generative Adversarial Networks [78].

Contents of this chapter are adapted with permission from - A. Kaspar, T. Oh, L. Makatura, P. Kellnhofer, W. Matusik, “Neural Inverse Knitting: From Images to Manufacturing Instructions”, ICML 2019. http://proceedings.mlr.press/v97/kaspar19a.html. [Copyright by the authors].

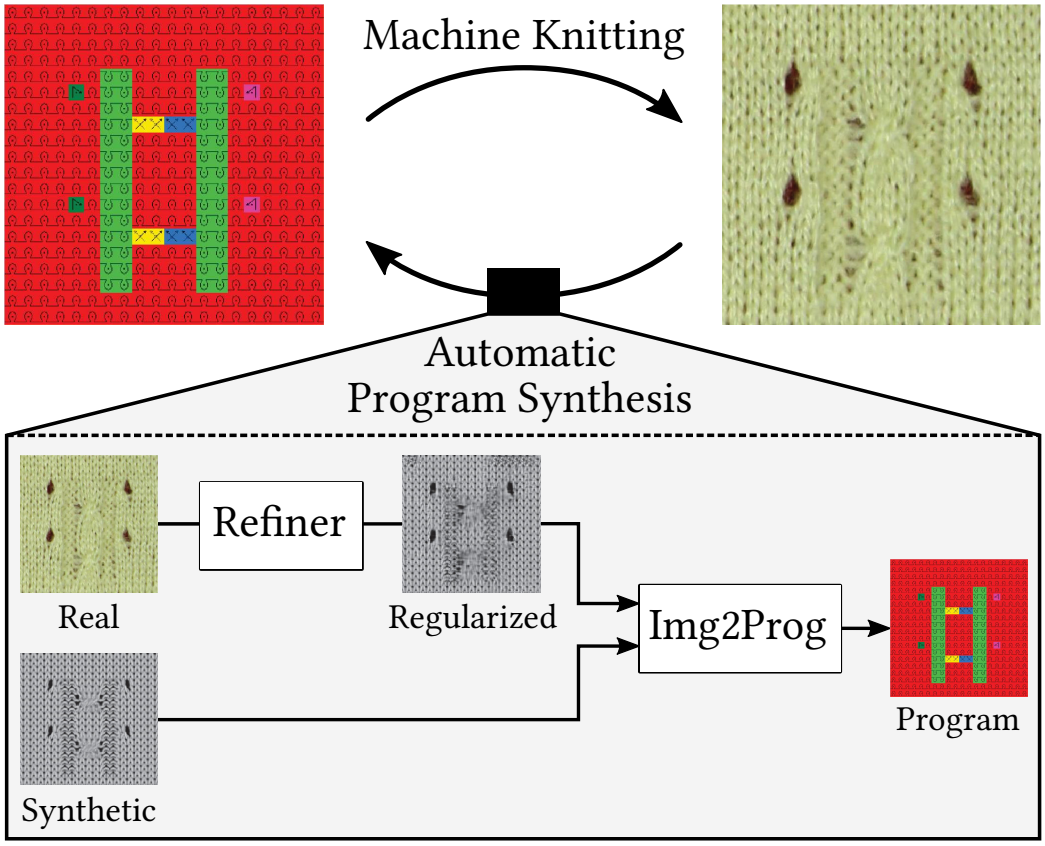


Figure 4-1: Illustration of our inverse problem and solution. An instruction map (*top-left*) is knitted into a physical artifact (*top-right*). We propose a machine learning pipeline to solve the inverse problem by leveraging synthetic renderings of the instruction maps.

4.1 Introduction

The workflow we envision starts with a user taking a picture of some knitted garment (or other knitted textile). We then reverse-engineer it to synthesize the corresponding knitting instructions. Given the instructions (or some higher-level description of the knitted artifact), the user can then either reproduce the original sample, or adapt it for a novel design. However, reverse-engineering a knitted design can be very challenging: knitted garments are typically not completely visible in a single image, they may include 3D effects due to shaping, they may include multiple kinds of yarns interacting in complicated manners, etc.

To simplify the problem, we consider the subspace of knitting programs that tackle simpler 2D knitting patterns, and we restrict ourselves to the use of a single type of

yarn in each pattern. The importance of 2D patterns in knitted textile is evident in pattern books [49, 154], that contain instructions for hundreds of decorative designs that have been manually crafted and tested over time. Unfortunately, these pattern libraries are geared towards hand-knitting and they are often incompatible with the operations of industrial knitting machines. Even in cases when a direct translation is possible, the patterns are only specified in stitch-level operation sequences that still need to be manually specified and tested for each machine type, similarly to low-level assembly programming.

Our main idea is to leverage the advances of recent works in machine learning that enable image translation across different domains [70, 78, 155, 194]. In Section 4.2, we describe our modeling of knitted patterns such that they become directly compatible with image translation mechanisms. Section 4.3 goes through our dataset acquisition process, highlighting one of our key challenges: getting enough supervision data. Sections 4.4 and 4.5 present our learning approach to combining different types of supervised data (real and synthetic) and the details of its implementation, illustrated in Figure 4-1. We evaluate our method in Section 4.6, and further discuss results and potential improvements in Section 4.7.

4.2 Machine Knitting Instructions

As described in Section 3.2, current knitting machine programming is done using low-level stitch codes that mix information about knitting operations (e.g., *knit*, *tuck*, *transfer*, etc.) and machine states (e.g., bed racking, speed, stitch tension). Incidentally, the design of these stitch codes appears to have been intentionally geared toward simplifying user patterning and often allows for a very regular descriptions of knitting patterns as 2D grids of stitch codes. Thus, we decided to create our own variation of those, targeted exclusively at stitch patterns, with the added intention of being machine-learnable.

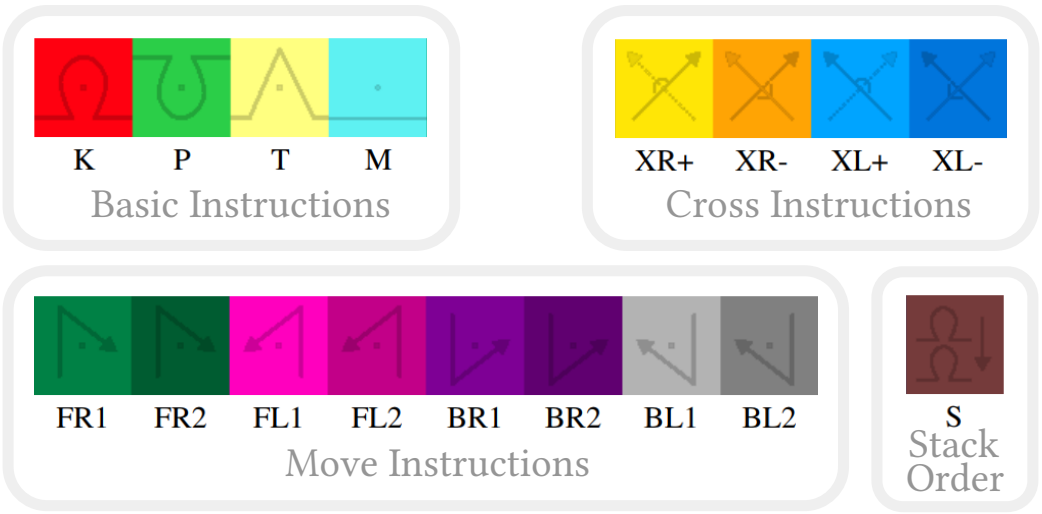
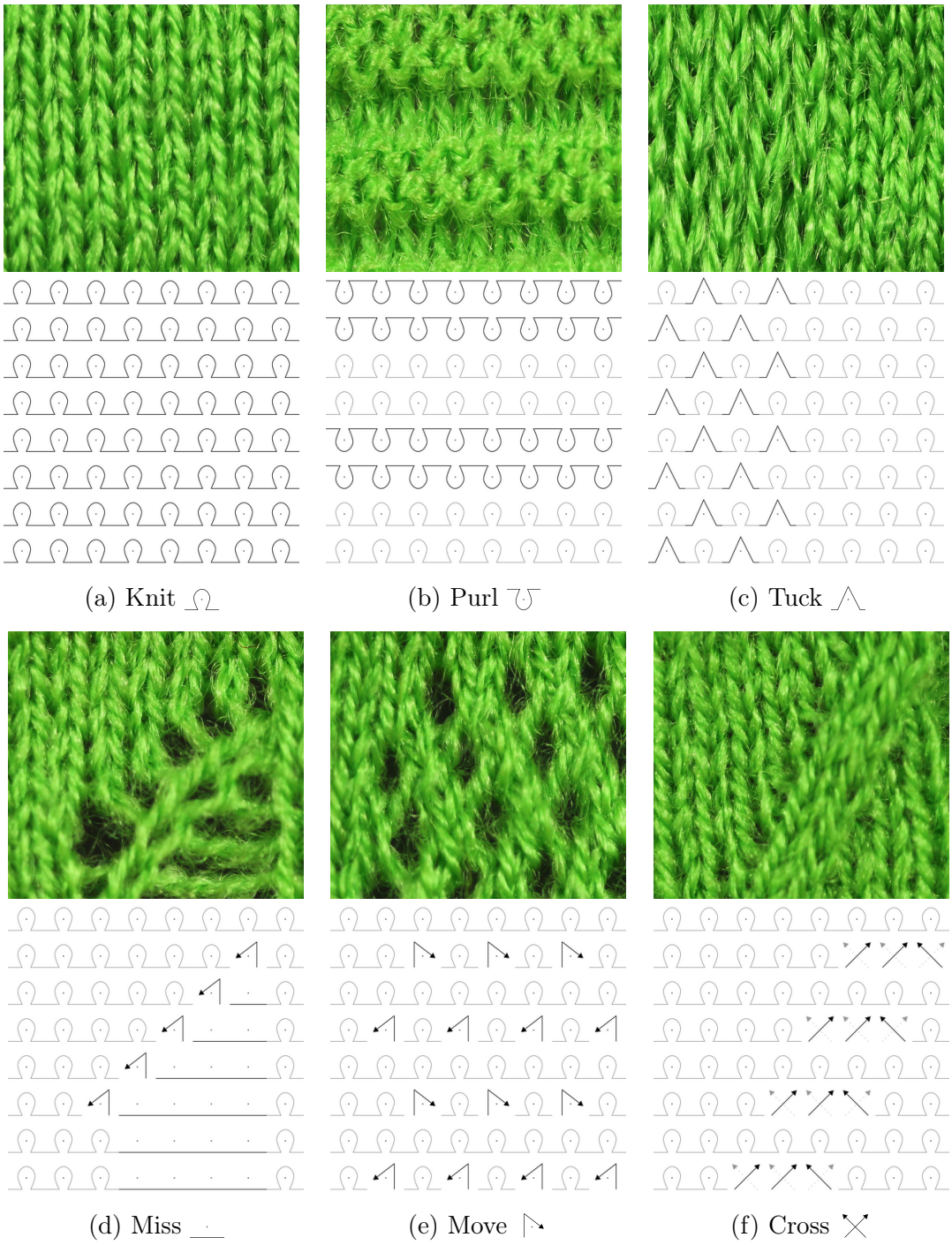


Figure 4-2: *Images*: abstract illustration and color coding of our 17 instructions. *Text*: instruction codes, which can be interpreted using the initial character of the following names: **K**nit and **P**url (front and back knit stitches), **T**uck, **M**iss, **F**ront, **B**ack, **R**ight, **L**eft, **S**tack. Finally, **X** stands for *Cross* where + and − are the ordering (upper and lower). *Move* instructions are composed of their initial knitting side (**F**ront or **B**ack), the move direction (**L**eft or **R**ight) and the offset (1 or 2).

4.2.1 A Domain-Specific Language for Patterns

In order to make our inverse design process tractable and efficient, we devised a set of 17 pattern instructions (derived from a subset of the hundreds of instructions from Shima Seiki [150]). These instructions include all basic knitting pattern operations and they are specified on a regular 2D grid that can be parsed and executed line-by-line. We first detail our pattern instruction set, visualized in Figure 4-2, and then we explain how they are sequentially processed by the machine.

Basic Instructions. These include the three base needle operations **Knit**, **Tuck** and **Miss**. Then, patterning books [49, 154] all suggest an additional **Purl** operation that combines the basic *Knit* with a sequence of transfers making it happen on the opposite needle bed. While the needle operation is the same as *Knit*, the fact that it happens on the opposite bed has a big impact on the resulting stitch appearance, as shown in Figure 4-3.



(a) Knit (b) Purl (c) Tuck

(d) Miss (e) Move (f) Cross

Figure 4-3: The main stitch operations with 8×8 pattern illustrations, both as a knitted artifact (top) and a colorless diagram (bottom).

Move and Cross. The combination of transfers with *racking* allows the movement of loops within a same bed and is extensively used in lace patterns that create local holes and changes of wale flow. We separate such higher-level operations into two

groups: **Move** instructions only consider combinations that do not cross other such instructions so that their relative scheduling does not matter, and **Cross** instructions are done in pairs so that both sides are swapped, producing what is known as *cable* patterns. The scheduling of *Cross* instructions is naturally defined by the instructions themselves. These combined operations do not create any new loop by themselves, and thus we assume they all apply a *Knit* operation before executing the associated needle moves, so as to maintain spatial regularity, similarly to Shima [150].

Stacking Order. Finally, transfers also allow different stacking orders when multiple loops are joined together. We model this with our final **Stack** instruction, and similarly to *Move* and *Cross*, it also additionally creates a new *Knit* stitch for spatial regularity. The corresponding symbols and color coding of the instructions are shown in Figure 4-2, whereas Figure 4-3 illustrates each of the pattern instruction types.

4.2.2 From High- to Low-level Instructions

Given a line of instructions, the sequence of operations is done over a full line using the following steps:

1. The current stitches are transferred to the new instruction side without racking;
2. The base needle operation (*Knit*, *Tuck* or *Miss*) is executed;
3. The needles of all transfer-related instructions are transferred to the opposite bed without racking;
4. Instructions that involve moving within a bed proceed to transfer back to the initial side using the appropriate racking and order;
5. Stack instructions transfer back to the initial side without racking.

Instruction Side

The only instructions requiring an associated bed *side* are those performing a *Knit* operation. We thus encode the bed side in the instructions (*Knit*, *Purl*, *Moves*),

except for those where the side can be inferred from the local context. This inference applies to *Cross* which use the same side as past instructions (for aesthetic reasons), and *Stack* which uses the side of its associated *Move* instruction. Although this is a simplification of the design space, we did not have any pattern with a different behavior.

Differences from Shima

While our instruction set is greatly inspired by the patterning codes available from Shima, there are two main differences: (1) it is restricted to pattern-related operations and (2) it does not introduce ambiguities. Shima's instructions include many shaping operations as well as complex context-sensitive variations of stitch codes (e.g. their link processing concept). Furthermore, their cable instructions are ambiguous: they form explicit pairs that are distinguished by requiring separate pairing codes for adjacent cables, whereas we rely exclusively on the direction as a pairing mechanism.

4.3 Dataset for Knitting Patterns

While machine knitting can produce a large amount of pattern data reasonably quickly, we still need to specify these patterns (i.e., generate reasonable pattern instructions), and acquire calibrated images for supervised learning. The latter acquisition ends up being the main bottleneck towards obtaining enough supervision data if we restrict ourselves to real images of knitted patterns. Thus, we augment the real data with synthetic data that can be acquired more efficiently.

4.3.1 Pattern Instructions

We extracted pattern instructions from the proprietary software KnitPaint [150]. These patterns have various sizes and span a large variety of designs from cable patterns to pointelle stitches, lace, and regular reverse stitches.

Given this set of initial patterns (around a thousand), we normalized the patterns by computing crops of 20×20 instructions with 50% overlap, while using default

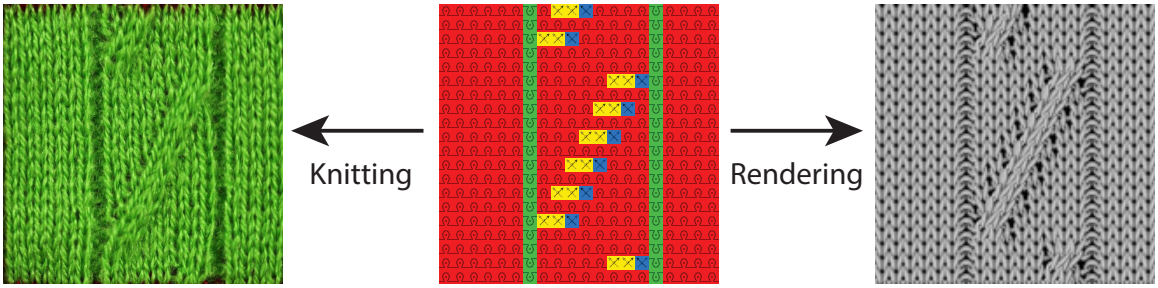


Figure 4-4: Different parts of our dataset (from left to right): real data images, machine instructions, and black-box rendering.

front stitches for the background of smaller patterns. This provided us with 12,392 individual 20×20 patterns (after pruning invalid patterns since random cropping can destroy the structure).

We then generated the corresponding images in two different ways: (1) by knitting a subset of 1,044 patches, i.e., *real* data, and (2) by rendering all of them using the basic pattern preview from KnitPaint, i.e., *synthetic*, simulated data. Figure 4-4 illustrates an example of supervised data triplet.

4.3.2 Knitting Many Samples

An important consideration for capturing knitted patterns is that their tension should be as regular as possible so that knitting units would align with corresponding pattern instructions. We initially proceeded with knitting and capturing patterns individually but this proved to not be scalable. Over an initial 1-month period, we produced and captured 372 patterns (front and back of 186 knitted samples) with a pattern-by-pattern capture setup shown in Figure 4-5. The individual patterns were framed with 2-colors intarsia to make the course/wale mapping easier (or potentially automate it completely). While knitting was fast, the capture bottleneck ended up being image acquisition, which required a careful sample-by-sample installation and tensioning with bowel pins.

We then chose to knit sets of 25 patterns over a 5×5 tile grid, each of which would be separated by both horizontal and vertical tubular knit structures. The tubular structures are designed to allow sliding 1/8 inch steel rods which we use

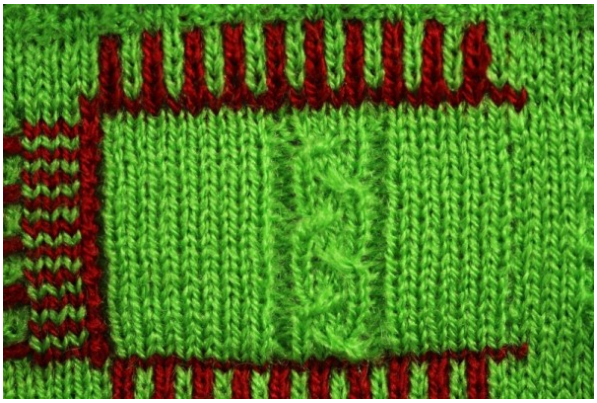



Figure 4-5: Our initial capture setup and a sample pattern illustrating the frame made of intarsia. The pattern tension was controlled with bowel pins inserted at specific holes that were programmed in the fabric.

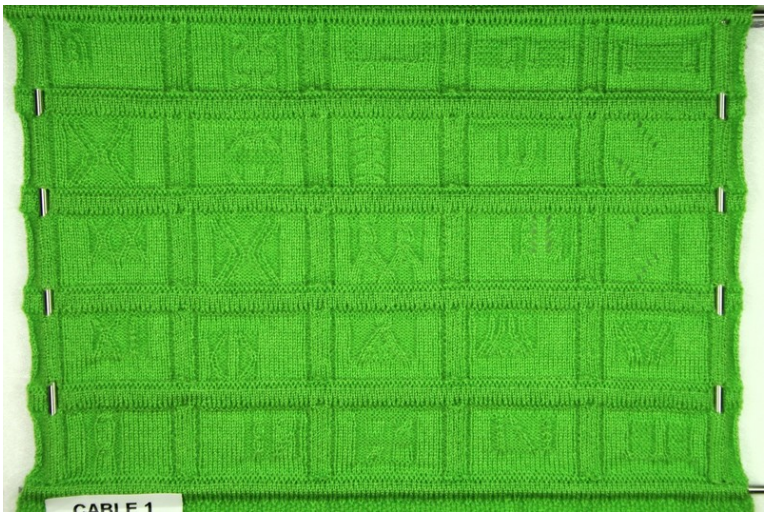



Figure 4-6: Our updated capture setup and a sample of 5×5 knitted patterns with tension controlled by steel rods. In many cases, corner rods were sufficient, whereas more complicated patterns required additional internal rods to reduce the local deformations.

to normalize the tension, as shown in Figure 4-6. Over a week, we produced and captured an additional 1716 valid patterns (front and back of 858 knitted samples). We had to discard some of the patterns in a few sheets as they were too distorted or had local yarn failure. The image registration took an additional week to finally result in our complete 2,088 real samples (including the initial 372 patterns). The frequency of different instruction types is shown in Figure 4-7.

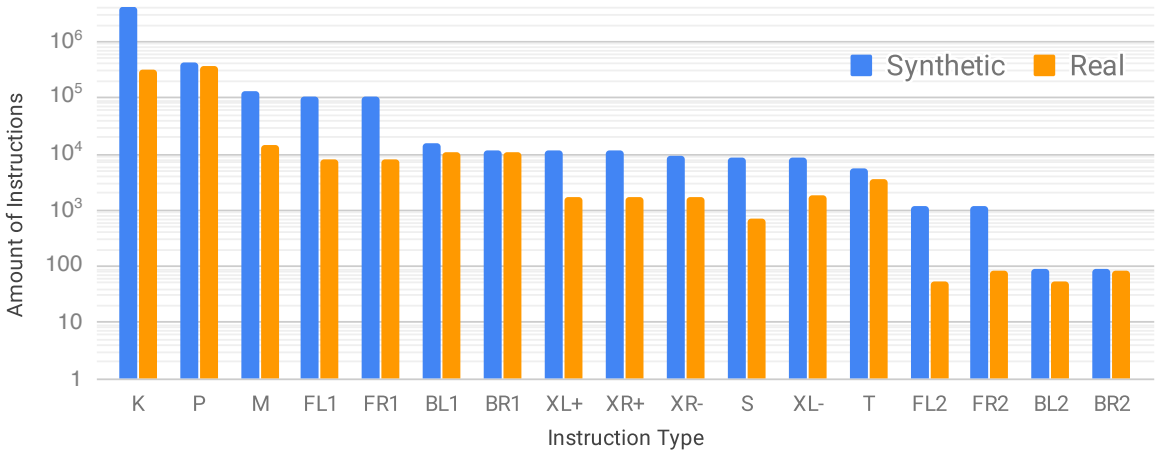


Figure 4-7: Instruction counts in decreasing order, for synthetic and real images. Note the logarithmic scale of the Y axis.

4.4 Learning Framework

In this section, we present our deep neural network model that infers a 2D knitting instruction map from an image of a knitted pattern. We first provide the theoretical motivation of our framework, and then we describe the loss functions we used.

4.4.1 Learning from Different Domains

When we have a limited number of real data, it is appealing to leverage simulated data because high quality annotations are automatically available. However, learning from synthetic data is problematic due to apparent domain gaps between synthetic and real data. We study how we can further leverage simulated data. We are motivated by the recent work, Simulated+Unsupervised (S+U) learning [155], but in contrast to them, we develop our framework from the generalization error perspective.

Let $\mathcal{X}$ be input space (image), and $\mathcal{Y}$ output space (instruction label), and $\mathcal{D}$ a data distribution on $\mathcal{X}$ paired with a true labeling function $y_{\mathcal{D}}: \mathcal{X} \to \mathcal{Y}$. As a typical learning problem, we seek a hypothesis classifier $h: \mathcal{X} \to \mathcal{Y}$ that best fits the target function y in terms of an expected loss: $\mathcal{L}_{\mathcal{D}}(h, h') = \mathbb{E}_{x \sim \mathcal{D}}[l\left(h(x), h'(x)\right)]$ for classifiers h, h', where $l: \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}_+$ denotes a loss function. We denote its empirical loss as $\hat{\mathcal{L}}_{\hat{\mathcal{D}}}(h, h') = \frac{1}{|\hat{\mathcal{D}}|} \sum_{i=1}^{|\hat{\mathcal{D}}|} l(h(x_i), h'(x_i))$, where $\hat{\mathcal{D}} = \{x\}$ is the sampled dataset.

In our problem, since we have two types of data available, a source domain $\mathcal{D}_S$ and

a target domain $\mathcal{D}_T$ (which is real or simulated as specified later), our goal is to find h by minimizing the combination of empirical source and target losses as α-mixed loss, $\hat{\mathcal{L}}_\alpha(h,y) = \alpha\hat{\mathcal{L}}_S(h,y) + (1-\alpha)\hat{\mathcal{L}}_T(h,y)$, where $0\leq\alpha\leq1$, and for simplicity we shorten $\mathcal{L}_{\mathcal{D}_{\{S,T\}}}=\mathcal{L}_{\{S,T\}}$ and we use the parallel notation $\mathcal{L}_{\{S,T\}}$ and $\hat{\mathcal{L}}_{\{S,T\}}$. Our underlying goal is to achieve a minimal generalized target loss $\mathcal{L}_T$. To develop a generalizable framework, we present a bound over the target loss in terms of its empirical α-mixed loss, which is a slight modification of Theorem 3 of Ben-David et al. [19].

Theorem 1. *Let $\mathcal{H}$ be a hypothesis class, and $\mathcal{S}$ be a labeled sample of size m generated by drawing βm samples from $\mathcal{D}_S$ and $(1-\beta)m$ samples from $\mathcal{D}_T$ and labeling them according to the true label y. Suppose $\mathcal{L}$ is symmetric and obeys the triangle inequality. Let $\hat{h} \in \mathcal{H}$ be the empirical minimizer of $\hat{h} = \arg\min_h \hat{\mathcal{L}}_\alpha(h,y)$ on $\mathcal{S}$ for a fixed $\alpha \in [0,1]$, and $h_T^* = \arg\min_h \mathcal{L}_T(h,y)$ the target error minimizer. Then, for any $\delta \in (0,1)$, with probability at least $1-\delta$ (over the choice of the samples), we have*

$$\tfrac{1}{2}|\mathcal{L}_T(\hat{h},y) - \mathcal{L}_T(h_T^*,y)| \leq \alpha\left(\mathrm{disc}_{\mathcal{H}}(\mathcal{D}_S,\mathcal{D}_T) + \lambda\right) + \epsilon, \tag{4.1}$$

where $\epsilon(m,\alpha,\beta,\delta) = \sqrt{\frac{1}{2m}\left(\frac{\alpha^2}{\beta} + \frac{(1-\alpha)^2}{1-\beta}\right)\log(\frac{2}{\delta})}$, *and* $\lambda = \min_{h\in\mathcal{H}} \mathcal{L}_S(h,y) + \mathcal{L}_T(h,y)$.

Compared to Ben-David et al. [19], Theorem 1 is purposely extended to use a more general definition of discrepancy $\mathrm{disc}_{\mathcal{H}}(\cdot,\cdot)$ [111] that measures the discrepancy of two distributions and to be agnostic to the model type (simplification), so that we can clearly present our motivation of our model design. The proof of Theorem 1 together with the discrepancy definition can be found in Appendix A.

Theorem 1 shows that mixing two sources of data is possible to achieve a better generalization in the target domain. The bound is always at least as tight as either of $\alpha = 0$ or $\alpha = 1$ (The case that uses either source or target dataset alone). Also, as the total number of the combined data sample m is larger, a tighter bound can be obtained. We consider 0-1 loss for l in this section for simplicity, but not limited to.

A factor that the generalization gap (the right hand side in Equation 4.1) strongly depends on is the discrepancy $\mathrm{disc}_{\mathcal{H}}(\mathcal{D}_S,\mathcal{D}_T)$. This suggests that we can achieve a tighter bound if we can reduce $\mathrm{disc}_{\mathcal{H}}(\mathcal{D}_S,\mathcal{D}_T)$. We re-parameterize the target distri-

bution $\mathcal{D}_T$ as $\mathcal{D}_R$ so that $\mathcal{D}_T{=}g \circ \mathcal{D}_R$, where g is a distribution mapping function. Then, we find the mapping g^* that leads to the minimal discrepancy for the empirical distribution $\hat{\mathcal{D}}_R$ as:

$$\begin{aligned} g^* &= \arg\min_g \mathrm{disc}_{\mathcal{H}}(\hat{\mathcal{D}}_S,\ g \circ \hat{\mathcal{D}}_R) \\ &= \arg\min_g \max_{h,h' \in \mathcal{H}} |\mathcal{L}_{\hat{\mathcal{D}}_S}(h,h') - \mathcal{L}_{g \circ \hat{\mathcal{D}}_R}(h,h')|, \end{aligned} \tag{4.2}$$

which is a min-max problem. Even though the problem is defined for an empirical distribution, it is intractable to search the entire solution space; thus, motivated by Ganin et al. [54], we approximately minimize the discrepancy by Generative Adversarial Networks (GAN) [58]. Therefore, deriving from Theorem 1, our empirical minimization is formulated by minimizing the convex combination of source and target domain losses as well as the discrepancy as:

$$\hat{h}, \hat{g} = \arg\min_{h \in \mathcal{H}, g \in \mathcal{G}} \hat{\mathcal{L}}_\alpha(h,y) + \tau \cdot \mathrm{disc}_{\mathcal{H}}(\hat{\mathcal{D}}_S, g \circ \hat{\mathcal{D}}_R). \tag{4.3}$$

Along with leveraging GAN, our key idea for reducing the discrepancy between two data distributions, i.e., domain gap, is to transfer the real knitting images (target domain, $\hat{\mathcal{D}}_R$) to synthetic looking data (source domain, $\hat{\mathcal{D}}_S$) rather than the other way around, i.e., making $\hat{\mathcal{D}}_S \approx \hat{g} \circ \hat{\mathcal{D}}_R$. The previous methods have investigated generating realistic looking images to adapt the domain gap. However, we observe that, when simulated data is mapped to real data, the mapping is a one-to-many mapping due to real-world effects, such as lighting variation, geometric deformation, background clutter, noise, etc. This introduces an unnecessary challenge to learn $g(\cdot)$; thus, we instead learn to neutralize the real-world perturbation by mapping from real data to synthetic looking data. Beyond simplifying the learning of $g(\cdot)$, it also allows the mapping to be utilized at test time for processing of real-world images.

We implement h and g using convolutional neural networks (CNN), and formulate the problem as a local instruction classification[1] and represent the output as a 2D

[1]While our program synthesis deals with multiple instruction classes, for simplicity, we consider the binary classification here, and this can be extended to multiple classes [151].

array of classification vectors $\vec{\mathbf{s}}_{(i,j)} \in [0;1]^K$ (i.e., softmax values over $k \in K$) for our $K = 17$ instructions at each spatial location (i, j). In the following, we describe the loss we use to train our model $h \circ g$ and then we detail the training procedure.

4.4.2 Loss Function

We use the cross entropy for the loss $\mathcal{L}$. We supervise the inferred instruction to match the ground-truth instruction using the standard multi-class cross-entropy $\mathrm{CE}(\vec{\mathbf{s}}, \vec{\mathbf{y}}) = -\sum_k y_k \log(s_k)$ where s_k is the predicted likelihood (softmax value) for instruction k, which we compute at each spatial location (i, j).

For synthetic data, we have precise localization of the predicted instructions. In the case of the real knitted data, human annotations are imperfect and this can cause a minor spatial misalignment of the image with respect to the original instructions. For this reason, we allow the predicted instruction map to be globally shifted by up to one instruction. In practice, motivated by multiple instance learning [48], we consider the minimum of the per-image cross-entropy over all possibles one-pixel shifts (as well as the default no-shift variant), i.e., our complete cross entropy loss is

$$\mathcal{L}_{\mathrm{CE}} = \frac{1}{Z_{CE}} \min_d \sum\nolimits_{i,j \in \mathcal{N}_s} \mathrm{CE}(\vec{\mathbf{s}}_{(i,j)+d}, \vec{\mathbf{y}}_{(i,j)}), \tag{4.4}$$

where $d \in \{(dx, dy) \,|\, dx, dy \in \{-1, 0, +1\}\}$ is the pattern displacement for the real data and $d \in \{(0, 0)\}$ for the synthetic data. The loss is accumulated over the spatial domain $\mathcal{N}_s = \{2, \ldots, w-1\} \times \{2, \ldots, h-1\}$ for the instruction map size $w \times h$ reduced by boundary pixels. $Z_{CE} = |\mathcal{N}_s|$ is a normalization factor.

4.5 Implementation Details

The base architecture is illustrated in Figure 4-1. We implemented it using TensorFlow [1]. The prediction network `Img2prog` takes 160×160 grayscale images as input and generates 20×20 instruction maps. The structure consists of an initial set of 3 convolution layers with stride 2 that downsample the image to 20×20 spatial resolu-

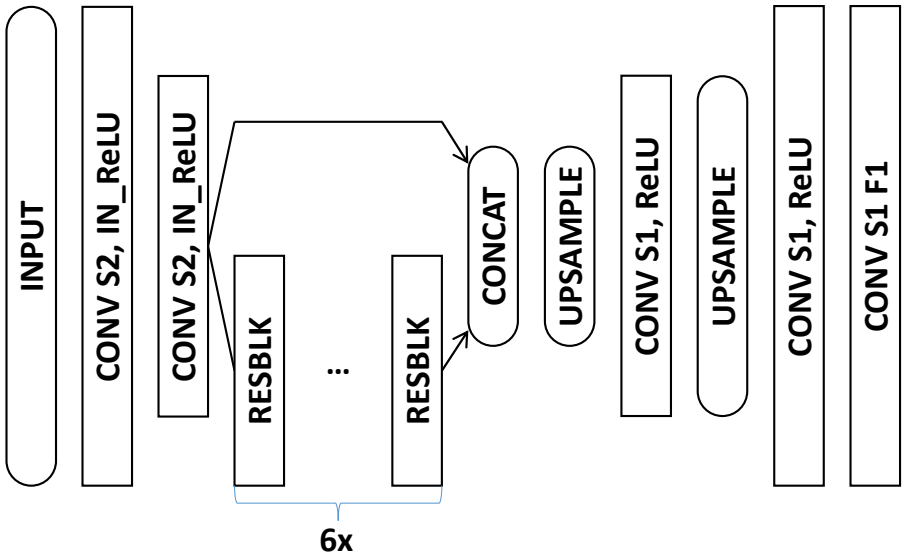


Figure 4-8: The illustration of the `Refiner` network architecture, where S#N denotes the stride size of #N, `IN_ReLU` indicates the Instance normalization followed by ReLU, `Resblk` is the residual block that consists of ConvS1-ReLU-ConvS1 with shortcut connection [66], Upsample is the nearest neighbor upsampling with the factor $2\times$, F is the output channel dimension. If not mentioned, the default parameters for all the convolutions are the stride size of 2, $F = 64$, and the 3×3 kernel size.

tion, a feature transformation part made of 6 residual blocks [66, 194], and two final convolutions producing the instructions. The kernel size of all convolution layers is 3×3, except for the last layer which is 1×1. We use instance normalization [168] for each of the initial down-convolutions, and ReLU everywhere.

We solve the minimax problem of the discrepancy $\text{disc}(\cdot,\cdot)$ w.r.t. g using the least-square Patch-GAN [194]. Additionally, we add the perceptual loss and style loss [82] between input real images and its generated images and between simulated images and generated images, respectively, to regularize the GAN training, which stably speeds up the training of g.

4.5.1 The `Refiner` Network

Our refinement network translates real images into regular images that look similar to synthetic images. Its implementation is similar to `Img2prog`, except that it outputs the same resolution image as input. The layer configuration is shown in Figure 4-8.

4.5.2 Loss Balancing Parameters

When learning our full architecture with both `Refiner` and `Img2prog`, we have three different losses: the cross-entropy loss $\mathcal{L}_{CE}$, the perceptual loss $\mathcal{L}_{\text{Perc}}$, and the Patch-GAN loss.

Our combined loss is the weighted sum

$$\mathcal{L} = \lambda_{\text{CE}}\mathcal{L}_{\text{CE}} + \lambda_{\text{Perc}}\mathcal{L}_{\text{Perc}} + \lambda_{\text{GAN}}\mathcal{L}_{\text{GAN}} \tag{4.5}$$

where we used the weights: $\lambda_{\text{CE}} = 3$, $\lambda_{\text{Perc}} = 0.02/(128)^2$ and $\lambda_{\text{GAN}} = 0.2$. The losses $\mathcal{L}_{\text{Perc}}$ and λ_{GAN} are measured on the output of `Refiner`, while the loss λ_{CE} is measured on `Img2prog`.

The perceptual loss [82] consists of the feature matching loss and style loss (using the gram matrix). If not mentioned here, we follow the implementation details of [82], where VGG-16 [156] is used for feature extraction, after replacing max-pooling operations with average-pooling. The feature matching part is done using the `pool3` layer, comparing the input real image and the output of `Refiner` so as to preserve the content of the input data. For the style matching part, we use the gram matrices of the {`conv1_2`, `conv2_2`, `conv3_3`} layers with the respective relative weights {0.3, 0.5, 1.0}. The measured style loss is between the synthetic input image and the synthetic-like output of the `Refiner` network.

For $\mathcal{L}_{\text{GAN}}$ and the loss for the discriminator, the least-square Patch-GAN loss [194] is used. We used $\{-1, 1\}$ for the regression labels for respective fake and real samples insted of the label $\{0, 1\}$ used in [194].

For training, we normalize the loss λ_{CE} to be balanced according to the data ratio of a batch. Specifically, for example, suppose a batch consisting of 2 real and 4 synthetic samples, respectively. Then, we inversely weighted the respective cross entropy losses for real and synthetic data by the weights of $\frac{4}{6}$ and $\frac{2}{6}$, so that the effects from the losses are balanced. This encourages the best performance to be expected at near $\alpha = 0.5$ within a batch.

4.5.3 Data Augmentation

We use multiple types of data augmentation to notably increase the diversity of yarn colors, lighting conditions, yarn tension, and scale:

- **Global Crop Perturbation**: we add random noise to the location of the crop borders for the real data images, and crop on-the-fly during training; the noise intensity is chosen such that each border can shift at most by half of one stitch;
- **Local Warping**: we randomly warp the input images locally using non-linear warping with linear RBF kernels on a sparse grid. We use one kernel per instruction and the shift noise is a 0-centered gaussian with σ being $1/5$ of the default instruction extent in image space (i.e. $\sigma = 8/5$);
- **Intensity augmentation**: we randomly pick a single color channel and use it as a mono-channel input, so that it provides diverse spectral characteristics. Also note that, in order to enhance the intensity scale invariance, we apply instance normalization [168] for the upfront convolution layers of our encoder network.

4.5.4 Training Procedure

We train our network with a combination of the real knitted patterns and the rendered images. We oversample the real data to achieve 1:1 mix ratio with the previously mentioned data augmentation strategies. We train with 80% of the real data, withholding 5% for validation and 15% for testing, whereas we use all the synthetic data for training.

According to the typical training method for GAN [58], we alternate the training between discriminator and the other networks, h and g, but we update the discriminator only every other iteration, and the iteration is counted according to the number of updates for h and g.

We trained our model for $150k$ iterations with batch size 2 for each domain data using ADAM optimizer with initial learning rate 0.0005, exponential decay rate 0.3

every $50,000$ iterations. The training took from 3 to 4 hours (depending on the model) on a Titan XP GPU.

4.5.5 Data Post-Processing

The presented framework does not enforce hard constraint on the output semantics. This implies that some outputs may not be machine-knittable as-is.

More precisely, the output of our network may contain invalid instructions pairs or a lack thereof. We remedy to these conflicts by relaxing the conflicting instruction, which happens in only two cases:

1. Unpaired Cross instructions – we reduce such instructions into their corresponding Move variants (since Cross are Moves with relative scheduling), and

2. Cross pairs with conflicting schedules (e.g., both pair sides have same priority, or instructions within a pair's side having different priorities) – in this case, we randomly pick a valid schedule (note that its impact is only local).

This is sufficient to allow knitting on the machine. Note that Stack are semantically *supposed* to appear with a Move, but they don't prevent knitting since their operations lead to the same as Knit when unpaired, and thus do not require any specific post-processing.

4.6 Evaluation

We first evaluate baseline models for our new task, along with an ablation study looking at the impact of our loss and the trade-off between real and synthetic data mixing. Finally, we look at the impact of the size of our dataset..

Accuracy Metric For the same reason our loss in Eq. (4.4) takes into consideration a 1-pixel ambiguity along the spatial domain, we use a similarly defined accuracy. It is measured by the average of $\max_d \frac{1}{N_{\text{inst}}} \sum_{i,j} \mathbb{I}[y_{\text{GT}(i,j)} = \arg\max_k s^k_{(i,j)+d}]$ over the whole

dataset, where $N_{\text{inst}} = Z_{\text{CE}}$ is the same normalization constant as in Eq. (4.4), y_{GT} the ground-truth label, $\mathbb{I}[\cdot]$ is the indicator function that returns 1 if the statement is true, 0 otherwise. We report two variants: `FULL` averages over all the instructions, whereas `FG` considers all instructions but the background – i.e., we discard the most predominant instruction in the pattern.

Perceptual Metrics For the baselines and the ablation experiments, we additionally provide perceptual metrics that measure how similar the knitted pattern would look. An indirect method for evaluation is to apply a pre-trained neural network to generated images and calculate statistics of its output, e.g., Inception Score [144]. Inspired by this, we learn a separate network to render simulated images of the generated instructions and compare it to the rendering of the ground truth using standard PSNR and SSIM metrics. Similarly to the accuracy, we allow for one instruction shift, which translates to full 8 pixels shifts in the image domain.

4.6.1 Comparison to Baselines

Table 4.1 compares the measured accuracy of predicted instructions on our real image test set. We also provide qualitative results in Figure 4-10 and 4-11.

The first 5 rows of Table 4.1-(a1-5) present results of previous works to provide snippets of other domain methods. For CycleGAN, no direct supervision is provided and the domains are mapped in a fully unsupervised manner. Together with Pix2pix, the two first methods do not use cross-entropy but L1 losses with GAN. Although they can provide interesting image translations, they are not specialized for multi-class classification problems, and thus cannot compete. All baselines are trained from scratch. Furthermore, since their architectures use the same spatial resolution for both input and output, we up-sampled instruction maps to the same image dimensions using nearest neighbor interpolation.

S+U Learning [155] used a refinement network to generate a training dataset that makes existing synthetic data look realistic. In this case, our implementation uses our base network `Img2prog` and approximates real domain transfer by using style

Table 4.1: **Performance comparison to baseline methods on our real image test dataset.** The table shows translation invariant accuracy of the predicted instructions with and without the background and PSNR and SSIM metrics for the image reconstruction where available. More is better for all metrics used.

Method	Accuracy (%)		Perceptual	
	Full	**FG**	**SSIM**	**PSNR**
(a1) CycleGAN [194]	57.27	24.10	0.670	15.87 dB
(a2) Pix2Pix [78]	56.20	47.98	0.660	15.95 dB
(a3) UNet [138]	89.65	63.99	0.847	21.21 dB
(a4) Scene Parsing [193]	**91.58**	**73.95**	**0.876**	**22.64 dB**
(a5) S+U [155]	91.32	71.00	0.864	21.42 dB
(b1) `Img2prog` (real only) with CE	91.57	71.37	0.866	**21.62 dB**
(b2) `Img2prog` (real only) with MILCE	**91.74**	**72.30**	**0.871**	21.58 dB
(c1) Refiner + `Img2prog` ($\alpha = 0.9$)	93.48	78.53	0.894	**23.28 dB**
(c2) Refiner + `Img2prog` ($\alpha = 2/3$)	**93.58**	**78.57**	0.892	23.27 dB
(c3) Refiner + `Img2prog` ($\alpha = 0.5$)	93.57	78.30	**0.895**	23.24 dB
(c4) Refiner + `Img2prog` ($\alpha = 1/3$)	93.19	77.80	0.888	22.72 dB
(c5) Refiner + `Img2prog` ($\alpha = 0.1$)	92.42	74.15	0.881	22.27 dB
(d1) Refiner + `Img2prog`++ ($\alpha = 0.5$)	**94.01**	**80.30**	**0.899**	**23.56 dB**

transfer. We tried two variants: using the original Neural Style Transfer [55] and CycleGAN [194]. Both input data types lead to very similar accuracy (negligible difference) when added as a source of real data. We thus only report the numbers from the first one [55].

4.6.2 Impact of Loss and Data Mixing Ratio

The second group in Table 4.1-(b1-2) considers our base network h (`Img2prog`) without the refinement network g (`Refiner`) that translates real images onto the synthetic domain. In this case, `Img2prog` maps real images directly onto the instruction domain. The results generated by all direct image translation networks trained with cross-entropy (a3-5) compare similarly with our base `Img2prog` on both accuracy and perceptual metrics. This shows our base network allows a fair comparison with the competing methods, and as will be shown, our final performance (c1-5, d1) is not gained from the design of `Img2prog` but `Refiner`.

Table 4.2: **Performance of *Refined+Img2prog++* measured per instruction over the test set.** This shows that even though our instruction distribution has very large variations, our network is still capable of learning some representation for the least frequent instructions (3 orders of magnitude difference for FR2, FL2, BR2, BL2 compared to K and P).

Instruction	K	P	T	M	FR1	FR2	FL1	FL2
Accuracy [%]	96.52	96.64	74.63	66.65	77.16	100.0	74.20	83.33
Frequency [%]	44.39	47.72	0.41	1.49	1.16	0.01	1.23	0.01

Instruction	BR1	BR2	BL1	BL2	XR+	XR-	XL+	XL-	S
Accuracy [%]	68.73	27.27	69.94	22.73	60.15	62.33	60.81	62.11	25.85
Frequency [%]	1.22	0.02	1.40	0.02	0.22	0.18	0.19	0.22	0.12

The third group in Table 4.1-(c1-5) looks at the impact of the mixing ratio α when using our full architecture. In this case, the refinement network g translates our real image into a synthetic looking one, which is then translated by `Img2prog` into instructions. This combination favorably improves both the accuracy and perceptual quality of the results with the best mixing ratio of α=2/3 as well as a stable performance regime of $\alpha \in [0.5, 0.9]$, which favors more the supervision from diverse simulated data. While ϵ in Theorem 1 has a minimum at α=β, we have a biased α due to other effects, disc$(\cdot)$ and λ.

We tried learning the opposite mapping g (from synthetic image to realistic looking), while directly feeding real data to h. This leads to detrimental results with mode collapsing. The learned g maps to a trivial pattern and texture that injects the pattern information in invisible noise – i.e., adversarial perturbation – to enforce that h maintains a plausible inference. We postulate this might be due to the non-trivial one-to-many mapping relationship from simulated data to real data, and overburden for h to learn to compensate real perturbations by itself.

In the last row of Table 4.1-(d1), we present the result obtained with a variant network, `Img2prog++` which additionally uses skip connections from each down-convolution of `Img2prog` to increase its representation power. This is our best model in the qualitative comparisons of Figure 4-10.

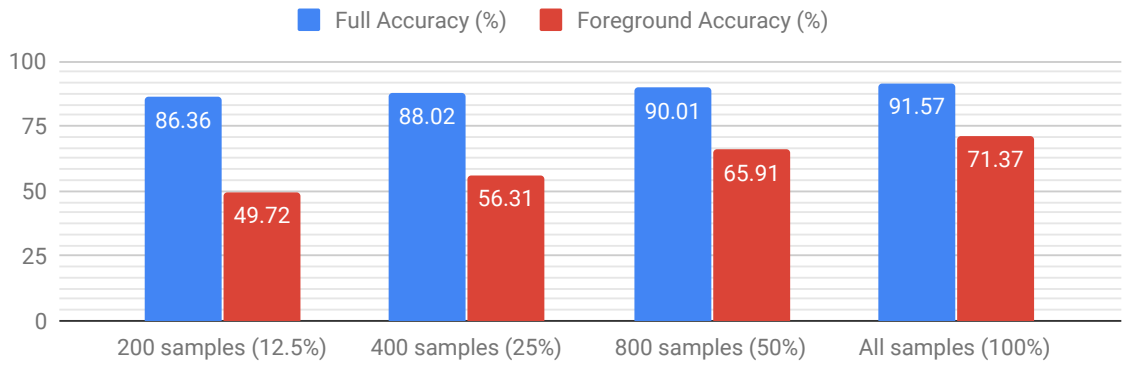


Figure 4-9: The impact of the amount of real training data (from 12.5% to 100% of the real dataset) over the accuracy.

Finally, we check the per-instruction behavior of our best model, shown through the per-instruction accuracy in Table 4.2. Although there is a large difference in instruction frequency, our method still manages to learn some useful representation for rare instructions but the variability is high. This suggests the need for a systematic way of tackling the class imbalance [72, 104].

4.6.3 Impact of Dataset Size

In Figure 4-9, we show the impact of the real data amount on accuracy. As expected, increasing the amount of training data helps (and we have yet to reach saturation). With low amounts of data (here 400 samples or less), the training is not always stable – some data splits lead to overfitting.

4.6.4 Larger Models

In our baseline, we compared with a sample architecture from [193], which we made small enough to compare with our baseline `Img2prog` implementation. Furthermore, our baseline implementations were all trained from scratch and did not make use of pre-training on any other dataset.

Here, we provide results for a much larger variant of that network, which we name *Large Scene Parsing*, and makes use of pre-training on ImageNet [143]. The quantitative comparison is provided in Table 4.3, which shows that we can achieve

Table 4.3: **Performance comparison with the larger scene parsing network from Zhou et al. [193]** . (d2) uses pre-training on ImageNet [143] and a much larger number of parameters (51.4M vs. 1.4M – with M for *Millions*).

Method	Accuracy (%)		Perceptual		Params
	Full	**FG**	**SSIM**	**PSNR**	
(d1) Refiner + img2prog++ ($\alpha = 1/2$)	94.01	80.30	0.899	23.56 dB	1.4 M
(d2) Large Scene Parsing with pre-training [193]	**94.95**	**83.46**	**0.908**	**24.58 dB**	51.4 M

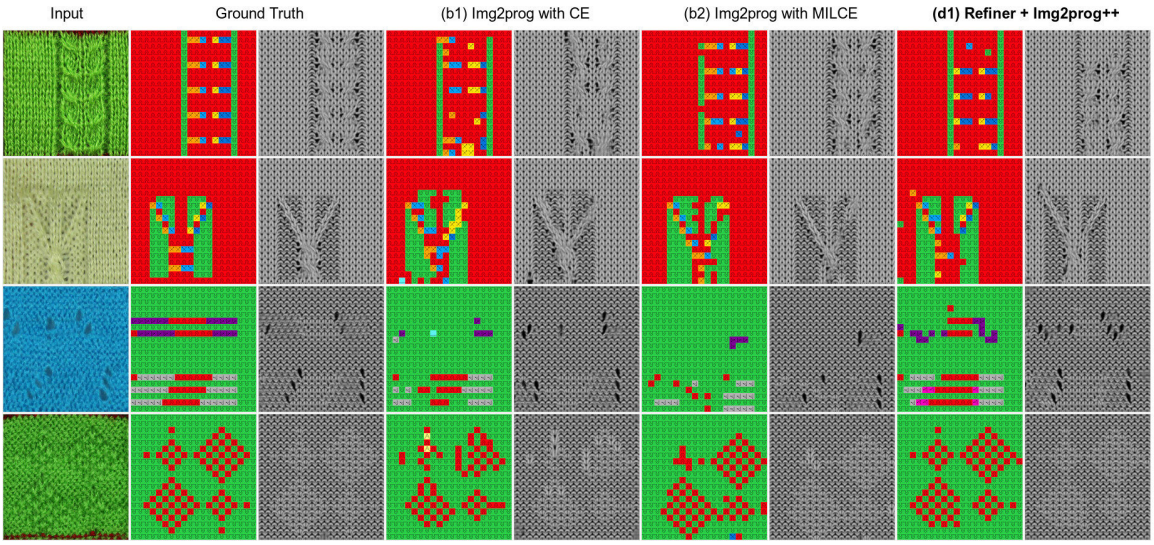


Figure 4-10: Comparisons of instructions predicted by different versions of our method. We present the predicted instructions as well as a corresponding image from our renderer.

even better accuracy than our best current results using our `Refiner`+`Img2prog`++ combination. However, note that this comes with a much larger model size: ours has 1.4M parameters, whereas *Large Scene Parsing* has 51.4M. Furthermore, this requires pre-training on ImageNet with millions of images (compared to our model working with a few thousands only).

4.6.5 Knitting the Inferred Programs

While the accuracy and perceptual metrics provide some important quantitative information, they are incomplete in describing the extents of what can be done with

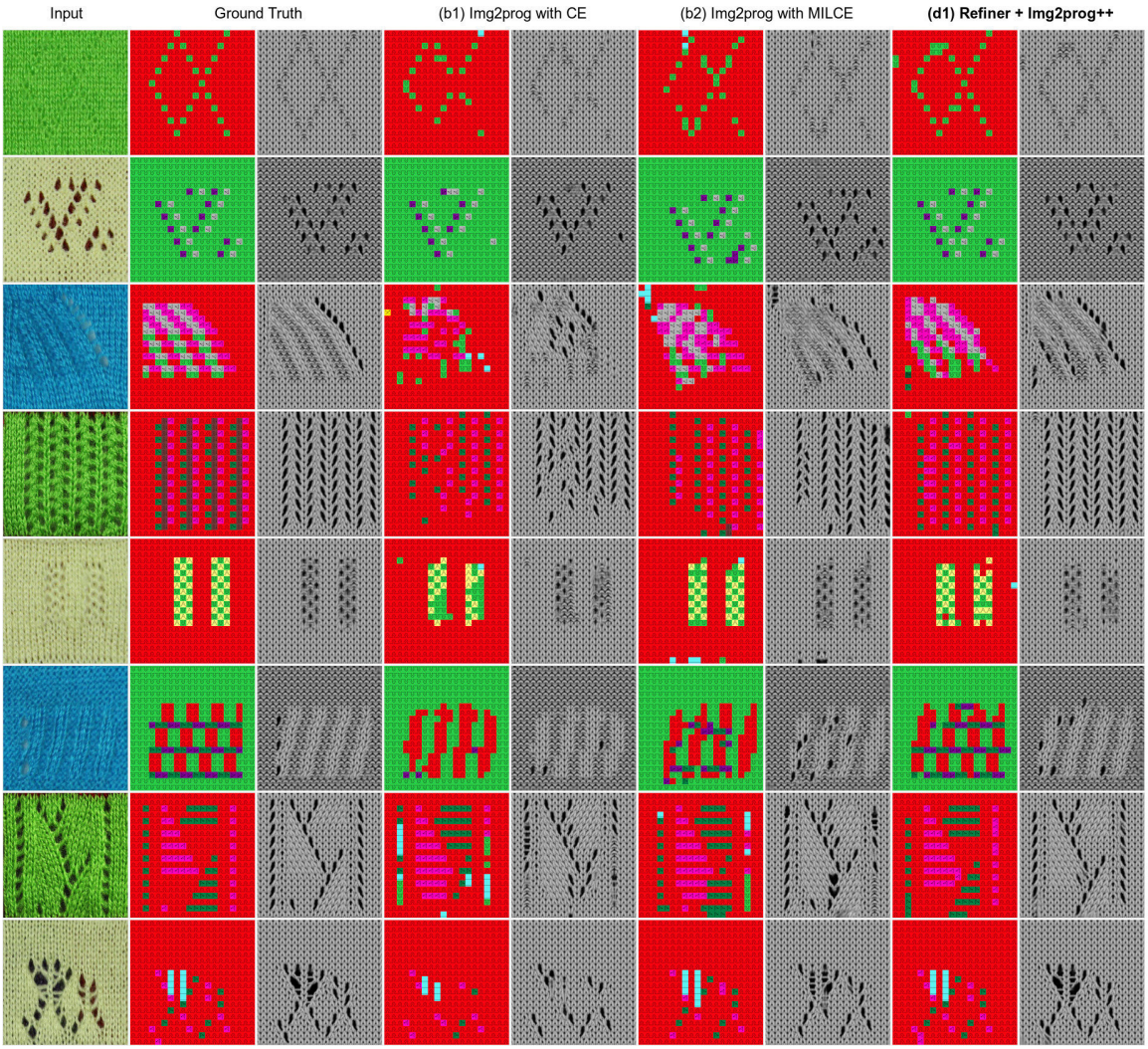


Figure 4-11: Additional qualitative comparisons of instructions predicted by different versions of our method, with their renderings.

the current system. Thus, we explicitly knitted a few samples from the inferred programs from the test set. The test results can be grouped into four categories: *perfect* results, *minor* errors that do not induce large deformations, *larger* errors that change the topology but are fine to knit, and *catastrophic* errors that are dangerous to knit. Most of the test results have minor or major errors – they create slight variations in the pattern but are mostly knittable. Figure 4-12 illustrates a few *larger* inference errors and the result after knitting them. Interestingly, although the programs are visibly different, the knitted results appear a lot similar to the original samples. Figure 4-13 shows the only two test samples we consider as catastrophic examples.

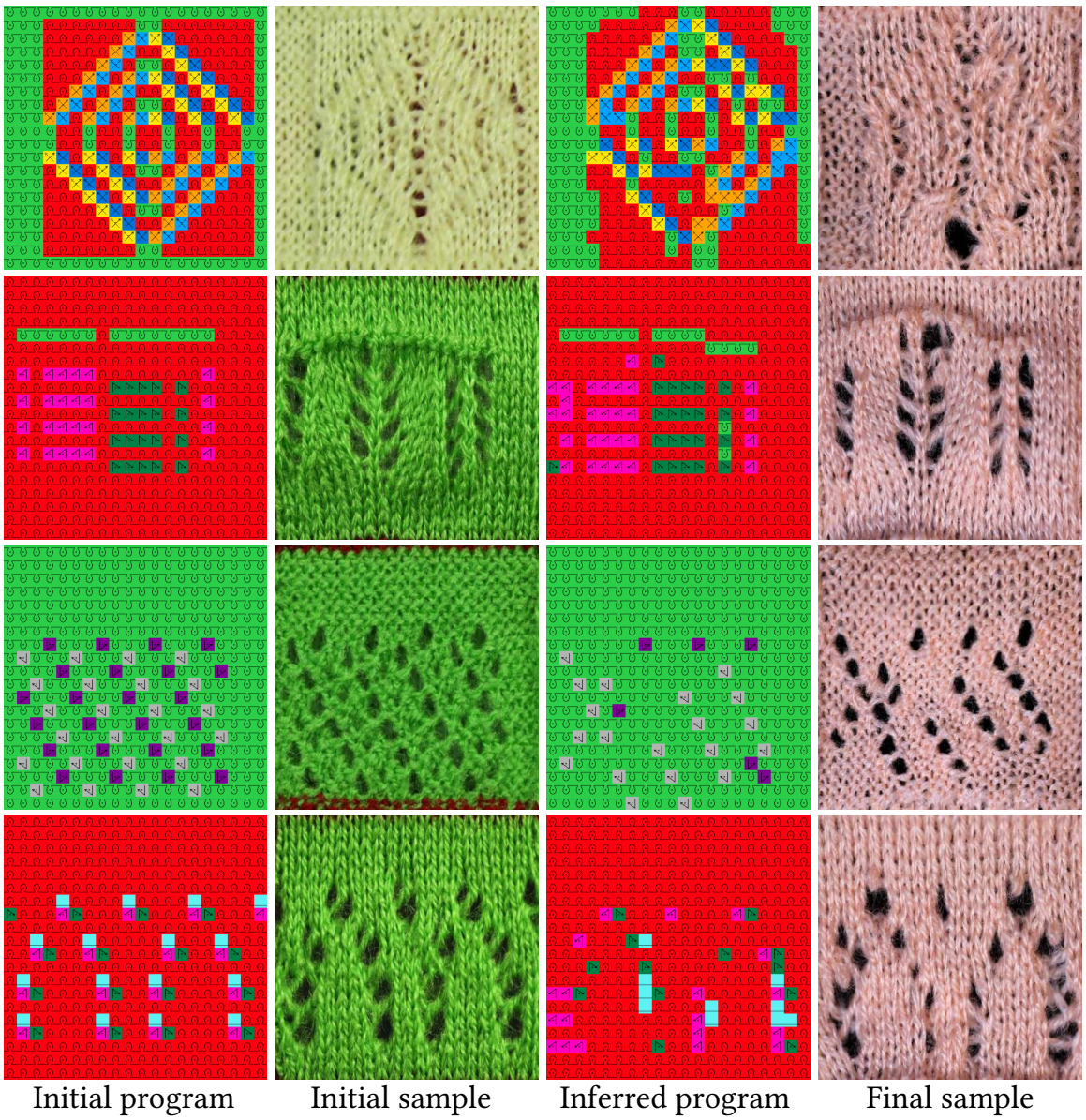


Figure 4-12: Examples of erroneously inferred programs that are still knittable.

Knitting them would fail and may likely break needles. The first one illustrates a limitation of one of our assumptions, namely that the *reverse* pattern taken from the back image of a stitch pattern may not necessarily "classify" as a proper knitting pattern from the aesthetic perspective. The second one showcases complicated long-distance interactions and a failure to recover them – notably move instructions used for lace and holes. Knitting any of those two would likely lead to failures and more complicated yarn pile-up, with broken needles.

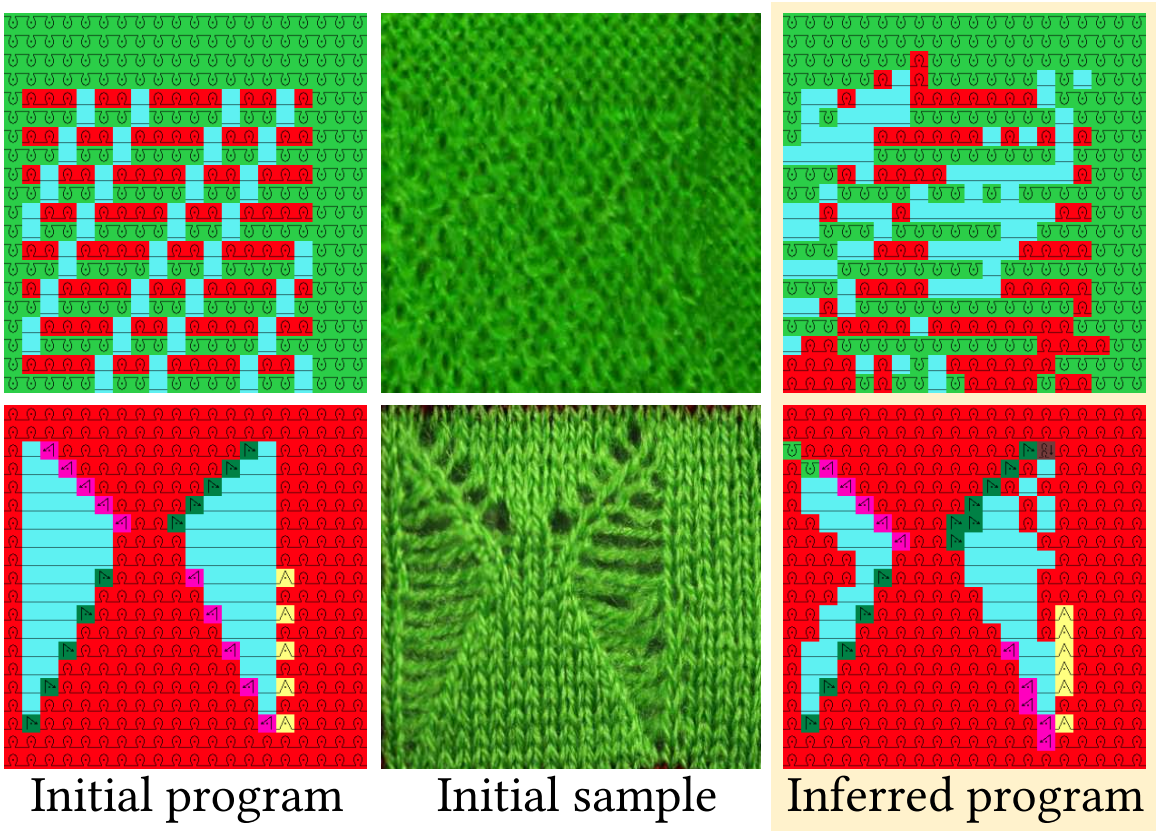


Figure 4-13: The two main test samples that are definitely not knittable as-is.

4.7 Discussions and Related Work

4.7.1 Pattern Scale Identification

Our base system assumes that the input image is taken at a specific zoom level designed for our dataset. This is not true for a random image. We currently assume this to be solved by the user given proper visual feedback (i.e., the user would see the pattern in real-time as they scan their pattern of interest with a mobile phone).

We investigated the potential of automatically discovering the scale of the pattern. Our base idea is to evaluate the confidence of the output instruction map for different candidate scales and to choose the one with highest confidence. Although the softmax output cannot directly be considered as a valid probability distribution, it can serve as an approximation, which can be calibrated for [62]. As a proof of concept, we take a full 5×5 pattern image from our dataset and crop its center at different scales

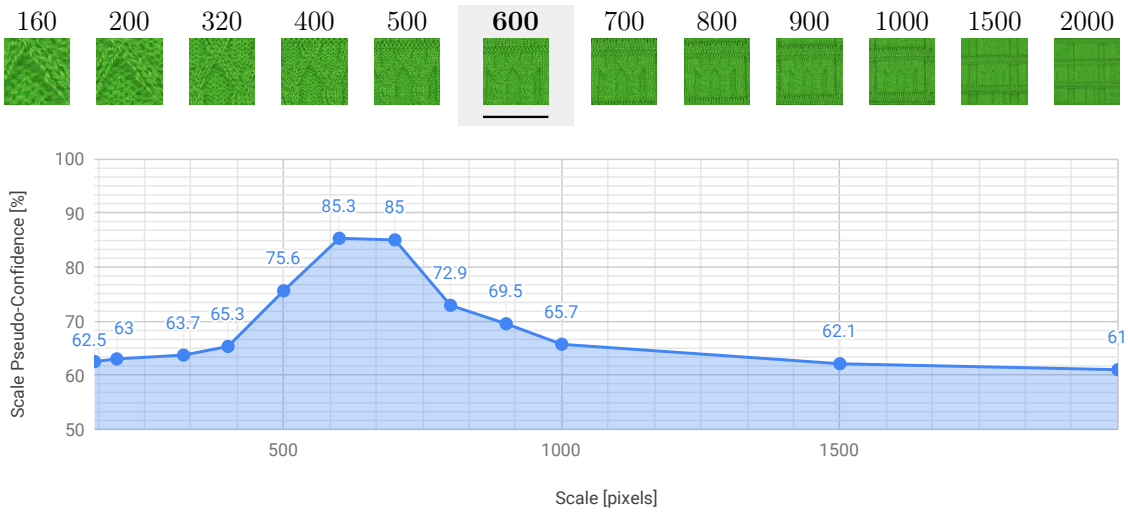


Figure 4-14: Scale identification experiment. *Top row*: cropped input image at corresponding scales with the correct pixel scale in bold with a light-gray background. *Plot*: pseudo-confidence curve showing a peak at the correct pixel scale (600).

from 160 pixels to 2000 pixels of width. We then measure the output of the network and compute a *scale pseudo-confidence* as the average over pixels of the maximum softmax component.

In Figure 4-14, we show a sample image with crops at various scales, together with the corresponding uncalibrated pseudo-confidence measure, which peaks at around 600 pixels scale. Coincidentally, this corresponds to the scale of our ground truth crops for that image.

This suggests two potential scenarios: (1) the user takes a much larger image and then that pattern image gets analysed offline to figure out the correct scale to work at using a similar procedure, and then generates a full output by using a tiling of crops at the detected scale, or (2) an interactive system could provide scale information and suggest the user to get closer to (or farther from) the target depending on the confidence gradient.

4.7.2 Learning with Simulated Data

The presented learning framework demonstrates a way to effectively leverage both simulated and real knitting data. There have been a recent surge of adversarial learn-

ing based domain adaptation methods [70, 155, 167] in the simulation-based learning paradigm. They deploy GANs and refiners to refine the synthetic or simulated data to look real. We instead take the opposite direction to exploit the simple and regular domain properties of synthetic data. Also, while they require multi-step training, our networks are end-to-end trained from scratch and only need a one-sided mapping rather than a two-sided cyclic mapping [70].

4.7.3 Semantic Segmentation

Our problem is to transform photographs of knit structures into their corresponding instruction maps. This resembles semantic segmentation which is a per-pixel multi-class classification problem except that the spatial extent of individual instruction interactions is much larger when looked at from the original image domain. From a program synthesis perspective, we have access to a set of constraints on valid instruction interactions (e.g. *Stack* is always paired with a *Move* instruction reaching it). This conditional dependency is referred to as context in semantic segmentation, and there have been many efforts to explicitly tackle this by Conditional Random Field (CRF) [33, 140, 192]. They clean up spurious predictions of a weak classifier by favoring same-label assignments to neighboring pixels, e.g., Potts model. For our problem, we tried a first-order syntax compatibility loss, but there was no noticeable improvement. However we note that Yu and Koltun [189] observed that a CNN with a large receptive field but without CRF can outperform or compare similarly to its counterpart with CRF for subsequent structured guidance [33, 192]. While we did not consider any CRF post processing in this work, sophisticated modeling of the knittability would be worth exploring as a future direction.

Another apparent difference between knitting and semantic segmentation is that semantic segmentation is an easy – although tedious – task for humans, whereas parsing knitting instructions requires vast expertise or reverse engineering.

Finally, we tried to use a state-of-the-art scene parsing network with very large capacity and pretraining [193] which led to similar results to our best performing setup, but with a significantly more complicated model and training time.

4.7.4 Neural Program Synthesis

In terms of returning explicit interpretable programs, our work is closely related to program synthesis, which is a challenging, ongoing problem.[2] The recent advance of deep learning has made notable progress in this domain, e.g., Devlin et al. [47], Johnson et al. [83]. Our task would have potentials to extend the research boundary of this field, since it differs from any other prior task on program synthesis in that: 1) while program synthesis solutions adopt a sequence generation paradigm [89], our type of input-output pairs are 2D program maps, and 2) the domain specific language is newly developed and applicable to practical knitting.

[2] A similar concept is *program induction*, in which the model learns to mimic the program rather than explicitly return it. From our perspective, semantic segmentation is closer to program induction, while our task is program synthesis.

Chapter 5

Primitive-Based Garment Design

A common strategy for designing complex artifacts consists in composing simpler building blocks. This is the case of most CAD software for engineering [11, 94, 108, 137]. One typical property of such workflows is that the corresponding designs can be customized by modifying user-defined parameters that correspond to important semantic features. In this chapter, we ask the following question:

> “What are necessary and/or sufficient building blocks for creating knitting designs that can be customized?”

5.1 Knitting Templates

Existing commercial software [150, 159] provide garment templates with a set of customizable properties (sizing, shaping types, etc.) such as in Figure 5-1. Templates are an effective starting point for creating a design as they wrap expert knowledge and allow for simple parameter-based customization.

Contents of this chapter are adapted with permission from - A. Kaspar, L. Makatura, W. Matusik, “Knitting Skeletons: Computer-Aided Design Tool for Shaping and Patterning of Knitted Garments”, UIST 2019. https://doi.org/10.1145/3332165.3347879. [Copyright by the authors].

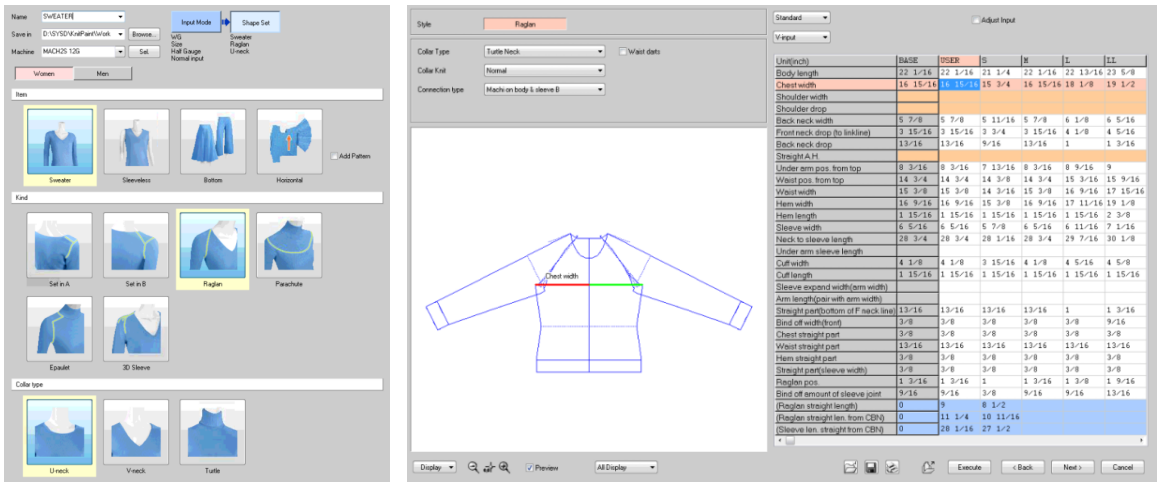


Figure 5-1: Examples of knitting template dialogs for a sweater in KnitPaint [150]: the categories of sweaters (left), the sizing information (right).

5.1.1 Limitations of Existing Templates

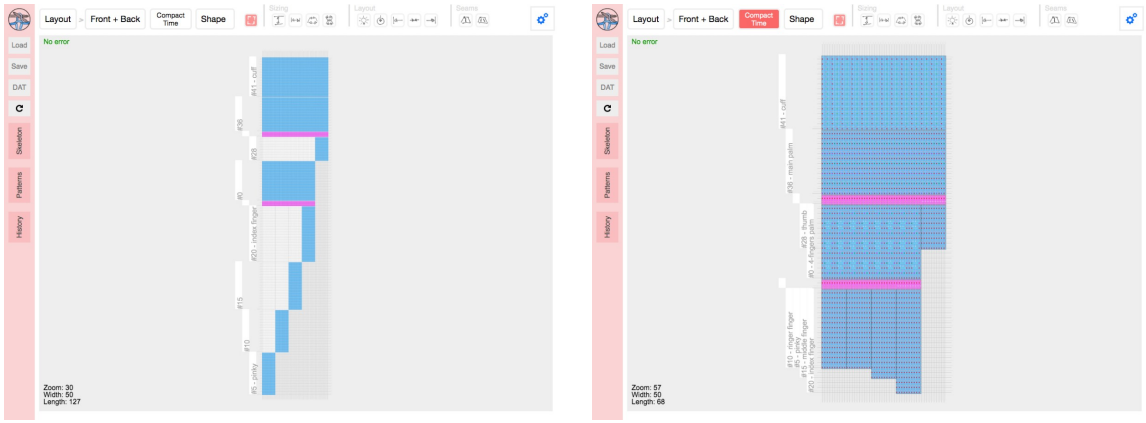
Unfortunately, the set of templates is obviously limited and template parameters only modify the garment geometry (known as *shaping*). The surface texture and appearance (including *patterning* and *colorwork*) must be specified through separate tools, or by manipulating the local instructions generated from the templates.

The two main limitations we wish to alleviate are: (1) existing templates cannot be composed, which limits potential to a fixed set of predefined shapes, and (2) these templates lack a bidirectional decoupling between shaping and patterning, such that any alteration of shape parameters requires recreating the associated patterns.

5.1.2 Existing Primitives for Knitting

McCann et al. [116] propose a tool to compose knitting primitives on the time–needle bed and introduce a binding strategy based on a novel algorithm for automating the scheduling of loop transfers.

Our work differs by (1) relying on a *split/merge* primitive to form branching structures instead of relying on manual binding, (2) not requiring the user to specify the needle bed layout, but instead inferring it automatically given fully parametric primitives, and (3) supporting stitch patterns on top of the shaping primitives.



(a) Full time-needle bed layout (b) Compacted layout for local editing

Figure 5-2: The *time–needle bed* depicts the knitting process over time. We provide a *compact* version that collapses suspended stitches to allow a local composition of primitives instead of the traditional composition over time. Both sides can be inspected separately or together.

5.1.3 Proposed Workflow

We present here a typical workflow session, and then elaborate on several individual components and features of our system. The following sections detail the shaping primitives (Section 5.2), patterning DSL (Section 5.3) and system implementation (Section 5.4), before discussing our results (Section 5.5) and the feedback from non-expert user experience (Section 5.6).

Typical Workflow

Our user starts from a base shape primitive (flat or tubular sheet) and modifies its shape parameters (e.g. size, layout, seams) by interactively manipulating them on the time-needle bed (Figures 5-2 and 5-3, detailed below). These interactions include dragging primitive boundaries for sizing, as well as dragging layout elements to change their location.

The user can also use a contextual menu to directly edit all exposed properties. In the global context (no shape selected), users can create new shapes and define user parameters (such as a glove's base finger length) that can refer to each other to create parametric design dependencies for templates. While hovering over a shape

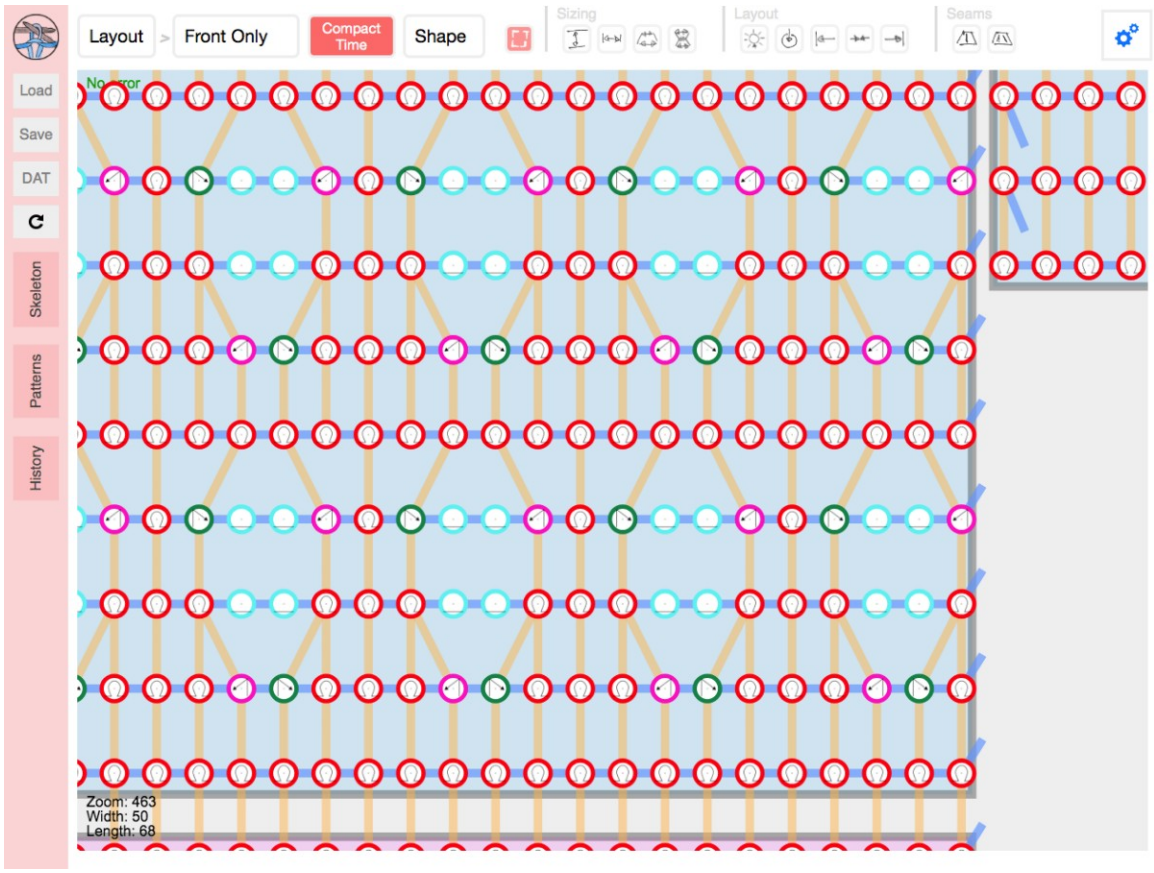


Figure 5-3: By zooming on the layout, we can inspect the local patterning operations and the simulated pattern flow.

primitive, users can rename it, delete it, or edit any of its properties. By clicking on a shape boundary (called an *interface*), the user can "extend" the given shape by connecting it to an available interface with valid matching parameters, or by creating a new primitive (which will automatically connect to the selected interface, and select matching parameters).

After creating the desired shape, the user switches to the pattern editing mode, where the toolbar actions affect individual stitches. In this mode, the user can either (1) draw the desired pattern directly onto a shape primitive, similarly to a pixel editing program, or (2) write pattern programs using our DSL within an editor that interactively previews the patterns on the time–needle bed. As the user zooms in/out, we display varying levels of information, including the type of pattern operation, and the local yarn topology (course and wale connections).

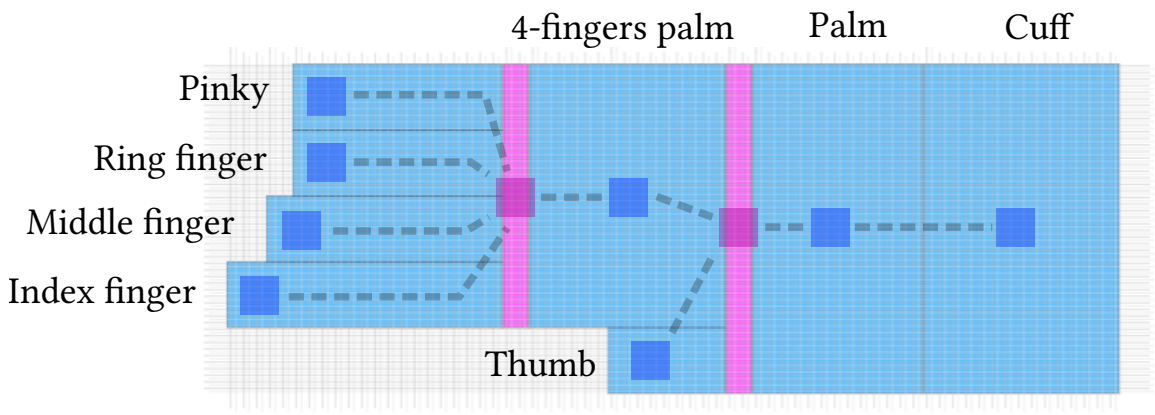


Figure 5-4: Sideways view of a compact glove in our system. The underlying skeleton graph is highlighted on top, with *tubular sheet* nodes in blue and *split* nodes in fuchsia.

Finally, the user can visualize the yarn structure with a force layout tool, save the resulting skeleton, load a new one, or inspect and export the necessary machine code.

Shape Skeleton

The recent work of Narayanan et al. [122] showed that any shape whose skeleton can be drawn on a plane without self-intersection can be knitted with a V-bed weft knitting machine. This motivates our underlying shape representation, which is a skeleton graph whose nodes are shape primitives, and edges are connections between node interfaces, as illustrated in Figure 5-4. The garment shape is defined by the node types, connections and parameters. The final surface pattern is defined by pattern layers associated with each node, and applied on the stitches locally. Together, these produce the final knitted structure.

By construction, our shape primitives allow for a continuous yarn path within each node and across interfaces, thus ensuring knittable skeletons. However, issues can still arise since (1) shape parameters across interfaces may be in conflict (e.g. different widths), and (2) user patterns may produce unsound structures or put excessive stress on the yarn. We identify such problems, but we do not fix them, because there is typically no "correct" solution without knowing the user's intention. Instead, we issue warnings (detailed later), and let the resolution to the user.

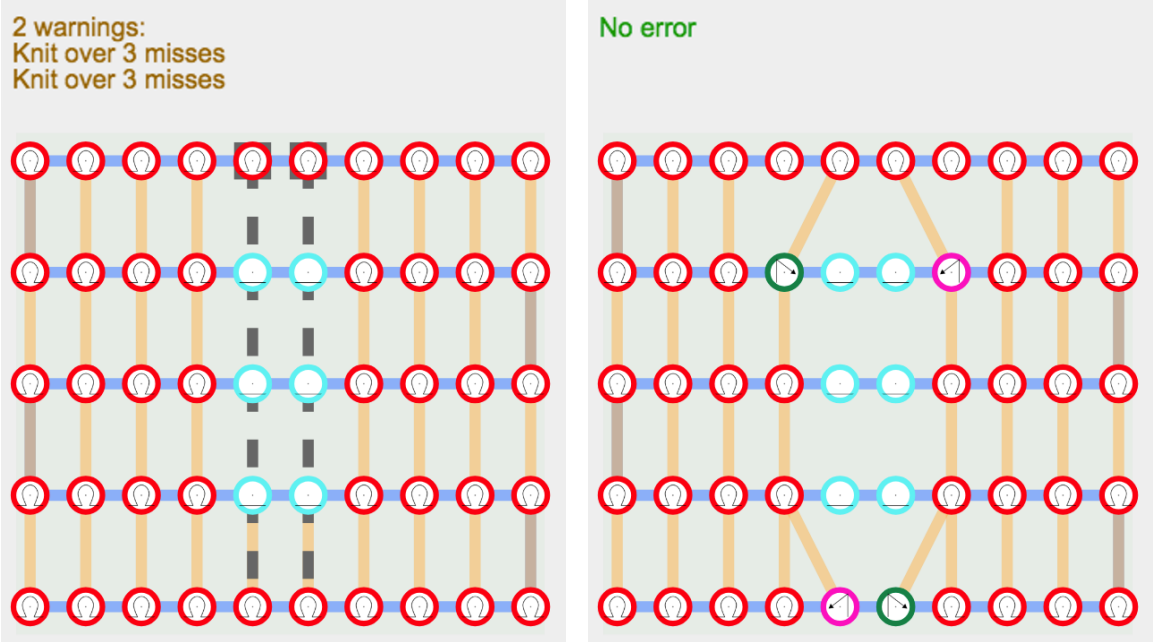


Figure 5-5: Warnings regarding a long-term dependency that would collapse the yarn (left). By highlighting the conflict dependencies, the user can more easily fix the pattern (right).

Time–Needle Bed

The main visualization onto which the shape skeleton is composed is the *time–needle bed*. It is a common representation for machine knitting [116, 150], illustrated in Figure 5-2. The actual bed layout is automatically computed as the user extends or modifies the underlying skeleton. This representation has two advantages: (1) it directly shows the time process followed by the knitting machine, which allows us to produce interpretable warnings if the user creates undesirable knitting structures (see Figure 5-5), and (2) it introduces a grid regularity, which allows the user to draw complex patterns in a manner similar to layered image editing.

Yarn Interpretation and Simulation

Our system interprets the yarn path through time to provide warnings and errors to the user as they create their shape and combine patterns, as shown in Figure 5-5. The main issues we catch are (1) unsafe yarn tension prone to yarn breakage, (2) too many yarn loops on a needle, risking pile-up or failed operation, and (3) reverse stitches

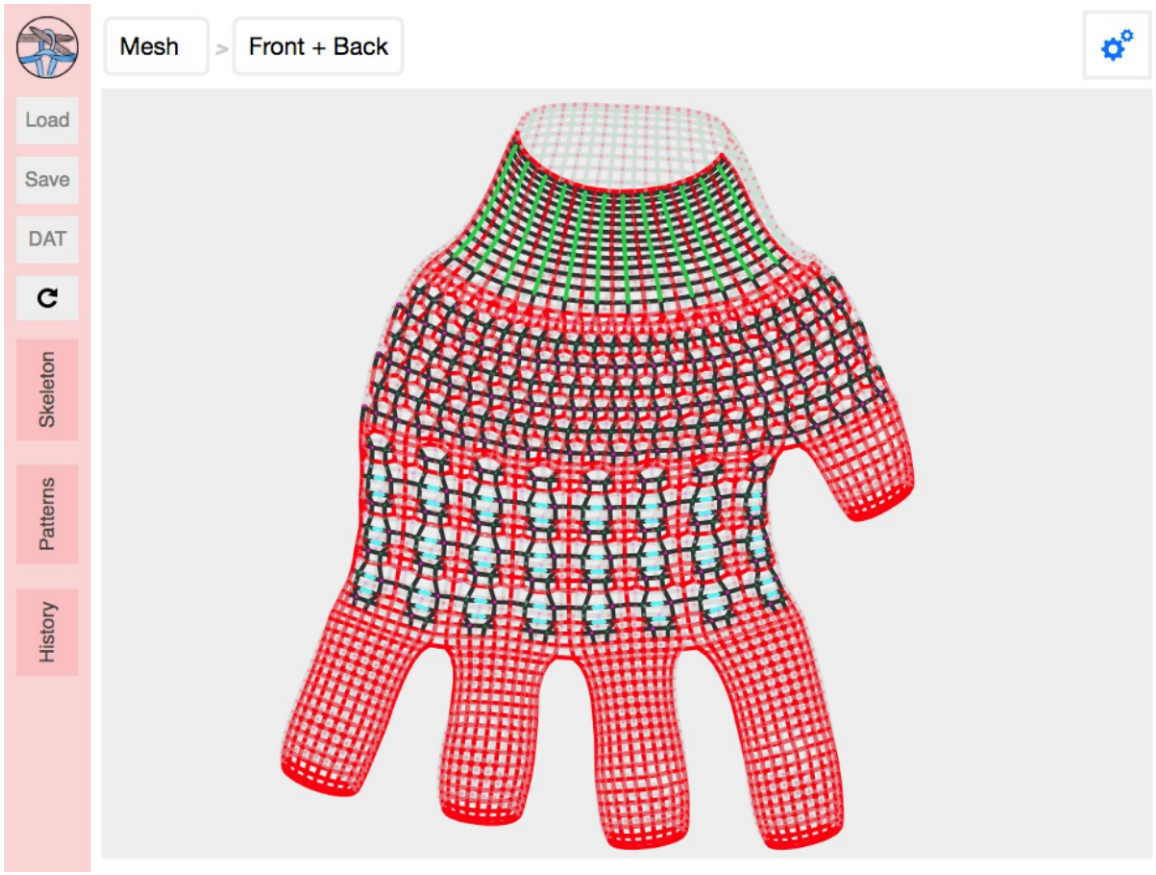


Figure 5-6: Force-layout simulation to preview the impact of the yarn stress forces on the final shape.

when the opposite bed is occupied (e.g. in full-gauge tubular knitting). We provide feedback both textually with the types of issue and potential fixes, and visually by highlighting the problematic stitches together with their conflict dependencies.

We also provide a force-layout graph simulation [172] as an approximate preview of the yarn deformation after knitting, as illustrated for a glove in Figure 5-6.

Low-Level Machine Code

Finally, we provide a view to inspect the low-level code that is generated for the current layout, as illustrated in Figure 5-7. This allows experienced designers and machine operators to inspect the actual program used by the machine.

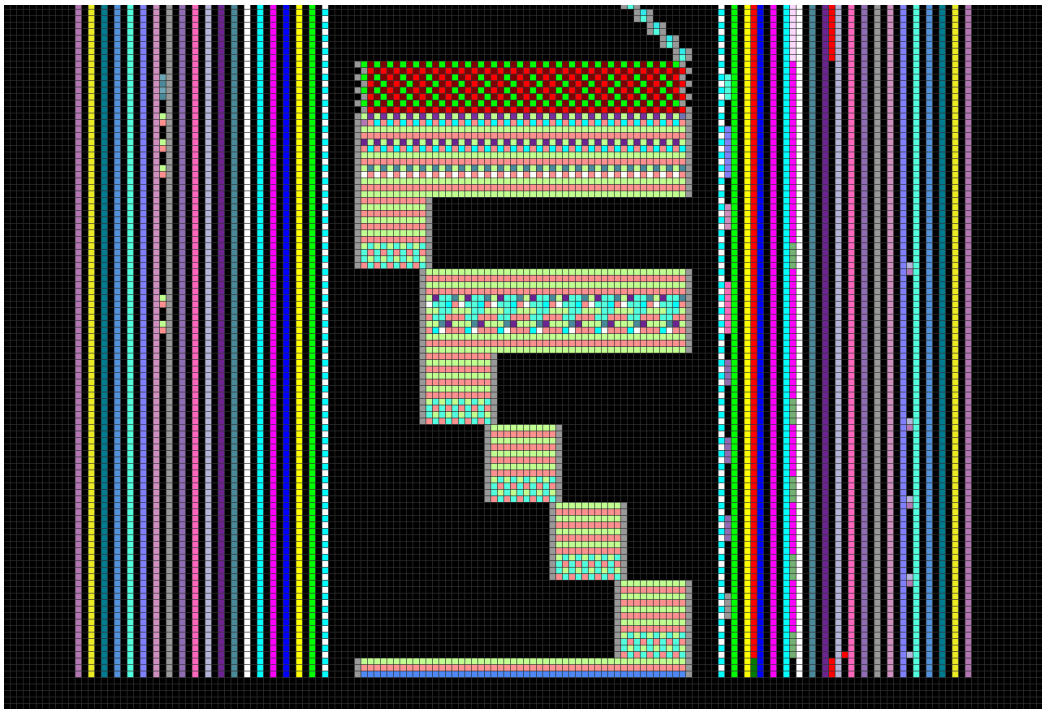


Figure 5-7: Part of the low-level instructions for a simplified version of the glove, to be processed with KnitPaint [150].

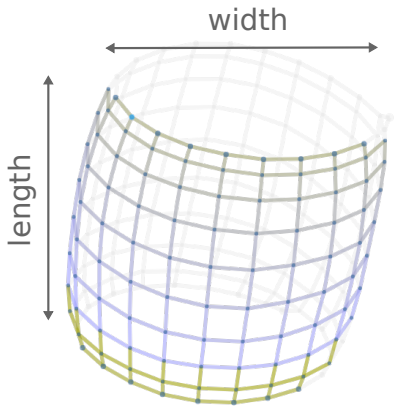
5.2 Parametric Shape Primitives

Each of our three knitting primitives (*Sheet*, *Joint*, and *Split*) play a specific role in the garment's final shape. We detail each primitive and its properties, then provide more details on our skeleton editing paradigm.

As skeleton nodes, all primitives have a `name` (for visualization and pattern references), a `pattern`, and a `gauge`, which we detail in the next section. All nodes also define a set of *interfaces* which can be connected to other nodes.

5.2.1 Sheet / Tube

The *Sheet* primitive is the base component for knitting any flat or tubular structure. Its two main parameters are its `length`, defining the number of courses making up the sheet, and its `width`, defined over the length. At a high-level, the user can modulate the width as a piecewise linear function over the normalized length interval $[0; 1]$, yielding non-rectangular profiles illustrated in Figure 5-9. If desired, the user can



Property	Values
Type	`Flat`, `Tubular`
Length	Integer
Width	$[0;1] \to \mathbb{R}_{>0}$
Shaping	`Uniform, Sides, Left, Right, Center, Custom`
Alignment	`Left, Right, Center`
Bottom	Interface
Top	Interface

Figure 5-8: A tubular sheet, and the table of its properties

also customize the stitch-level shaping behavior (i.e. placement of stitch increases and decreases) using one of multiple predefined behaviours, or a user-provided function that specifies how to allocate wale connections when changing the course width. We provide details for these functions in Section 5.2.5, including examples of how the shaping behavior affects the appearance of the yarn with the location of seams. The primitive layout can be customized by choosing a specific `alignment`, which impacts both the bed layout and the yarn stress distribution. This primitive has two interfaces: the *top* and the *bottom*.

5.2.2 Joint

Our *Joint* primitive captures the second shaping process, called *short rows*, which only knit across a subsection of the current course, while suspending the other stitches. These partial rows induce bending in the structure, as in a sock heel. A *Joint* represents a collection of such short rows. The user can specify the number of short `rows`, the `width` of each, and their respective `alignment`. Users can also specify a `layout`, which controls the normalized location of short rows along the interface boundaries, i.e. their offset for flat knitting, or the rotation for tubular knitting. The interfaces are the same as for the *Sheet* primitive.

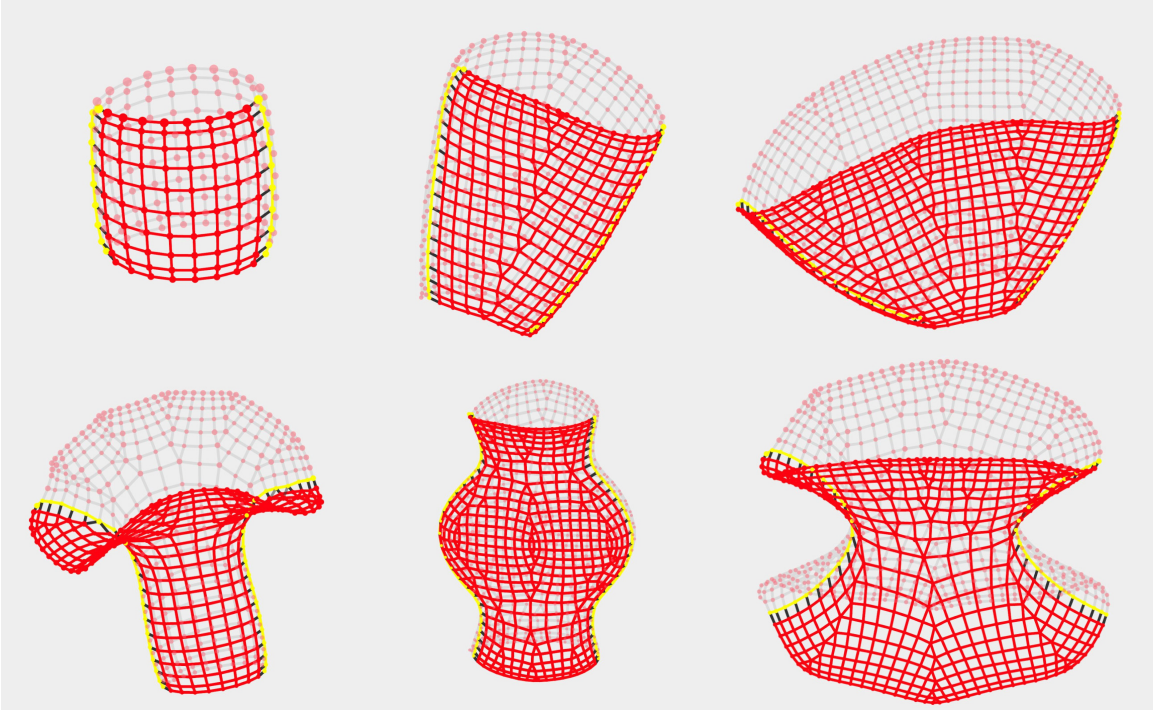
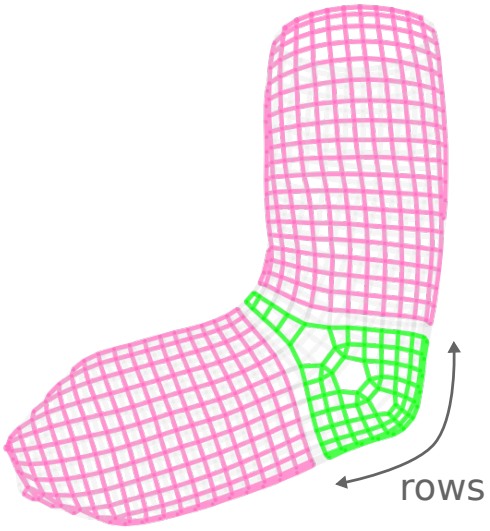


Figure 5-9: Different variations of a tubular sheet's width function. The yellow stitch nodes highlight the boundaries between front and back on the time–needle bed layout.

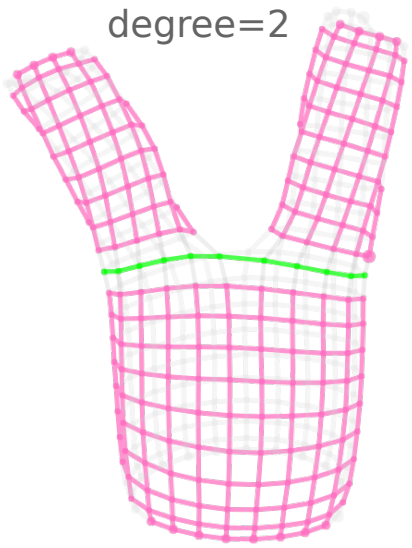


Property	**Values**
Rows	Integer
Width	$[0;1] \to \mathbb{R}_{>0}$
Layout	$[0;1]$ or "`auto`"
Alignment	`Left, Right, Center`
Bottom	Interface
Top	Interface

Figure 5-10: Joint primitive as the heel of a sock, and the table of its properties

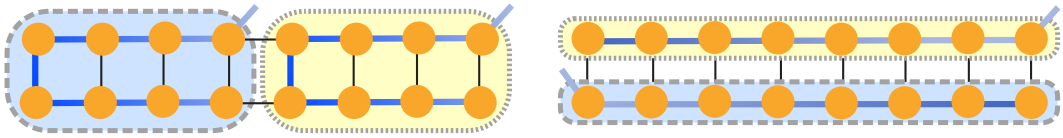
5.2.3 Split / Merge

Finally, our *Split* primitive allows for more complicated structures (like gloves) that require topological branching and/or merging. It merely consists of a *base* interface



Property	Values
Degree	Integer
Layout	$[0;1]^d$ or "auto"
Alignment	`Uniform, Left, Right, Center`
Folded	`True` or `False`
Base	Interface
Branches	List[Interface]

Figure 5-11: Split primitive between one sheet branching into two, and the table of its properties



(a) Tubular branches
Folded = `True`

(b) Flat branches
Folded = `False`

Figure 5-12: Diagram illustrating the difference between *folded* and *non-folded* splits for a tubular base across the two needle beds. The two branches are highlighted with different colors.

and a set of *branch* interfaces. It has a branching *degree* together with a branch *layout*. For automatic layouts, the user can also provide the branch `alignment`. Furthermore, for tubular bases, the branching can be *folded* (tubular branches) or not (flat branches) as illustrated in Figure 5-12. Flat bases only allow flat branches. Finally, since the interface connections are not restricted in any direction, this primitive can be used both to subdivide an interface or to merge multiple interfaces together.

5.2.4 Editing Primitive Parameters

Our system allows multiple interaction strategies. One can work exclusively with the abstract skeleton graph, and edit parameters using a tabular inspection panel.

Alternatively, one can drag the mouse to interactively extend the shapes on the bed layout. In this approach, more complicated parameters can be specified using the contextual menu that allows the same fine-grained control as the tabular parameter panel.

When specifying parameters through the input panel or the contextual menu, the user can enter either direct numbers, or `#expressions` that introduce global *design parameters*. These are global variables that can be reused across different inputs and node parameters, providing a means to expose important design parameters (such as a glove's width or length scale). For example, the user could specify the length of a glove finger via `(#Len + #LenDelta)` which introduces two parameters, `#Len` and `#LenDelta`. Each of these could be independently applied for the other finger specifications, and any changes to the variable value would be reflected globally. These expressions can also refer to node properties with `@prop`. For example, the width of a sheet could be made equal to its length using the expression `@length`.

5.2.5 Programmatic Shaping

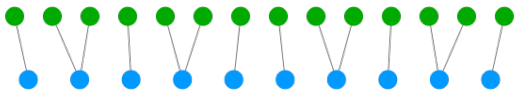
Our system allows the user to specify shaping programs for the *Sheet*/*Tube* primitives. Although the main change in shape from such primitive comes from the difference in course sizes, the local wale connections have two important impacts:

- They induce visual seams around the change of wale flow
- They impact the stitch stability (and thus can lead to knitting failures)

A shaping program describes a function which links stitch units between two consecutive course rows of sizes M and N. Since the order is not important as we are creating bidirectional wale connections, we assume that $M \leq N$. The program takes as input M, N, as well as the current index i within the first course ($0 \leq i \leq M$), and potentially other context parameters. Its output is a (possibly empty) sequence of mappings between cells of the first course and cells of the second course. For two courses of same sizes, the simplest mapping is $i \rightarrow i$. Two common shaping programs

(*uniform* and *center*) are illustrated in Figure 5-13. Given the mapping from the shaping program, our system then creates corresponding *wale connections* between the associated stitch units.

In comparison, systems that build upon meshes such as AutoKnit [122] or Stitch Meshes [186, 187, 190] do not directly provide user control over increases and decreases. Instead, these are implicitly encoded in the original mesh. Figure 5-13 shows that we can use shaping programs to control the location of seams, which is of interest when customizing a garment.



```
let ratio = M / N;
let j1 = round(i * ratio);
let j2 = round((i+1) * ratio);
if(j2 == j1 + 1){
  // i maps to single j1
  i -> j1;
} else {
  // i splits into two
  // at each sector boundary
  i -> (j1, j1 + 1);
}
```

Listing (5.1) Uniform shaper program

```
let d = N - M;
let o1 = round(M/2 - d/2);
let o2 = o1 + d;
if(i < o1)
  i -> i; // left part
else if(i < o2){
  // increase splits in center
  i -> (o1 + 2*(i-o1),
        o1 + 2*(i-o1) + 1);
} else
  i -> (i+d); // right part
```

Listing (5.2) Center shaper program

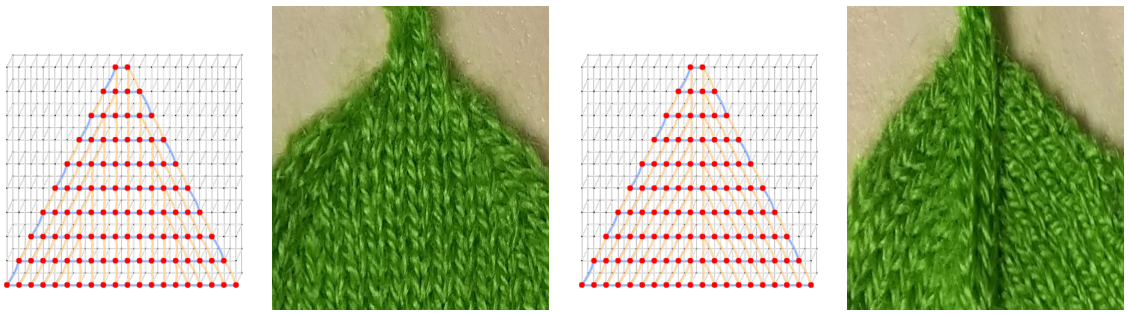


Figure 5-13: Standard shaper programs: (left) *uniform* distributes the increases and decreases uniformly, (right) *center* accumulates them in the center of the course. Notice the visible seam in the center. At the top are program illustrations for $M = 10$, $N = 14$ (these numbers are here for illustration, they vary for every row in a real knitting structure). At the bottom are the bed layouts as well as the knitted results for a flat triangular sheet.

Category	Methods	Explanation
Filtering	`all(), filter(pred)`	Stitches that match a logic predicate
Sets	`or(x,y), and(x,y), minus(x,y), inverse()`	Standard set operations on collections of stitches
Indexed	`wales(rng), courses(rng), select(c, w)`	Indexed stitches within a range
Neighborhood	`neighbors(rng), boundaries()`	Stitches at some distance from the selection
Named	`named(), shape(name), itf(name)`	Stitches of a named entity
Masking	`stretch(grid), tile(grid), img(src)`	Stitches that match a given grid mask

Table 5.1: Our categories of pattern queries with their main methods and usage explanation

5.3 Patterning

Given a shape skeleton, our system assembles stitches for each of its nodes, and then merges stitch information at the interfaces, producing a stitch graph whose nodes are individual stitch units. Initially, the stitch connections (course and wale) are defined by the shape primitives. Each stitch also includes a pattern *operation* that describes how to modify the stitch with respect to its surrounding neighborhood. These operations allow the user to design special surface textures and appearance on top of the shape.

5.3.1 Pattern Operations

For the pattern operations of each stitch, we use the instruction set described in Section 4.2 [91]. The main difference is that we apply those instructions not on a one-sided regular grid, but on the stitch graph, which is then projected back to the time-needle bed to generate the final low-level machine code. Figures 4-2 and 4-3 illustrate the types of pattern operations we support.

Importantly, our pattern operations do not create or delete stitch units. Instead, they modify how individual units are interpreted, either by providing a different

operation to the target needle (*purl* or *tuck* instead of the default *knit*), or by altering the wale connections (*miss*, *move* and *cross*). In the case of *miss*, any previous wale connections of the missed stitch are transferred to the subsequent one. *Move* and *cross* operations change the subsequent wale connection to a neighboring one along the next course.

If a neighboring wale target does not exist (e.g. at the border of a flat sheet), then the operation is not applied. Note that courses of tubular sheets are treated as cyclic, so a neighboring wale connection always exists (possibly on the other bed). The system ignores *move* and *cross* operations on irregular stitches (increase/decrease) to ensure structural soundness.

5.3.2 Patterning DSL

To apply pattern operations on the stitch graph, we designed a *domain specific language* in which the user first specifies a subset of stitches of interest – the **query** – onto which they can **apply** a given patterning operation.

Our types of *queries* are listed in Table 5.1, and a subset is illustrated in Figure 5-14. The main query is `filter`, on which all other queries are based (with some specialized implementations to improve speed).

5.3.3 Drawing Layers

All our patterns are synthesized using our DSL. We split the pattern specification into a sequence of layers: (1) an initial global layer spanning all stitches, (2) varying sequences of per-node layers modifying stitches of specific nodes, and (3) a final global layer. By default, all base layers are empty and we assume the base pattern is a standard knit stitch.

Users can write their own program, use pre-existing ones, or interactively draw node pattern layers in a manner similar to image editing software that allows pixel-level modifications. We preview the impact of the patterns on the wales, as illustrated in Figures 5-3 and 5-5, where *move* operations displace the upper wale targets.

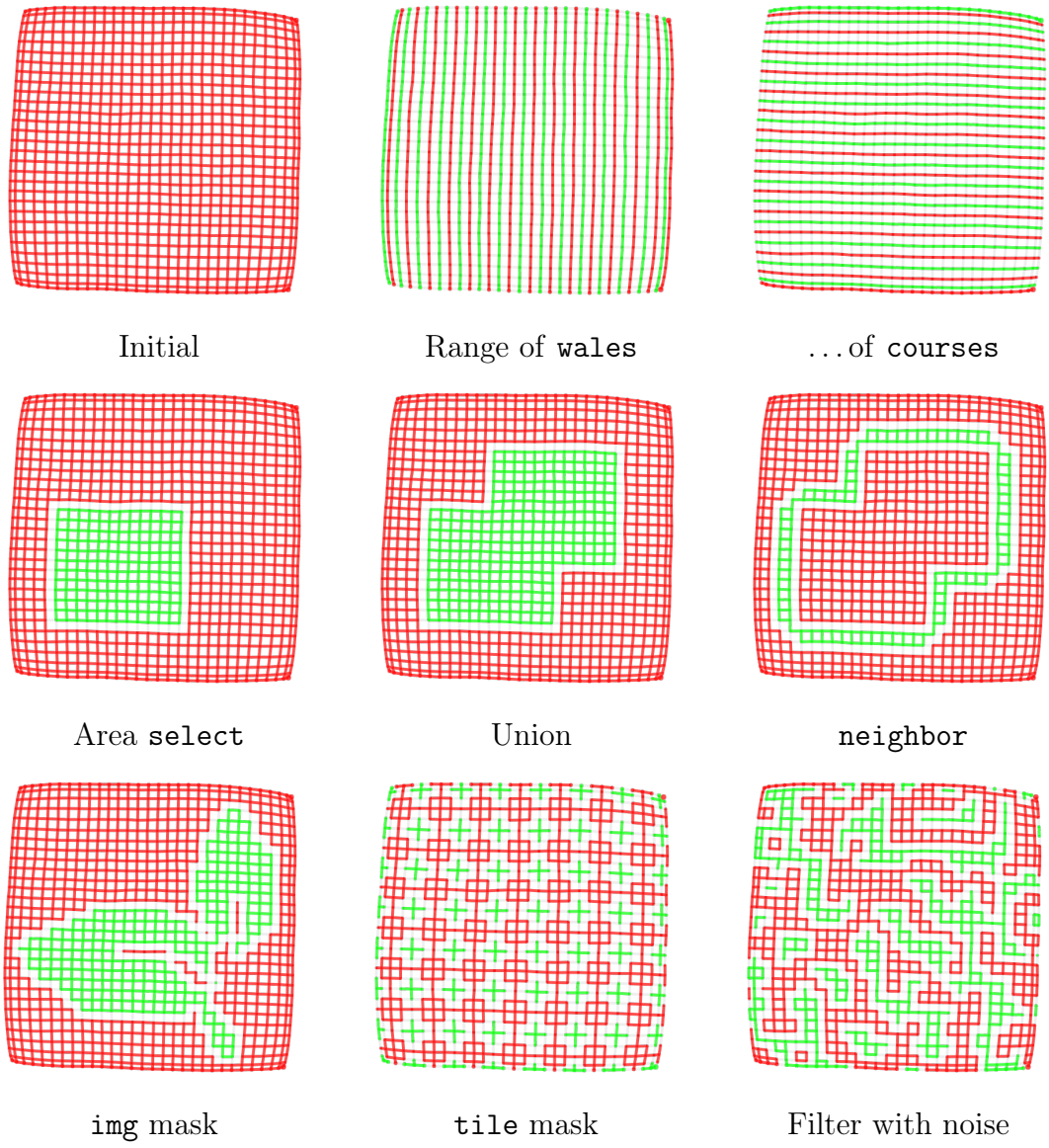
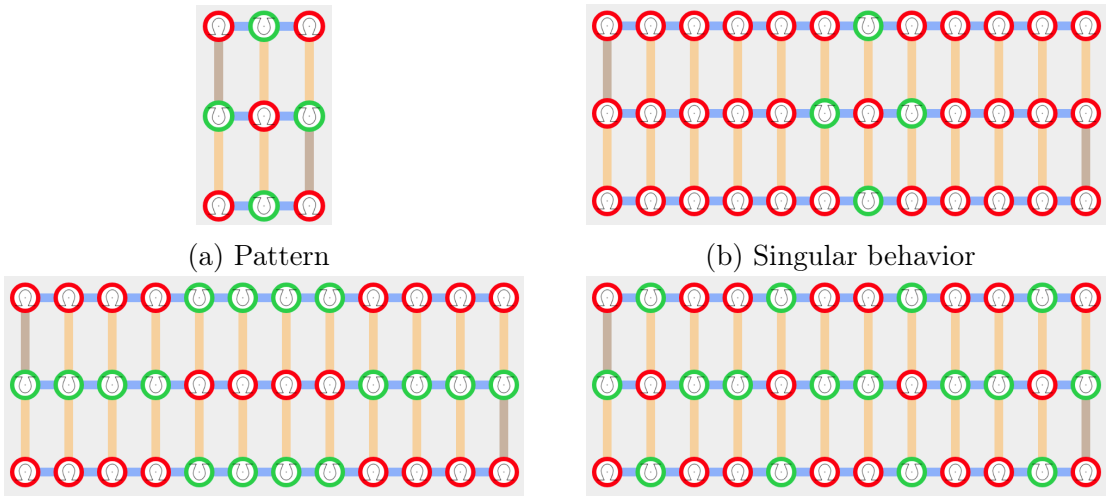


Figure 5-14: Illustrations of some of the main pattern queries, each highlighted on a 30×30 flat sheet.

Our pattern layers can be exported, imported and resampled for different shapes and sizes. The resampling behaviour can be specified by using different types of layers. We provide three pattern drawing types – **singular**, **scalable** and **tileable** – whose behaviors are illustrated in Figure 5-15.



(a) Pattern (b) Singular behavior

(c) Scalable behavior (d) Tileable behavior

Figure 5-15: A base 3×3 pattern and illustrations of the different resampling behaviors for each of our layer types.

Singular Layers

This is our default drawing mode, which does not resample the initial pattern, but simply modifies its location to account for the change in size (e.g., by centering the original pattern).

Scalable Layers

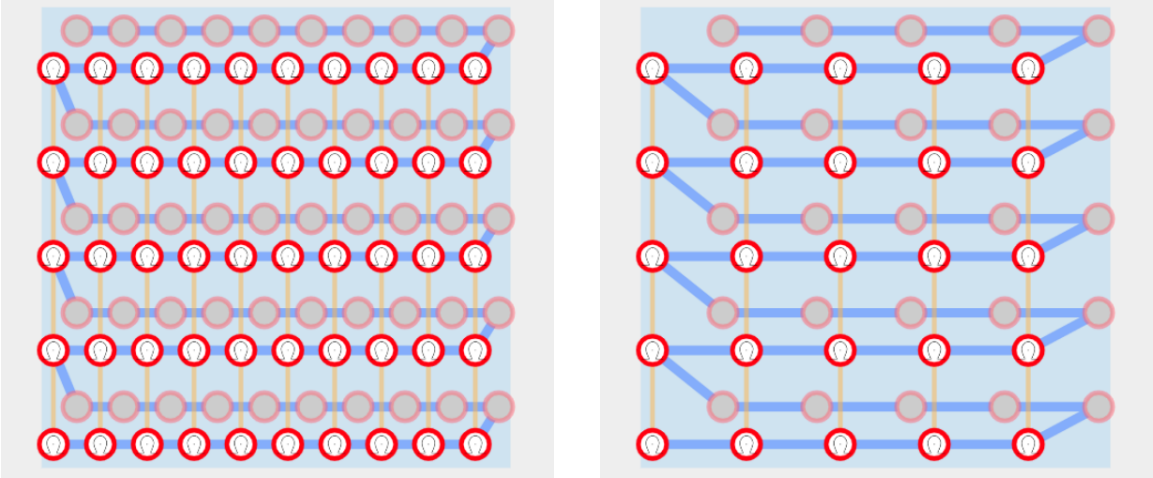
These layers resample their pattern by nearest neighbor resizing. In this mode, we do not allow *cross* operations, which are coupled in paired groups and typically applied with a limited local range constraint to prevent yarn breakage.

Tileable Layers

These layers resample their pattern by applying a modulo operation so as to create a tiling of the original pattern.

From Drawings to Programs

Drawings are stored as semi-regular grids of operations, which can be empty (for no operation, the default). To apply the drawing, we transform it into a basic program



(a) Tubular in full gauge. width = 10 (b) Tubular in half gauge. width = 5

Figure 5-16: Illustration of the gauge parameter. The `width` is modified to keep the same bed support. The half-gauge variant uses different offsets between beds to allow reverse stitches.

that makes use of the drawing data together with a resampling function depending on the type of layer. *Singular* layers relocate the drawing information, whereas *scalable* and *tileable* layers use the `stretch` and `tile` functions, respectively.

5.3.4 Half-Gauge Knitting

For each primitive, the user can choose a desired `gauge` (either *full* or *half*). This property is important because pattern operations that modify the side of the stitch operation (regular vs reverse, or "purl") can only occur on a given stitch if the needle directly across from it (on the opposite bed) is empty. In the case of tubular fabric, the opposite bed holds the other side, which can lead to conflicts. In such case, *half-gauge* knitting is a typical solution, which consists in knitting on every other needle, with both sides offset such that any needle's counterpart on the opposite bed is empty, as shown in Figure 5-16. Note that the need for half-gauge depends on both the shape *and the pattern*. Even if it is unnecessary, it may still be desirable because it creates looser fabric. Thus, we do not automatically choose which gauge to use, but let the decision to the user. For full-gauge primitives, we detect conflicting patterns and show a warning to suggest switching gauge.

5.4 Implementation Overview

We provide a brief overview of our pipeline implementation. It borrows ideas from both Stitch Meshes [186, 190] and Automatic 3D Machine Knitting [116, 122]. The initial input is the user-generated skeleton as well as a starting interface to knit from. The pipeline can be divided into the following stages:

1. **Stitch graph computation** – each node is translated into a set of stitch courses, the course traversal is scheduled and the yarn is traced to resolve missing connectivity;

2. **Pattern application**: pattern layers are translated into their corresponding programs and executed, effectively assigning a pattern instruction to each stitch;

3. **Layout optimization**: the node courses are organized into layout groups, whose offsets and bed sides are optimized iteratively to reduce the stress between stitches;

4. **Knitting interpretation**: the whole time–needle bed is generated, and the yarn path interpreted using both associated pattern instructions and wale connections to generate a sequence of passes (cast-on, actions, transfers, cast-off);

5. **Knitting simulation**: the bed passes are simulated to evaluate potential dangerous yarn situation (large moves, yarn piling up) and generate appropriate warning or error messages;

6. **Code generation**: the bed passes are translated into low-level machine instructions including specialized needle cast-on and cast-off procedures to prevent yarn from unravelling.

Additionally, for visualization purposes, we optionally compact the time–needle bed by removing some of the suspended stitch sections. This generates a more compact time–needle bed visualization in which nearby courses are located close-by in "time" – the time axis becoming loosely defined.

5.4.1 Stitch Graph Computation

The computation of the stitch graph is done in three steps: (1) per-node courses are generated and bound within their respective node, (2) courses are topologically sorted and scheduled for traversal, (3) the yarn path is traced and additional missing connectivity is resolved or created as needed.

Shape Generation Each skeleton node is individually transformed into a generic shape made of a sequence of stitch courses, some of which are annotated as interfaces (with respective names similar to that of the skeleton nodes). Each shape's course is assembled with its neighboring courses using the node's shaper program, layout and alignment properties.

Then all node's interfaces are processed: connected ones are bound across shapes, and disconnected ones are either left as-is or closed if chosen by the user.

At this stage, note that some stitch connections have not been generated. Notably, course connections across different courses require orientation information which depends on scheduling, happening next.

Course Scheduling Given the various shapes and their courses, we first group them into connected components, and then each group is separately scheduled, course-by-course. The scheduling is done by topologically sorting the courses according to knitting order constraints: we require previous branches to be knitted before allowing merge operations in *split* nodes.

Yarn Tracing Given the course schedule, we can now trace the path of the yarn and generate course connections. This also involves creating additional continuity stitches so that the yarn doesn't jump between far away stitches.

5.4.2 Patterning

At this stage, we have the final stitch graph, and we use our patterning layers, transforming them into programs that assign pattern instructions to each stitch.

5.4.3 Layout Optimization

From the course schedule, we generate individual disjoint needle bed assignments. The offsets and relative sides between courses of a same node are fixed to optimal assignments without taking other nodes into accounts, creating groups of fixed bed layouts.

Then, the bed layout groups are optimized w.r.t. each-other's offsets and relative sides. The main trick here is that we can easily pre-compute optimal alignments by approximating the yarn stress between two full beds by pre-computing their stitch pairs, as well as their corresponding center of mass, which should typically align for the minimal stress assignment.

5.4.4 Knitting Interpretation

The time-needle bed is generated by aggregating the layout groups, and a first pass interpret each bed time, bed after bed, generating consecutive per-bed passes over time.

This generates the following (potentially empty) passes for each time-step:

- **Cast-on**: empty needles are cast yarn onto (requires specialized procedures to ensure the yarn catches, which is different from typical already-cast knitting);
- **Actions**: per-needle actions are computed for each stitch of the current time's bed according to their number of up-going wales, their locations, the previous stitch actions, and their corresponding pattern instruction. Potential actions include: *Knit*, *Purl*, *Tuck*, *Miss*, *Knit Front/Back*, *Kickback Knit*, *Split Knit*;
- **Transfers**: necessary transfers are processed per-side at once, with relative orderings specified by the instruction types (e.g. Cross instructions, Stack instructions);
- **Cast-off**: stitches that should be cleared off the bed are cast-off using dedicated procedures to prevent unraveling.

5.4.5 Knitting Simulation

Given the interpretation, the actual knitting process is simulated to discover potential issues due to large yarn stretching or long interactions between stitch operations (e.g., knit-over-miss for large miss sections, which ends up collapsing the yarn, or several loops merging onto one, beyond the needle knitting capabilities). This produces potential warning and error messages that the user can use to correct their topology and node parameters.

5.4.6 Code Generation

The bed pass interpretations are translated into low-level machine instructions, which can then be processed by the machine (typically re-compiled there to check for further issues and assign machine parameters before knitting).

5.5 Results and Discussions

This section discusses achievable results from the perspective of an expert user, together with their limitations and potential improvements. We first consider the range of garments that are machine-knittable using our shaping primitives. We then showcase different pattern layer interactions as node parameters change. Finally, we cover the performance of the system, including interactivity limitations and necessary future work in terms of features.

5.5.1 Scope of Shaping Primitives

Our first collection of knitted results is illustrated on the 12 inch mannequin shown in Figure 5-17, together with adult-size versions of a patterned infinity scarf and a ribbed sock in Figure 5-18. We show the individual pieces in Figure 5-19, each knitted with a different yarn color. This includes:

- a *hat* using one cylindrical sheet with a narrow closed top and a wide open bottom;

Figure 5-17: Various garment prototypes on a 12 inch mannequin (left) and a glove with lace patterns (right).

- a *scarf* with pocket ends, using one main flat sheet and ends that are split-converted into cylinders (one end is open to let the other end through, and the other is closed as a pocket);

- a *yoked shirt* using one open tube split into three tubes (two for the sleeves, one for the main body);

- *sweatpants* as a waist tube with a split branching into two tubular structures;

- two *socks* using a joint for the heel and two tubes, one of which narrows down to the toes where it closes.

Figure 5-18: An infinity scarf with lace patterns (left) and a sock with ribs (right).

Limitations of the Shaping Primitives

By composing three types of primitives (*Sheet*, *Joint* and *Split*), our design space already spans many common garments including varieties of gloves, socks, scarves, headbands, hats, pants and sweatshirts. The garments which we do not handle nicely are traditionally sewed garments such as pullovers with *Raglan sleeves* or *drop shoulders*. The general challenge is specifying continuous interfaces that glue primitives over multiple courses.

One constraint of wholegarment industrial knitting machines is that they cannot instantaneously introduce a new garment section (i.e. tubular or flat sheets) with a perpendicular wale flow. Instead, perpendicular wale flows must be joined in a continuous gluing operation that connects the two suspended sections laterally, as illustrated in Figure 5-20.

The current primitives of this work do not support such continuous gluing op-

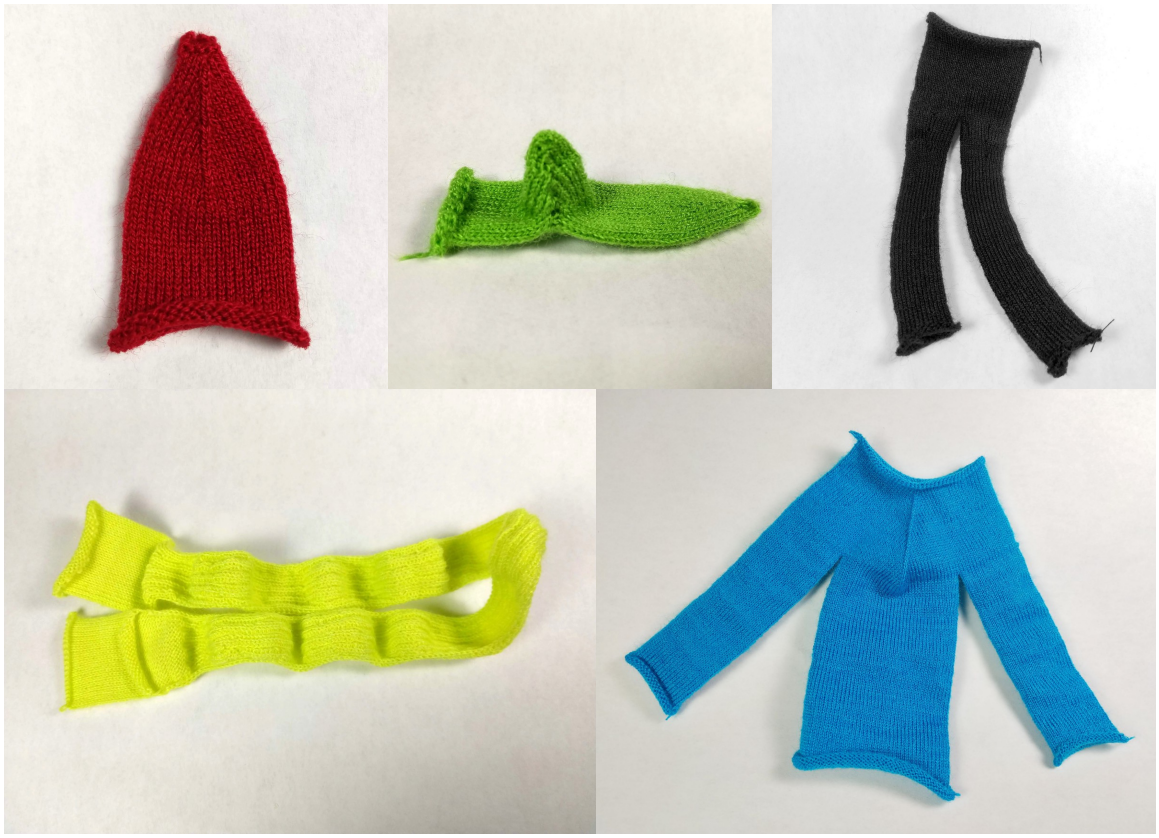


Figure 5-19: The individual garment pieces from the left of Figure 5-17. The scarf uses a single-sided part that would curl on itself by default. Thus we used simple 2 by 2 ribs to keep it flat.

eration. This notably means that we cannot knit garments with sleeves extending perpendicularly to the base trunk. Note that it is quite different from our *Joint* primitive whose flow is continuously changed from its `bottom` to its `top`, because it does not split the flow, but only redirect it in a single direction.

Possible solutions we envision include:

- Adding a specialized **T-Junction** primitive (although we are looking for a more general and simpler-to-specify primitive).
- Introducing **lateral interfaces** for *flat* sheet primitives, which – combined with *Joint*s and *Split* – would allow the creation of *T-Junction* structures (and more).
- Introducing an **Anchor** primitive that would allow specifying regions on top of current other primitives, to specify additional interfaces (e.g. hole sections,

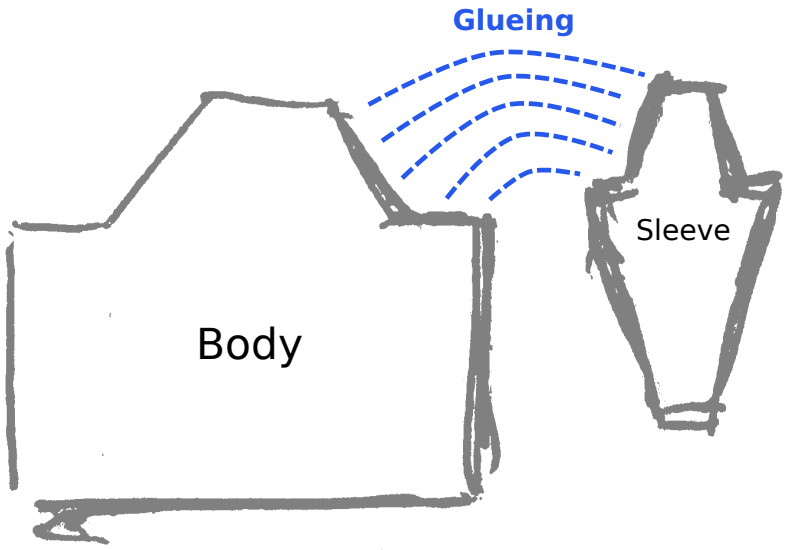


Figure 5-20: Illustration of one strategy to glue sleeves to the main body – here, a Raglan sleeve. The body and sleeves would both be knitted separately (i.e., next to each other, one at a time), and then they would be joined with a sequence of glueing operations joining both sides up to the neck section.

from which we could generate new lateral primitives).

Any of these unfortunately comes with many complication in terms of implementation and layout optimization because we cannot work with full courses alone and must allow binding courses to parts of other courses over time (i.e. for the lateral gluing operation). There is also the problem of how the user would parameterize the gluing process as many variants exist [27].

5.5.2 Pattern Layers in Action

Figure 5-22 illustrates how shape modifications alter two overlapping patterning layers. The background consists of a tileable layer repeating a sequence of left and right *moves* to create lacy holes, whereas the foreground is a scalable image mask applying normal knit stitches. As the sheet size changes, the layers behave differently: the background keeps tiling the same small move sequence, whereas the cat foreground expands with the size. Figure 5-23 illustrates an extension that includes a pure pro-

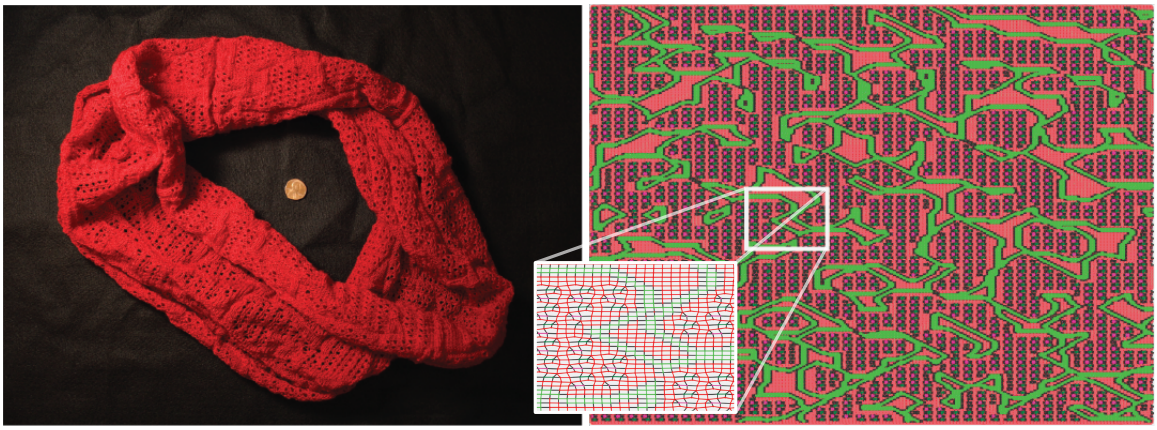


Figure 5-21: Visualization of the pattern of our infinity scarf (left) with our mesh visualization (right) and a close-up (center).

gram layer for the margin, and two complementary scalable foregrounds layers with different stitch operations (knit and purls).

In Figure 5-21, we visualize the mesh of a patterned infinity scarf, which uses a layer decomposition similar to the tiled lace. However, the tiled lace is applied within a program mask that makes use of *2D simplex noise* to create a random area selection. Its boundaries are mapped to regular stitches within 2 stitches, and to purls from 3 to 4 stitches away.

Then in Figure 5-24 we transfer the global hole pattern to two nodes of a glove skeleton with open fingers. This figure also illustrates the potential impact on shaping that some patterns have. Here, the cuff of the glove has a constant width that matches the palm node. The ribbed cuff shrinks considerably even though it is knitted over the same bed width (in half gauge). This is the same glove as in Figure 5-17.

Smarter Pattern Layers

Our pattern layers are not guaranteed to behave nicely when resampled or applied to a shaped primitive. A "correct" behaviour is often ill-defined: for example, at the top of a vertically striped hat, should the stripes become thinner or merge together? Thus, it is reasonable to defer to the user in such scenarios, as we do. A different, recently proposed strategy [71] would be to adapt *calibrated* patterns to the local context.

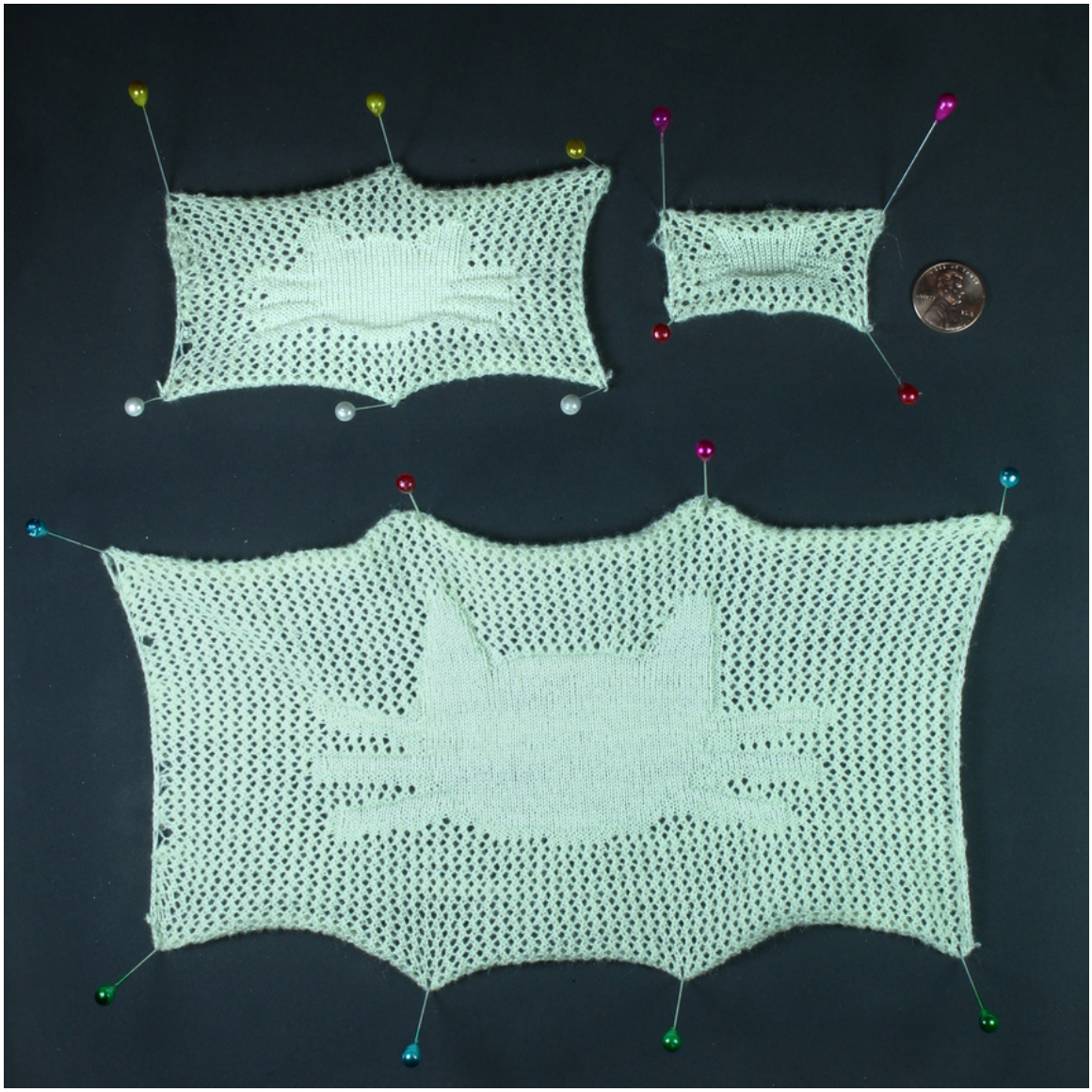


Figure 5-22: Impact of shape size on a two-layer pattern. The holes are tileable moves from Figure 4-3. The foreground cat is scalable and stretches with the shape.

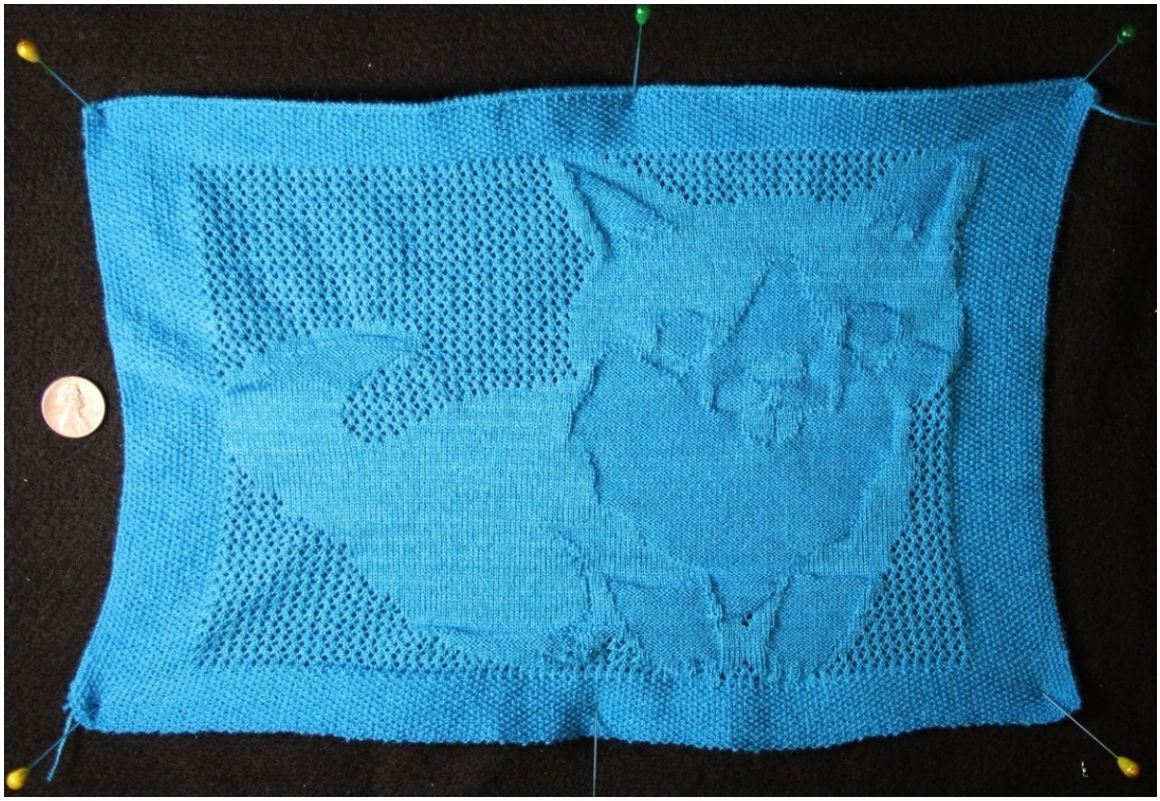


Figure 5-23: A four-layer pattern combining a tileable lace, a programmatic margin, and two scalable foregrounds for different shades of a Corgi.

Figure 5-24: Patterning the glove of Figure 5-17 from left to right: base shape, cuff in half gauge, half-gauge cuff with a rib pattern, and final glove with transferred hole pattern on main palm, as well as an additional pattern for the 4-fingers palm.

Table 5.2: Runtime performances of our system for the shapes within this chapter. All times are in milliseconds. The shape creation contains a *schedule* step but we ignore it as its runtime is negligible (i.e., ≤ 1). For the same reason, we ignore the *optimize* step of the layout computations.

⊢ Skeleton ⊣				⊢ Shape ⊣		*Pattern*	⊢ Layout ⊣		⊢ Yarn ⊣		
Name	***Nodes***	***Patterns***	***Stitches***	***Create***	***Trace***	***Develop***	***Create***	***Pack***	***Interpret***	***Simulate***	***Compact***
cat-32x32	1	2	1024	3	8	30	4	7	6	1	2
cat-64x64	1	2	4096	8	15	21	3	18	14	1	5
glove	10	3	5030	12	11	27	6	36	24	9	8
cat-128x128	1	2	16384	46	70	97	9	71	52	9	20
sock	6	1	17740	46	50	89	18	70	46	8	34
corgy	1	4	19200	22	78	129	13	83	60	11	24
beanie	3	1	28774	141	79	124	29	107	67	7	51
noisy-scarf	1	1	85000	199	328	1034	54	309	219	86	140

5.5.3 Performance

The running time for our system is highly dependent on the client machine and web browser being used. However, we still provide performance tables here to highlight the current processing bottlenecks: (1) the *stitch instantiation* and (2) the *pattern development*.

For both issues, we refer to Tables 5.2 and 5.3 where we provide timings for most of the shape skeletons within this chapter, excluding browser rendering times. These are captured on 64-bit Ubuntu 16.04 with Intel® Core™ i7-3820 CPU @ 3.60GHz i7 (8 processors) and 24GB of RAM. The test browser was Chromium 74. To have a reasonable idea of the peak performance, we ran the profiling by loading the skeleton file, then switching to *Compact* mode, and generating the output 7 times consecutively before actually measuring the times we report here. This warm-up leads the browser to optimize hot functions with its Just-In-Time compiler. We then export the machine code 3 times before measuring the code generation time.

The compaction time isn't actually needed except for shapes using branches (here mainly `glove`) but our implementation instantiates a new bed whether there is a need

Table 5.3: Summary of runtime performances of our system for the shapes within this chapter. All times are in milliseconds. *Code* is the additional step that happens when the user request the knitting program.

	⊢ Skeleton		⊣	⊢ Summary		⊣	*Code*
Name	***Nodes***	***Patterns***	***stitches***	***Shape***	***Pattern***	***Total***	***Generate***
cat-32x32	1	2	1024	33	51	63	27
cat-64x64	1	2	4096	65	62	86	40
glove	10	3	5030	116	119	143	46
cat-128x128	1	2	16384	278	259	375	77
sock	6	1	17740	272	265	361	101
corgy	1	4	19200	292	320	421	76
beanie	3	1	28774	482	386	606	96
noisy-scarf	1	1	85000	1335	1842	2369	335

for it (branching) or not, so the total update times might be smaller in practice if the user does not toggle bed compaction.

Generally, these timings are not meaningful as absolute numbers, but to understand the relative profile of different processes w.r.t. different shape complexities. Both shape and pattern updates have mostly linear runtimes as shown in Figure 5-25 for the considered shapes. However, the complexity of user patterns is of course going to be dependent on the user pattern. The linear behaviour for pattern development comes from the fact that filter operations traverse all stitches, and thus most of our operations end up being linear in that number.

Instantiating Stitches

The current implementation creates the layout from scratch at each update. This makes the implementation very simple, but prevents reuse of most of the fixed data. Unfortunately, reusing stitch information is not easy because simple modifications can have drastic impacts on the whole needle bed (e.g., adding a course at the beginning shifts all following stitches). For the shape part, our system runs mostly linearly in the number of stitches, and thus we are limited in the size of our garment shape.

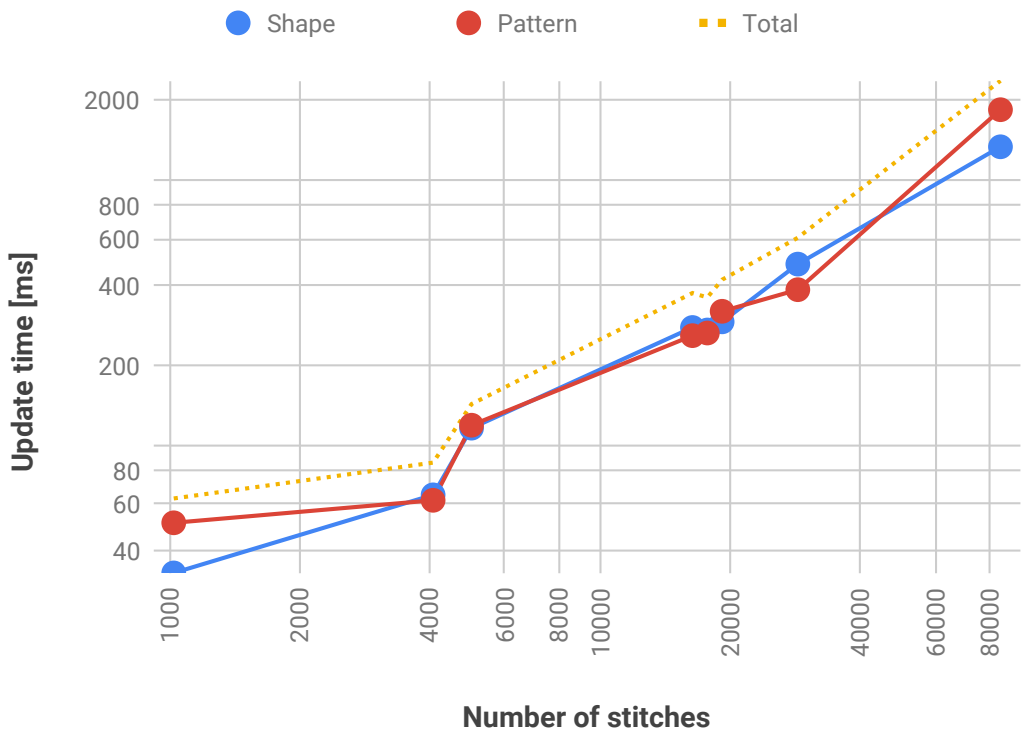


Figure 5-25: Plot of the update times for the models in this chapter, as given in the summary of Table 5.3. Shape update includes the time to rendering, minus the pattern development. Pattern update includes the time from pattern development to rendering. Both stitch and time axes have logarithmic scales.

When purely editing the pattern, we avoid recreating the whole data-structure since the pattern development does not remove or add stitches. Instead, we clear the stitch operations of all stitches, and re-apply the patterns, interpretation, simulation and compaction steps (since these are dependent on both the shape and the pattern).

Pattern Development

The pattern code evaluation is of course going to be longer to evaluate the more complex the pattern is. Our DSL implementation packs all stitches in a linear array and then mainly relies on filtering the set of indexed stitches for the queries, whereas the operations are done by traversing the current selected indices and applying the operation on the corresponding stitch objects. The implementation we provide performance for is purely CPU-based.

We expect possible improvements through a GPU implementation that would

compile our DSL as shaders or other compute programs for GPGPU.

Toward Large-Scale Interactivity

As shown in this section, the system can currently remain interactive with human-sized gloves, socks and beanies. Patterning or shaping full-sized sweaters or sweatpants is challenging because of their scale. Computationally, the garments require processing a very large amount of stitches. Their size also presents challenges for user pattern specification, as simple pixel-based operations are insufficient.

We expect to solve the computational challenge by using hierarchical data structures that do not instantiate all stitches but only the required information (e.g., at the boundaries or where the size changes). As for the design issue, we assume a similar hierarchical process would help. We envision using meta-patterns on higher-level stitch representations to be instantiated for the machine. Finally, recent patch-level pattern simulation [97] has shown promising interactive results.

5.5.4 Missing yet Desirable Features

Handling Multiple Yarn Carriers

The current system does not yet support multiple yarns on the bed at once (e.g., for intarsia). However we envision that such specification can be done similarly to our patterning by additionally introducing a stack of yarn at each stitch. The main modification would be that yarn tracing would now also involve the specification and optimization of the different yarn interactions. This would allow not only intarsia but also functional fiber routing, yarn inlays, spacer fabric, and pockets. The main difficulty lies on how to provide necessary user controls since there are often more than one way to schedule multiple yarn carriers in parallel, yet their interactions are often critical (i.e., avoiding tangling, properly connecting intarsia blocks, etc.).

Machine Independence

Currently, our system exports machine code for a set of specific machine targets: "whole-garment" knitting machines [150]. However, our design assumptions are that of general V-bed machines. Since our system uses very regular course structures, it should be easy to support exporting *Knitout* code [115], allowing us to potentially target other V-bed machines as well.

5.6 User Experience

To verify that our interface could be used by non-expert users and receive important feedback, we asked two potential users without prior knitting experience to use our system.

5.6.1 Procedure

The users were provided a 30 minute introduction to the basic operations involved in machine knitting, including the notions of shaping and the types of stitches. We also supplied a few sample videos of expert user sessions, and an introductory document for our user interface. We asked them to complete a few tasks: the first ones about patterning, and the later one for shape and pattern customization.

Patterning Task

For the first subtask, the users were given a base skeleton with a single flat sheet, featuring an initial program pattern that flattens its margins (to prevent the single-sided fabric from curling up). Their task was to draw some additional patterns on the sheet.

In the second subtask, we provided examples of lace patterns, and asked the users to create their own lace involving at least a few move operations.

For the last subtask, we provided a more complicated template of a wristband, which is similar to the pocketed scarf in Figure 5-19. The users had to import a

Figure 5-26: Lace patterns generated during our non-expert user sessions. The leftmost pattern was an expert reference that we provided for inspiration; the center and rightmost designs were novice user results.

pattern from their previous subtasks and apply it on the main part of the wristband.

Images of the physical artifacts created for the second and third subtasks are respectively shown in Figures 5-26 and 5-27.

Shaping / Patterning Customization Task

The second task was to customize an adult-size garment of their choice, given an initial skeleton.

Our users chose to customize a beanie and a glove. For the beanie, users had to change the shape profile of the top section, and optionally modify the pattern of the brim or core sections. For the glove, the goal was to change global, shared design parameters (finger length and width, and/or cuff length) instead of directly changing the individual node widths. The corresponding results are shown in Figures 5-28 and 5-30. Figure 5-29 shows a closeup on the laced beanie and its brim.

5.6.2 Feedback and Results

Although none of the users had prior knitting skills, each one successfully designed sophisticated, machine-knittable structures that we were able to fabricate. Users were surprised by their new ability to customize garments, especially the beanie.

During the design process, users cited a mismatch between their perception of the garment size (based on our bed visualization), and size of the actual knitted result.

Figure 5-27: The third patterning task required users to transfer an existing pattern onto a provided wristband template. The patterns were either designed by users in a previous step, or selected from our repository of pre-tested designs. The top row shows an expert reference; the middle and bottom rows are from our users.

This suggests that, beyond local editing, it may be worth tuning the relative size of wales and courses to perceptually match that of the real yarn. However, realistic sizing is an open problem, as current tension parameters are hand-tuned and must be adapted for complicated patterns (e.g., changing the tension impacts the garment size substantially). It will require a better simulation that takes the yarn tension into account.

Figure 5-28: A reference beanie on the left and two customized beanies on its right. The rightmost one required a few passes to adjust the lace pattern and its tension.

Our users generally found the patterning interface intuitive, as the image editing analogy was sufficiently familiar. Still, it was difficult for them to reason about the ultimate effect of complex lace patterns. Eventually, they discovered that the mesh simulation was more helpful to preview the pattern impact, and made extensive use of it. They alternated between the two views (layout and mesh). Furthermore, one user found that complex skeleton constraints could hinder pattern experimentation. Instead, they preferred to design their patterns on separate, flat sheets, then import the design onto the final structure. These behaviours emerged organically.

Figure 5-29: Beanie closeup showing the main section's lace and the curled brim with a knit/purl zigzag.

Although our users were allowed to create new shapes, they did not actively try to do so in our task setup. This suggests that non-experts would likely prefer to start from templates. Our tool might also be valuable for professionals, designers and other expert users, but we have not validated such cases.

Figure 5-30: Three glove variants. The green one took multiple attempts because of the complicated tension requirements associated with continuous cross patterns.

Knittability Constraints

Flat patterns were always knitted successfully on the first try. However, this was not the case for complex patterns on tubular structures, such as for the adult-size garments.

We show three result beanies, the right one having required multiple iterations to work properly. Our machine translation had very few issues, but some patterns triggered complications during the knitting process, mainly because of fabric pile-up, which arises from non-optimal yarn tension.

In the case of the gloves, all fully knitted from start without pile-up, but we discovered that the sequence of knitting had some unexpected impact on the yarn. Fingers are typically suspended on the bed before being knitted over to create the palm and one user decided to use complex patterns on the fingers themselves. This led to previous fingers being excessively stretched while suspended, and the yarn locally broke. Finding the appropriate tension was the main complication for both

the patterned beanie and glove, which required a few trial-and-error attempts. We describe the evolution of the green glove of Figure 5-30 in the next section.

5.6.3 Example of Issues and Iterations to Fix Them

The user design of the green glove took multiple iterations to converge to a design that would knit properly on the machine, illustrated in Figure 5-31. The initial design was intending to add gripping texture under the fingers. It was going to do so with continuously shifted cross patterns. Furthermore, its finger ends were to be closed with some shaping at their ends.

The first issue arose from the finger ends that had used very quick increases, with unstable kickback operations next to future increases. This led to a few holes at the finger tips. This can be fixed by changing the shaping, but also shows that one would definitely benefit from being able to manually specify the expected shaping seams, since then one could route the yarn correctly and avoid unstable increases.

The more interesting issue came from the move patterns on the fingers. The basic glove skeleton we used was starting the knitting from the fingers to avoid having to split the fingers into multiple knitting sections. This would have been required if starting from the cuffs, because a large knitted section is eventually suspended (the palm), from which smaller branches extends, one-by-one. In that case, the tension at the boundary of the branches increases the farther the finger knitting is, which eventually leads to the yarn breaking.

Typically, this can easily be solved for gloves by starting the knitting from the fingers. However, one issue is then that the fingers are kept suspended until the palm can be knitted (i.e., all fingers have been knitted). Unfortunately, this means that patterning the individual fingers with patterns that include large movements leads to the suspended yarn being stretched continuously. This is possible, but ended up breaking the yarn in random locations with the tension we used. A scheduling solution to this in half-gauge would be to collapse suspended structures onto a single bed side, as done in the scheduler of Narayanan et al. [122].

Instead, our user decided to move the pattern to the main palm, which solved the

Figure 5-31: The evolution of the green glove from right to left.

issues with the fingers (after also improving the shape of finger tips to be stable). Unfortunately, knitting continuous cables over a large area requires a correct tension parameter. Our initial setup was incorrectly estimating that the required yarn tension should be very loose (because of the many cables), but effectively they all combined into small 2×1 shifts, that do not stretch the yarn beyond 2 needle pitches globally. Having a wrongly loose tension typically leads to the yarn not catching correctly, and more unfortunately in it catching onto wrong needles on the sides. This leads to unpredictable behaviours and eventually led to fabric pile up (stopping the machine and also breaking a needle during the clearing process). Fixing the tension led to a completely fine glove.

This illustrates that although our system can allow users to create programs for shaping and patterning, there are still components missing to allow fully automated knitting as a service. We still require some expertise in how to handle the tension and other machine parameters, which brings up many interesting avenues for research.

Chapter 6

Sketch-based Garment Workflow

The traditional representation of garments for mass manufacturing is based on sketches that represent panels of fabric to be cut and then sewn together – the so-called cut & sew process. While the whole-garment knitting machines we consider in this thesis are typically used to *avoid* any cutting and sewing a posteriori, we consider here whether a similar sketch-based representation for knitted garment design is possible. The main question this chapter asks is the following:

> “How can we transcribe traditional garment sketches for whole-garment weft knitting?”

Cut & sew brings a plethora of design knowledge and resources, whereas whole-garment weft knitting promises digital customization together with important waste reduction through on-demand production. This chapter proposes a novel workflow that bridges both sides.

Contents of this chapter are adapted with permission from - A. Kaspar, K. Wu, Y. Luo, L. Makatura, W. Matusik, “Knit Sketching: from Cut & Sew Patterns to Machine-Knit Garments”, SIGGRAPH 2021. https://doi.org/10.1145/3450626.3459752. [Copyright by the authors].

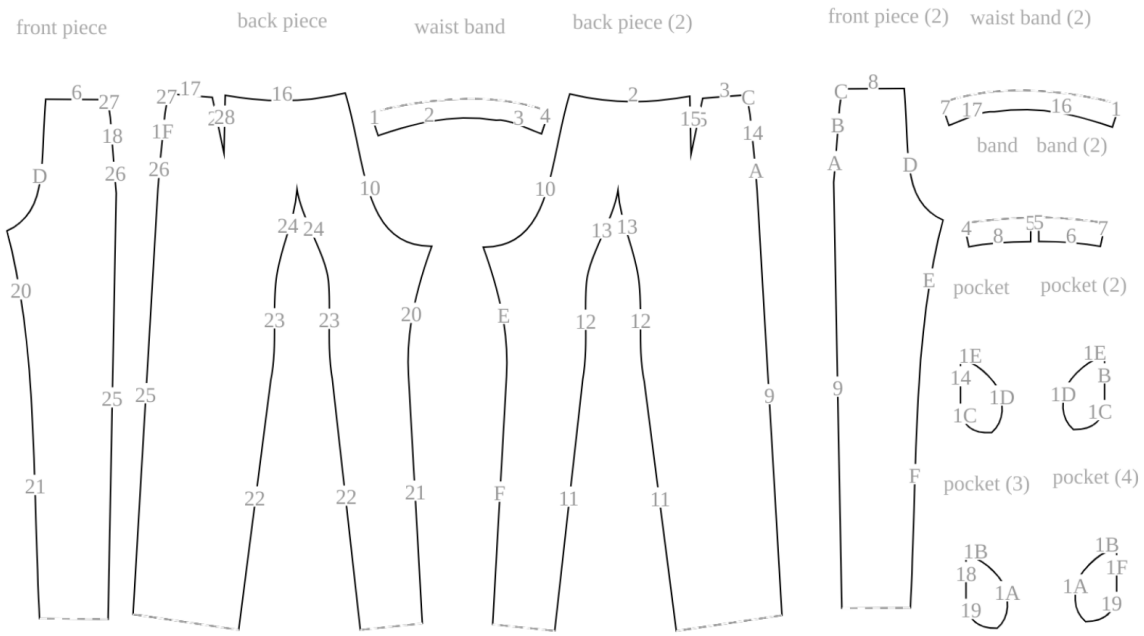


Figure 6-1: The segmentation of a sewing pattern for a pair of trousers with inseam pockets. Solid lines are linked by numbers, whereas dashed lines are not linked (i.e., they form open boundaries of the garment).

6.1 Traditional Garment Workflow

In the traditional garment making workflow, several flat panels are cut from 2D fabric and then sewn together along shared seams. Figure 6-1 illustrates an example of garment pattern traced from the fashion magazine BurdaStyle [1]. The 3D structure of the resulting garment (e.g., curvature, topology) can be arbitrarily complex, but it is fully prescribed by the 2D panel boundaries and their connectivity. That is, the panels capture the garment's 3D structure *intrinsically*. It is appealing to work in this lower-dimensional panel space because the intermediate (and resulting) blueprints can be easily edited, and they convey the designer's intention in a simple, compact, and precise manner. There is also a rich collection of sewing patterns available online for various clothing styles (e.g., BurdaStyle, Deer&Doe), and customization is straightforward with existing industrial design software, which offers short cycles between design and fabrication [25, 39, 113]. Most physical patterns typically come with grading information allowing for manual tuning of the size [119, 148], whereas

[1] http://www.burdastyle.com

some online collections provide customizable parametric pattern (e.g., FreeSewing [2]). However, there is no clear way to design knits for whole-garment knitting directly via a cut & sew pipeline. The fabrication process is inherently time-dependent and requires extra information during the design stage.

In this chapter, we combine the strengths of whole-garment knitting and the cut & sew design pipeline. For the garment design phase, we develop a user interface based on the powerful, low-dimensional representation from cut & sew. Then, for efficient garment construction, we propose to automatically translate the 2D panels into a full garment that is machine-knittable. Since our approach constructs the garment and its constituent fabric simultaneously, we can offer additional control over the *interior* of each panel, rather than being limited to the boundary.

6.1.1 Digital Garment Design

Interactive physically-based garment design is a challenging problem of long-standing interest [176]. Sketch-based design pipelines are particularly prominent [46, 77, 166, 179], because the familiar 2D-to-3D approach allows designers to use their existing experience and intuition. Other works allow designers to edit the garment directly in 3D space, by sketching the desired fold pattern of the draped fabric [102] or directly modifying garment shape [17]. After making the desired edits in 3D, the corresponding 2D patterns are generated via a simulation framework that incorporates design constraints. Utilizing design sensitivity analysis, Umetani et al. [169] presented an interactive tool for garment design that allows interactive bidirectional editing between 2D patterns and 3D draped garment shape. Several methods adjust parametric shape patterns to customize an existing pattern for specific individuals, such as profile template encoding/decoding [178], gradient descent method w.r.t. parametric patterns [118, 179], and learning-based methods [60, 181]. Berthouzoz et al. [20] combine machine learning with integer programming techniques for automatically parsing BurdaStyle sewing patterns and converting them into 3D garment models, whereas Shen et al. [153] combine sewing patterns and 3D body mesh data using a Generative

[2] http://freesewing.org

Adversarial Network. Finally, Huang et al. [73] generate garment models directly from a pair of front and back images. By contrast, we are interested in transforming sewing patterns into instructions for garment production on weft knitting machines.

6.2 From Sketches to Knitting Programs

A whole-garment knitting workflow based on cut & sew patterns presents several technical challenges. Since knit fabric relies on sequentially interlocking loops, the standard cut & sew panel representation must be augmented with time and direction information. Moreover, some cut & sew patterns do not immediately yield machine-knittable garments: in some cases, the knitting sequence produces undesired artifacts; in other cases, valid knitting sequences may not exist at all. To allow users to iteratively refine such designs, our system must offer rapid inference of the garment's final 3D structure and computation of the high-level knitting sequence.

A natural approach would be to combine existing methods that translate cut & sew patterns to 3D meshes [20], and then translate these into knitting programs using recent computational knitting methods [122, 123, 188] as illustrated in Figure 6-2. However, given that the complexity of both translations and their computational costs prevent interactive design, we choose to bypass the 3D mesh representation. Our pipeline operates exclusively in the 2D domain shown in Figure 6-1, which corresponds to the standard flattened view available in professional garment editing software. In particular, our computational workflow only relies on intrinsic surface metrics and local connectivity, thus bypassing a global 3D embedding of the desired garment (e.g., a 3D mesh). Although a 3D preview would still be helpful for designers, our work shows that all required knitting information can be inferred and efficiently computed from the intrinsic 2D representation.

Our high-level knitting sequence uses a representation similar to Narayanan et al. [122], with a time function over the sketch manifold that details the relative fabrication order among different areas of the garment. We develop new ways to solve for the time information and generate low-level stitch placement and knitting programs without

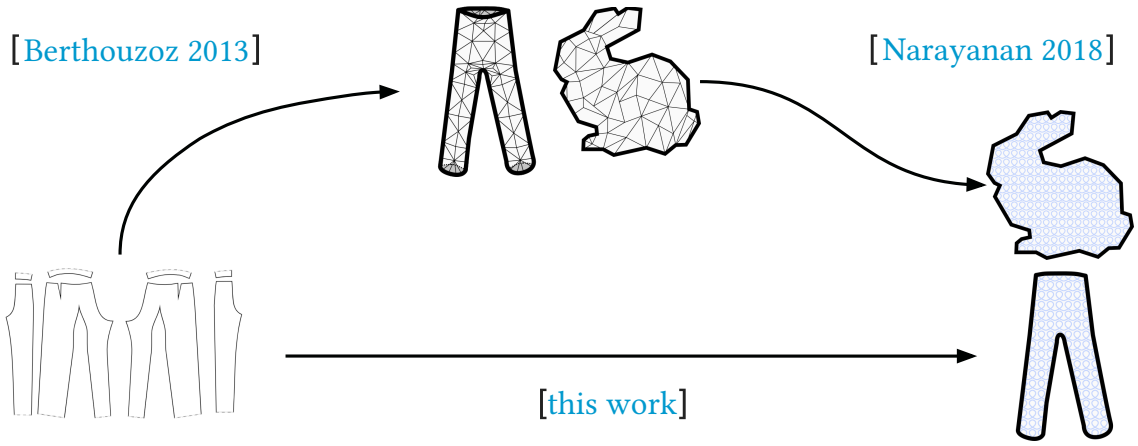


Figure 6-2: Illustration of the domains tackled by current workflows: from sketches to 3D meshes and back (digital garments), and from 3D meshes to machine knitting (concurrent workflows). Our workflow bypasses the 3D representation completely.

the use of a 3D mesh. Lastly, to support a wide range of garment structures, our scheduler provides basic support for mixed planar and tubular structures, which is a challenging problem not addressed by previous works: Narayanan et al. [122] compute a schedule dedicated to compound, tubular structures whereas Wu et al. [188] focus on complex flat panels that need to be bound manually.

In the rest of this section, we present an overview of our proposed user workflow. The remaining of the chapter provides details about our computational workflow: Sections 6.3 to 6.5 are the core of our new workflow (time computation, region decomposition and stitch sampling). Sections 6.6 to 6.8 detail our implementation of the rest of the pipeline, based on the work of Narayanan et al. [122]. Section 6.9 introduces our additional layer-based form of customization. Section 6.10 discusses results and Section 6.11 finishes with performance considerations for full interactivity.

6.2.1 Proposed User Workflow

Our approach hinges on the fact that complex 3D garments can be partitioned into a set of simple, closed regions that can be embedded within the 2D plane as in the traditional cut & sew workflow. In differential geometry, each region of the garment's 3D manifold is called a *chart*, and the 2D embedding is the chart's image under the flattening function. For notational simplicity, we use the term *chart* to refer to the

Figure 6-3: Summary of our workflow: (a) the user sketches a garment, links its boundaries and specifies time constraints; (b) the corresponding time function is computed, and its regions segmented; (c) given user sampling preferences (size and course/wale ratios), a stitch graph is sampled; (d) the user can provide additional seam annotations to influence the wale distribution until satisfied; (e) given knitting preferences, a schedule is generated and the physical artifact can then be knitted.

flattened 2D domain. As in differential geometry, the collection of 2D charts that fully prescribes a given garment is called an *atlas*.

This section provides an overview of our design process for a given atlas, as illustrated in Figure 6-3.

Sketching The user starts by inputting the desired atlas. Each chart is specified by its boundary shape, which is given by a closed poly-Bezier curve. The user can either draw the charts from scratch or import external SVG files (e.g., from existing cut & sew patterns).

Boundary Linking Users must also indicate the charts' intended connectivity by annotating boundary segments that should be linked in the final garment. Practically, each Bezier curve along the chart boundary is a linkable boundary segment. A pair of boundary segments should be considered *linked* if they are co-located on the assembled garment. We restrict the design of the base shape to be 2-manifold so that any boundary segment can be linked to at most one other segment.

Time Function Specification Knitting is a time-dependent process. The time function characterizes many important features of the garment such as the relative location of stitches and course/wale orientations. The user can design their own knitting time process by specifying different types of time constraints over the atlas.

Time and Region Computations Once the user has provided a linked atlas with the desired time function constraints, our system automatically solves for the time t over each chart (Section 6.3). Our system also provides feedback about the feasibility of the time function w.r.t. the knitting program space, and reports any notable physical issues, i.e., excessively large local time stretch which may lead to yarn breakage. Based on this time function, our system decomposes the atlas into simple regions (e.g., tubes and sheets) that are straightforward to knit (Section 6.4). These often coincide with semantically meaningful portions of the garment, such as the sleeves, torso, and yoke of a sweater. These steps are solved at an interactive frame rate, which allows the user to get continuous feedback as they adjust their desired shape and constraints.

Stitch Sampling Once the user converges to a garment specification, they set the desired sampling size (from sketch space to physical units) as well as course and wale sizes. Our system then constructs a *stitch graph* that, when knitted, will yield the desired garment (Section 6.5). The user can further tweak a set of weights to control the relative tradeoff between the size *accuracy* of the garment and the topological *simplicity* of the final stitch graph.

Scheduling and Fabrication Given the stitch graph, our system traces the real path of the yarn (Section 6.6), schedules stitches onto needles over time (Section 6.7), and outputs machine-independent instructions (Section 6.8). This takes into account any user preferences for fabrication (e.g., the type of increase stitch to use, and the cast-on/off procedures), and a list of user-specified *stitch programs* that map from stitch to knitting instructions, enabling colorwork and surface texture (Section 6.9). The resulting Knitout file [115] can be compiled for the target knitting machine before actually knitting it.

6.3 Computing the Knitting Time Function

Given a garment atlas with multiple linked charts, the first computational step is to determine the knitting time over the domain. In particular, we must determine the order in which the garment is knit, the orientation of each stitch course (row), and the wale (column) connections between rows. We define a continuous scalar field of the *time* t over the garment atlas to represent when the knitting process happens locally. The courses align with the *time isoline* curves, along which t remains constant. The wale connections follow the direction of the time gradient $\phi = \nabla t / \|\nabla t\|$. This direction field should be as smooth as possible, because variations in the field imply local stretching or contraction between stitches. Excessive deviations cause visual artifacts and potential failures during the knitting process, so they must be avoided.

From an optimization perspective, we are seeking a function t whose gradient is intrinsically smooth, i.e., minimizing

$$\int_M \|\nabla \cdot (\nabla t)\|^2 = \int_M \|\Delta t\|^2 \tag{6.1}$$

over the garment atlas M, subject to user constraints (either *soft* or *hard*). The first set of user constraints are *direction constraints*; these are curves whose tangents or normals dictate the orientation of the direction field ϕ. The second set are *time equality constraints*, which specify individual time isoline curves.

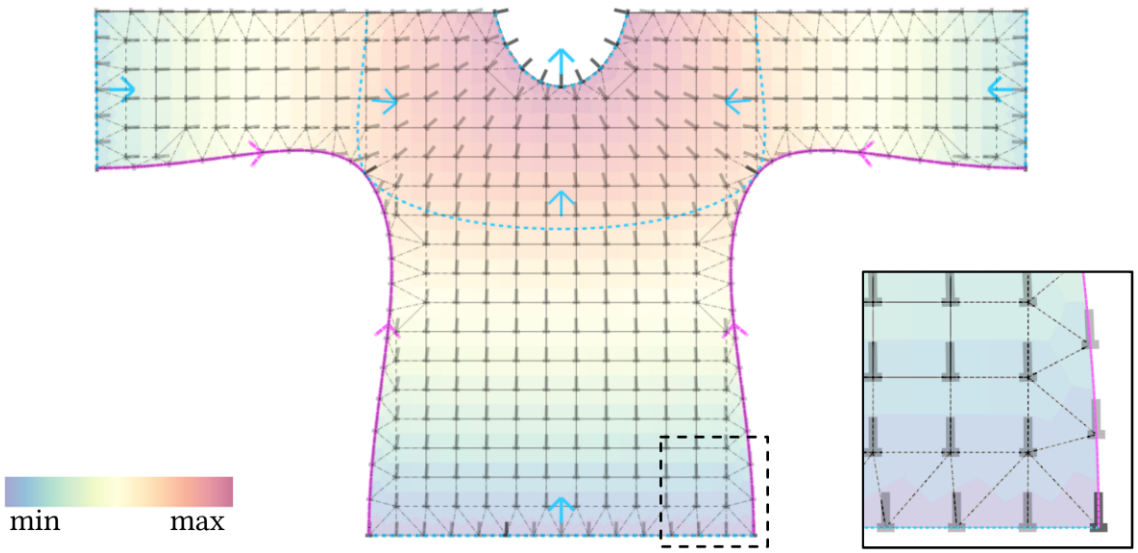


Figure 6-4: Color visualization of the time function over the back of a sweater, together with the underlying mesh illustrating the mixed quad-triangle neighborhoods. Each sample is annotated with a small tack $\perp$ representing the flow direction ϕ. The purple arrowed curves are flow direction constraints; the light blue curves with orthogonal arrows are time isoline constraints, with a given flow orientation.

Narayanan et al. [122] create a similar time function by specifying the start and end interfaces and interpolating the time in between with Laplacian interpolation. Instead, we provide control on intermediate isolines and the direction field, with an emphasis on interactive editing of the sketch domain and constraints. We solve for a suitable t automatically using a series of optimizations over increasingly fine chart discretizations.

6.3.1 Discretization

To discretize each chart into a mesh with a given resolution, we first generate *grid samples*, which are uniformly distributed throughout the interior of the chart using a regular grid with spacing Δ_s. Then, we generate a set of *boundary samples* to capture the boundary of each chart. These are distributed along the boundary as uniformly as possible while adhering to several guidelines.

In particular, we always require boundary samples at the start and end point of each boundary segment. If the boundary segment is not linked to any other segment,

we sample the rest of it using a uniform arc-length sampling that matches the local grid cell size, Δ_s. However, if the boundary segment *is* linked (i.e., it is co-located with another segment in the final garment), the linked boundaries must have a consistent representation that can be used to reconcile field values across the charts. To ensure this, we require a bijection between the samples on each linked boundary. Each pair of *linked samples* given by this bijection must be co-located in the final garment. With respect to the final garment, the boundary samples are distributed according to the larger of the linked charts' sampling rates. The spacing of the boundary samples may differ in each local chart embedding as the edge lengths of linked sketch borders are not required to match exactly.

The resulting boundary and grid samples are then connected to their neighboring samples in order to create the mesh over each chart. On the interior of the mesh, we use quads to connect the grid samples with their neighboring samples. The more complex boundary region between the interior and the border uses a Delaunay triangulation, as visualized in Figure 6-4.

For the sake of brevity, we introduce the following notation:

- A *vertex* v refers to some location on the inferred (but never explicitly instantiated) garment manifold. Each v corresponds to one or more samples embedded in the charts.

- $\mathcal{L}(v)$ is the set of samples that are images of v within the charts. $\mathcal{L}(v)$ has one element if v corresponds to an unlinked sample, or more than one if v corresponds to linked samples.

- $\mathcal{N}(s)$ is the set of neighboring samples that share an edge with sample s. All samples in $\mathcal{N}(s)$ must belong to the same chart, and cannot cross any boundary segments (including linked segments that belong to the same chart).

- $\mathcal{C}(s) = \{c | s \in \text{supp}(c)\}$ is the set of constraints which s is in the support of (i.e., c affects s directly).

Every sample s has an associated value for each of the two fields: the time $t(s)$ and

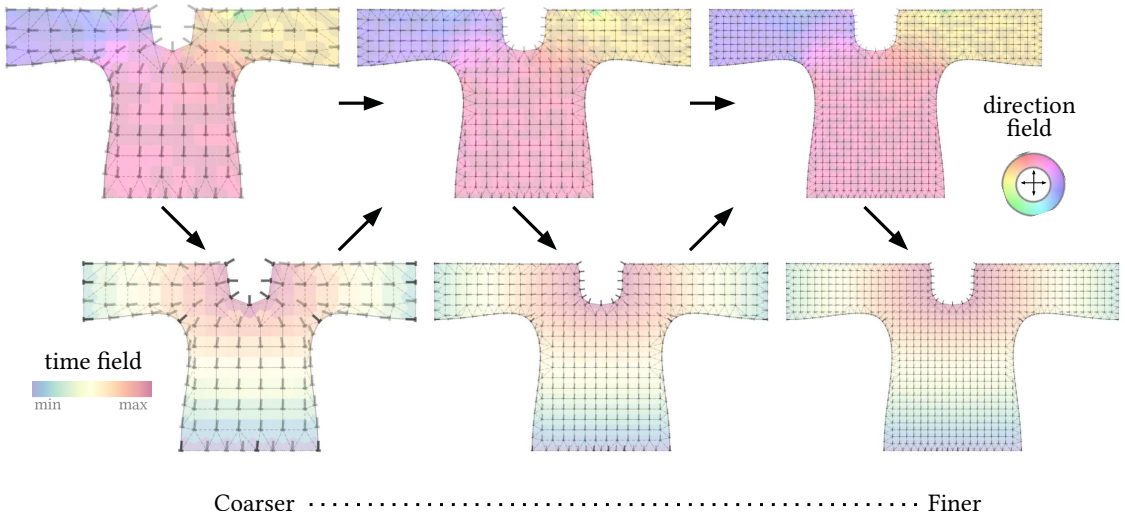


Figure 6-5: Illustration of the alternating iterations between solving for the direction field and the time field in a coarse-to-fine manner. Each mesh level and field (ϕ then t) is solved until convergence before moving to the next field and mesh level.

the direction $\phi(s)$. The quantities associated with linked samples are independent from one another, but we reconcile the values to ensure consistency. Both fields are extended over the entire chart domain by interpolation: linear over sample mesh edges, barycentric over triangles and bilinear over quads.

6.3.2 Computing Time and Direction Fields

Our strategy is to successively solve for the direction and time fields in a coarse-to-fine manner over meshes of increasingly higher resolution to ensure fast convergence, as illustrated in Figure 6-5. At each level, we solve for the direction field and integrate it to get the time function. To ensure interactivity and fast visual feedback, each optimization is done using Gauss-Seidel iterations that update the quantity at each sample. The updates are done first in the interior of the charts, and then along the border samples. The optimization stops once early termination criteria are satisfied or the maximum number of iterations have occurred. Then, the time and direction fields are upsampled for the next higher-resolution mesh, and the process repeats.

Solving for the Direction Field

We use the normalized direction averaging strategy of Jakob et al. [80] to efficiently solve for the direction field $\phi(s)$ at each sample s, namely:

$$\phi(s) \leftarrow \frac{\sum\limits_{s_{\mathcal{N}} \in \mathcal{N}(s)} w_{s_{\mathcal{N}}} \phi(s_{\mathcal{N}}) + \sum\limits_{c \in \mathcal{C}(s)} w_c \phi_c(s)}{\sum\limits_{s_{\mathcal{N}} \in \mathcal{N}(s)} w_{s_{\mathcal{N}}} + \sum\limits_{c \in \mathcal{C}(s)} w_c}, \; \phi(s) \leftarrow \frac{\phi(s)}{\|\phi(s)\|}, \tag{6.2}$$

where $w_{s_{\mathcal{N}}} = 1/\|p(s_{\mathcal{N}}) - p(s)\|$ and $w_c = \gamma_c / \|\Pi(s, c) - p(s)\|$ with γ_c being a per-constraint, positive, user-tunable weight. ϕ_c is the fixed direction that constraint c enforces on s; $p(s)$ refers to the position of s in local chart coordinates; and $\Pi(s, c)$ is its Euclidean projection onto the curve of c. For all samples in the support of *hard* constraints, we set $w_{s_{\mathcal{N}}} = 0$.

After each iteration over the full atlas, the directions across linked samples are reconciled to ensure a consistent solution: the samples must have the same orientation, but can have either the same direction (through-flow) or an opposite one (source or sink).

To compare direction vectors across different charts, a common coordinate system is necessary. Given vertex v, all the linked directions are rotated into the domain of one of the charts associated with v. Then, the average orientation is computed and transformed into individual directions that are rotated back to the local domains of the corresponding linked samples.

Integrating the Knitting Time

After the direction field has converged, we iteratively propagate the time over the atlas. On each iteration, the time is computed by (1) integrating it locally over the full atlas, (2) enforcing the time isoline constraints, and (3) averaging the time across linked samples across charts.

Before starting, we select one seed sample s_{seed} with a large neighborhood to propagate from, and fix its time to be $t(s_{\text{seed}}) = 0$.

Step 1. The time integration uses the converged direction field ϕ to update the time at vertex v based on the time values at its neighbors:

$$t(v) \leftarrow \frac{1}{|\mathcal{L}(v)|} \sum_{s \in \mathcal{L}(v)} \frac{1}{|\mathcal{N}(s)|} \sum_{s_\mathcal{N} \in \mathcal{N}(s)} [t(s_\mathcal{N}) + dt(s_\mathcal{N} \rightarrow s)]. \tag{6.3}$$

The $|\cdot|$ operator is the set cardinality and $dt(s_\mathcal{N} \rightarrow s)$ is the expected local time difference, computed as the dot-product $(\cdot)$ between the average direction and the position difference:

$$dt(s_\mathcal{N} \rightarrow s) = \frac{1}{2}[\phi(s_\mathcal{N}) + \phi(s)] \cdot [p(s) - p(s_\mathcal{N})]. \tag{6.4}$$

Step 2. The time isoline constraints are enforced by averaging the contribution of all samples within their support, and then back-propagating that average time to the individual samples. We average the expected time *after projecting the samples* onto the curve of the isoline constraint:

$$t(c) \leftarrow \frac{1}{|\text{supp}(c)|} \sum_{s \in \text{supp}(c)} \big[t(s) + dt[s \rightarrow \Pi(s, c)]\big], \tag{6.5}$$

$$t(s) \leftarrow t(c) - dt[s \rightarrow \Pi(s, c)]. \tag{6.6}$$

Step 3. Finally, the time is averaged across linked samples:

$$t(v) \leftarrow \frac{1}{|\mathcal{L}(v)|} \sum_{s \in \mathcal{L}(v)} t(s), \quad \text{then} \quad t(s) \leftarrow t(v)\big|_{s \in \mathcal{L}(v)}. \tag{6.7}$$

6.3.3 Termination

For each field, we measure the variation of the field among the samples inside of the charts at the end of each iteration I and stop the computation when it is below a given threshold, i.e., $\max_s |1 - \phi_I(s) \cdot \phi_{I-1}(s)| < \epsilon_\phi$ and $\max_s |t_I(s) - t_{I-1}(s)| < \epsilon_t$, respectively. Once the time function has been solved over our finest resolution sample mesh, the system checks its validity. If the function is deemed invalid, feedback is provided to the user and the processing stops. Otherwise, a topological opening is

applied to normalize the shape boundaries and simplify the later region computations.

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 Section 6.3.4 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. [122]. 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 Section 6.3.5). 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 Section 6.7, it also makes it hard to provide guarantees w.r.t. to knittability without restricting the design space.

6.3.4 Curvature and Time

The time integration from Equation 6.3 assumes that the time function has unit magnitude everywhere. Although individual sketches can be viewed as planar surfaces, the sketch atlas is in general not a developable surface. Furthermore, time isoline constraints typically induce local curvature.

We investigated the addition of a *curvature* term $\kappa = \|\nabla t\|$ as part of our time function decomposition. Our time integration update stays the same as Equation 6.3 whereas the local time difference $dt(\cdot \to \cdot)$ is updated to includes the local magnitude $\kappa(\cdot)$ to scale the direction as

$$dt(s_{\mathcal{N}} \to s) = \frac{1}{2}[\kappa(s_{\mathcal{N}})\phi(s_{\mathcal{N}}) + \kappa(s)\phi(s)] \cdot [p(s) - p(s_{\mathcal{N}})]. \tag{6.8}$$

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.

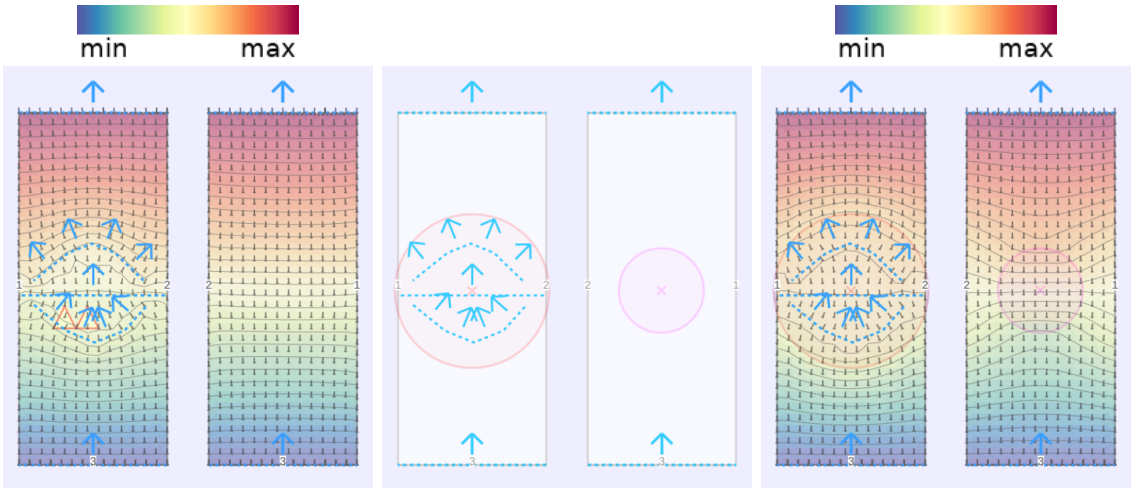
Specifically, close-by isoline constraints can induce large local curvature, and those specific cases tend to make the integration unstable in the region of induced high curvature as shown in Figure 6-6. In such cases, specifying the local curvature $\kappa(\cdot)$ becomes necessary to get a proper time function without local time extrema in the interior of the sketch domains.

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

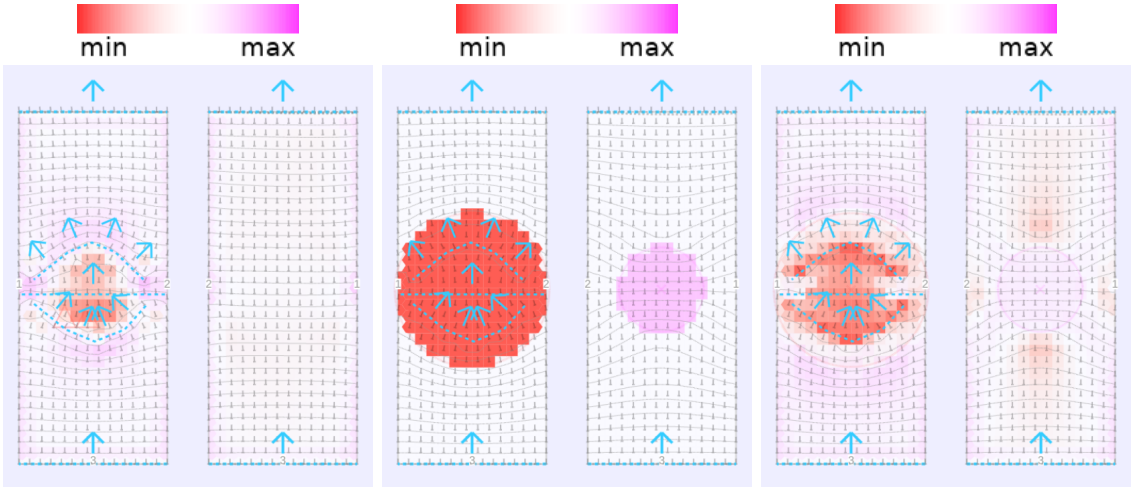
$$ts(v) = 2 \times \frac{\sum_{s \in \mathcal{N}(v)} |t(s) - t(v)|}{|\mathcal{N}(v)|}, \tag{6.9}$$

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



(a) Time, $\kappa = 1$ (c) Curvature annotations (e) Time, κ_u



(b) Time stretch, $\kappa = 1$ (d) User-defined κ_u (f) Time stretch, κ_u

Figure 6-6: 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 time result without user curvature, which has invalid local time extrema in the center-left section (triangular warning signs); (b) the corresponding time stretch (red when $\kappa < 1$, pink when $\kappa > 1$); (c) the closed tubular rectangle with time constraints and curvature annotations; (d) the corresponding 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.

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.

6.3.5 Topological Opening

As a post-processing step after the time function computation (or pre-processing step before the next 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 could potentially 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. 6-7 illustrates both, using the top of the beanie as an example sink to be opened. In general, the sources/sinks are either:

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

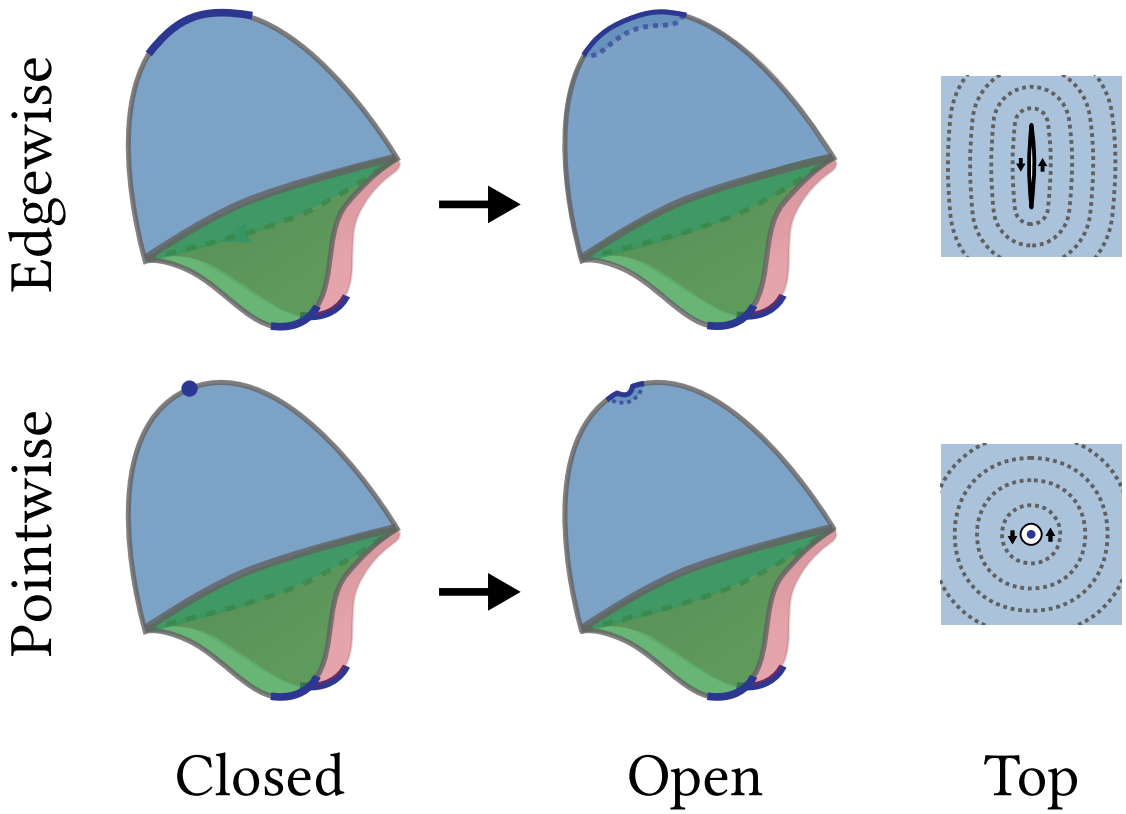


Figure 6-7: Illustration of the topological opening at the closed top of the beanie for two different **time extrema**: edgewise and pointwise. The top views show the opening in the center and a corresponding isoline profile (dashed lines). The extrema on the earflaps are already on the manifold boundaries and do not need opening.

top, and manually close by passing thread across all last stitches and pulling a thread which we close inside of the beanie.

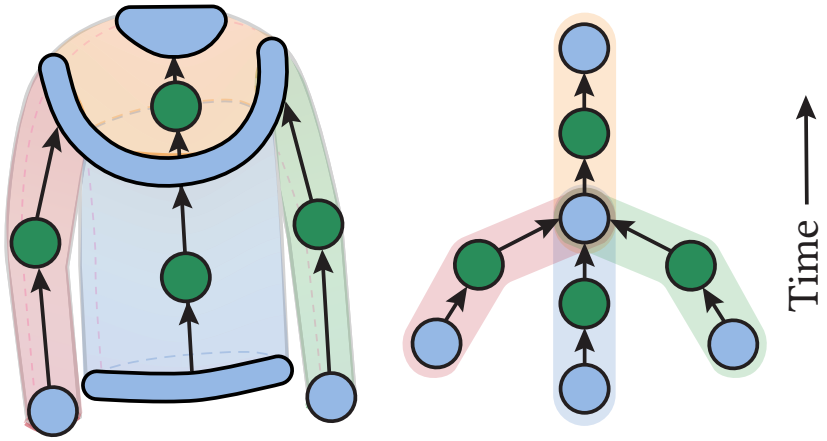
6.4 Region Graph Construction

The next step is to automatically decompose the linked garment into a minimal set of regions that are simple to knit, such as tubes and flat sheets, while conforming to the time and direction fields. A *simple region* must be knittable using only traditional forms of shaping, including stitch increases/decreases and short-rows. In particular, simple regions cannot contain non-trivial topological features like splits or merges. The time isolines that serve as *interfaces* between a garment's minimal set of simple regions are called *critical isolines*.

(a) Input time function on the garment manifold

(b) Isoline segments and separating vertices (■)

(c) Dependency paths for resolving the regions



(d) Equivalent views of the bipartite region graph relating the garment regions and isoline interfaces over time

Figure 6-8: Illustration of the steps of our region computation: (a) we start from the time function defined on the garment atlas, (b) we trace a set of isolines that is sufficient to segment the sketch domain into simple regions to knit, each isoline being further decomposed into different oriented segments, (c) we create regions on each side of the isoline segments and merge them by following dependency paths along the sketch manifold, and (d) we create the corresponding bipartite graph with 2-coloring separating nodes into regions and isoline interfaces.

These *critical isolines* include all isolines that coincide with:

- a topological split or merge,
- a topological change (from flat to tubular, or vice versa), or
- a boundary of the final garment manifold (e.g., the edge of a cuff or neckline).

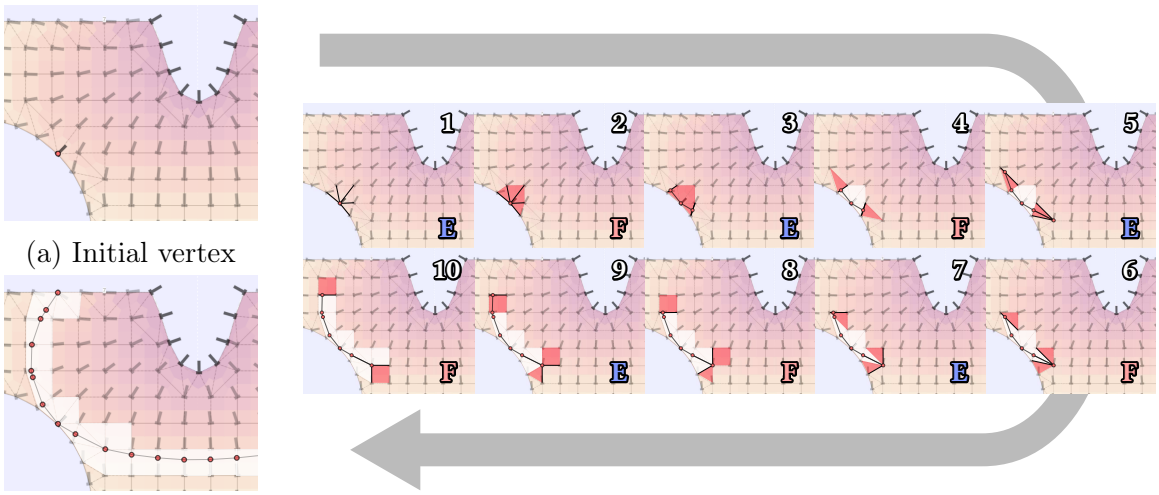
Note that regions may be bounded by some *portion* of a given isoline if the isoline coincides with a topological split, merge, or change, e.g., when joining the sweater sleeves and trunk into the yoke region in Figure 6-8b. Thus, we denote each simple region with a pair of lower and upper *isoline segment sets* $(\mathcal{S}^{\text{low}}, \mathcal{S}^{\text{up}})$, in which each set $\mathcal{S}$ contains a number of isoline segments $\{\sigma_0, \sigma_1, \ldots\}$ at time isoline L^{low} and L^{up}, with time values $t(L^{\text{low}}) < t(L^{\text{up}})$, respectively.

We aim to compute the set of critical isolines and simple regions, along with the *knitting time dependency* between them. However, it is difficult to directly extract all critical isolines, as topological structures are not always apparent from the linked atlas alone. Instead, we follow the 3-stage process outlined in Figure 6-8. Given a time-annotated garment, we first identify the collection of candidate isolines that are likely to serve as delimiters between neighboring regions (Section 6.4.1). Then, we build the directed connections between all simple regions (Section 6.4.2). Finally, we transform the region and interface dependencies into a directed acyclic bipartite graph (Section 6.4.3).

6.4.1 Tracing Candidate Isolines

We begin by identifying the set of vertices $\mathcal{V}$ from which we should trace candidate isolines. This includes vertices that are located at (1) *corners* (start or end of chart boundary segments), or (2) *local time extrema* along their boundary segment. The corner vertices are where the most common topological splittings and mergings happen; the time extrema along boundaries capture the remaining topological events (including flow sources and sinks). Note that $\mathcal{V}$ is *complete*, but not all its elements are *necessary*, e.g., subdividing a sketch boundary does not necessarily indicate a change in topology.

Beginning from vertex $v \in \mathcal{V}$ with time $t(v)$, an isoline is traced by alternating between two operations: (1) from a given neighborhood (vertex, edge, or face), find all adjacent edges that contain the given time t, and (2) from a given edge, find all faces that are adjacent to it. This generates a continuous isoline path over the garment manifold, as illustrated in Figure 6-9.



(a) Initial vertex

(c) Final isoline

(b) Sequence (1 to 10) of finding edges (E) and faces (F)

Figure 6-9: Illustration of isoline tracing: starting from a location (here a vertex), we alternate between the adjacent edges (E) that contain the given isoline time, and their adjacent faces (F), until we've traced the whole isoline domain. Here the color of the mesh visualizes its time function.

Any isoline paths that encompass non-trivial topological features (e.g., topological split, merge, or change) are then subdivided into multiple segments. The isolines are partitioned at the topologically critical points, or *separating vertices*, which are given by one of two scenarios. In the first case, a separating vertex coincides with more than two edges of the isoline for time t (as in the armpit of Figure 6-8b). The second case indicates the point at which an isoline path transitions between being on the *interior* of the garment manifold and being on its *boundary*. This is illustrated by the central isoline of the beanie, which separates the ear flaps from the main body in Figure 6-10. The isoline is primarily on the garment's interior, but it intersects the boundary at each of the two separating vertices (■). This is interpreted as a topological change over this isoline: the two lower flat regions merge into the upper circular body region.

6.4.2 Computing Regions from Dependency Paths

The next step is to uncover the simple regions by inferring the connectivity between the candidate isolines. Each simple region must be bounded by two sets of isoline

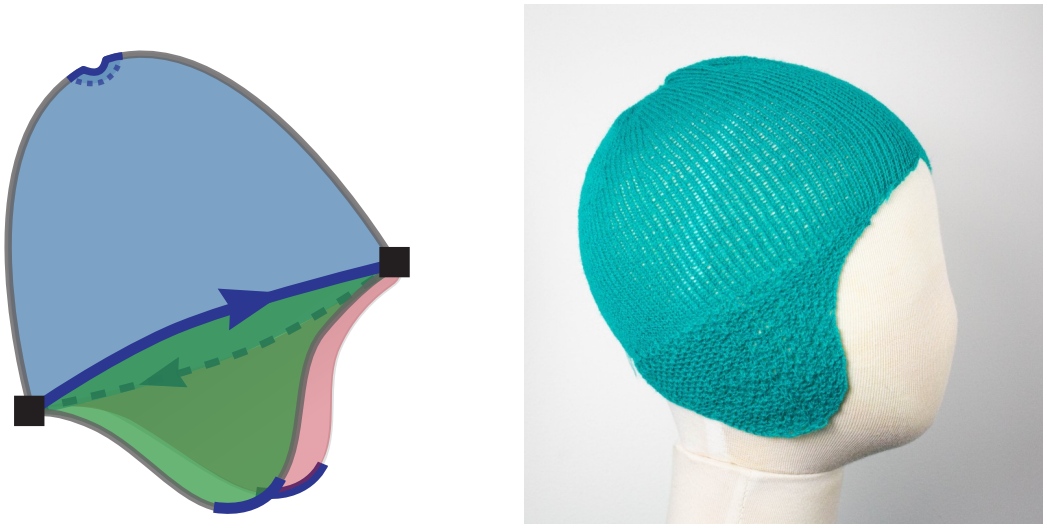


Figure 6-10: Examples of separating vertices (■) at the boundaries of the garment manifold. The central isoline is split into two segments separated by vertices that form transitions between being *inside* the shape (above each ear flap), and at its *boundary* (between both ear flaps). The top isoline surrounds a pointwise sink of the time function, which was topologically opened.

segments — $\mathcal{S}^{\text{low}}$ and $\mathcal{S}^{\text{up}}$ — that can be connected along a continuous path through the garment without passing through any other candidate isoline. Thus, the set of regions and their extents can be determined by tracing paths from each isoline segment to its next reachable neighbor, in order to confirm their local connectivity.

Each isoline segment σ_i can be associated with at most two regions: its *preceding* region (for which $\sigma_i \in \mathcal{S}^{\text{up}}$), and its *subsequent* region (for which $\sigma_i \in \mathcal{S}^{\text{low}}$). Initially, no other members of $\mathcal{S}^{\text{low}}$ or $\mathcal{S}^{\text{up}}$ are known, so it is only possible to allocate a partially-known region for each side of σ_i. Then, a dependency path is traced from each lower segment σ_i, eventually reaching another isoline segment σ_j (with $t(\sigma_i) < t(\sigma_j)$). This confirms that the subsequent region of σ_i and the preceding region of σ_j are identical and allows us to merge them into a single region with the union of the corresponding isoline segments on either side.

Our system generates dependency paths by following edges of the mesh in a specific time direction until some isoline segment is *reached*. *Reaching* essentially means that the last edge e of the path intersects an isoline. This intersection may occur within e or at an end vertex of e. In the former case, the dependency always indicates a

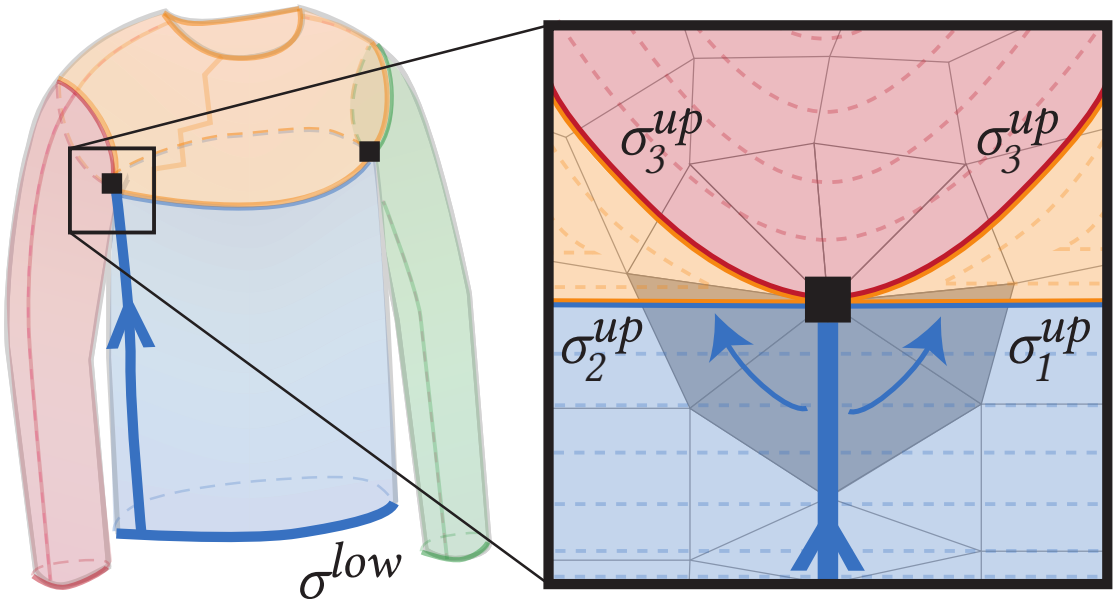


Figure 6-11: When a dependency path (blue line) reaches a candidate isoline at a separating vertex (■), we must take extra steps to determine which of the incident isoline segments (σ_1^{up}, σ_2^{up}, or σ_3^{up}) bound the region in question (blue). We decide this by traversing the triangle fan that surrounds the vertex, until reaching (or crossing) the nearest candidate isoline segment in each direction (σ_1^{up} and σ_2^{up}).

single isoline segment σ_j. If the latter case occurs at a separating vertex v, there may be several incident isoline segments that partition the local neighborhood around v into sectors representing distinct regions, as shown in Figure 6-11. In such a case, the dependency path only *reaches* the isoline segment(s) that delimit the sector from which the path originated.

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

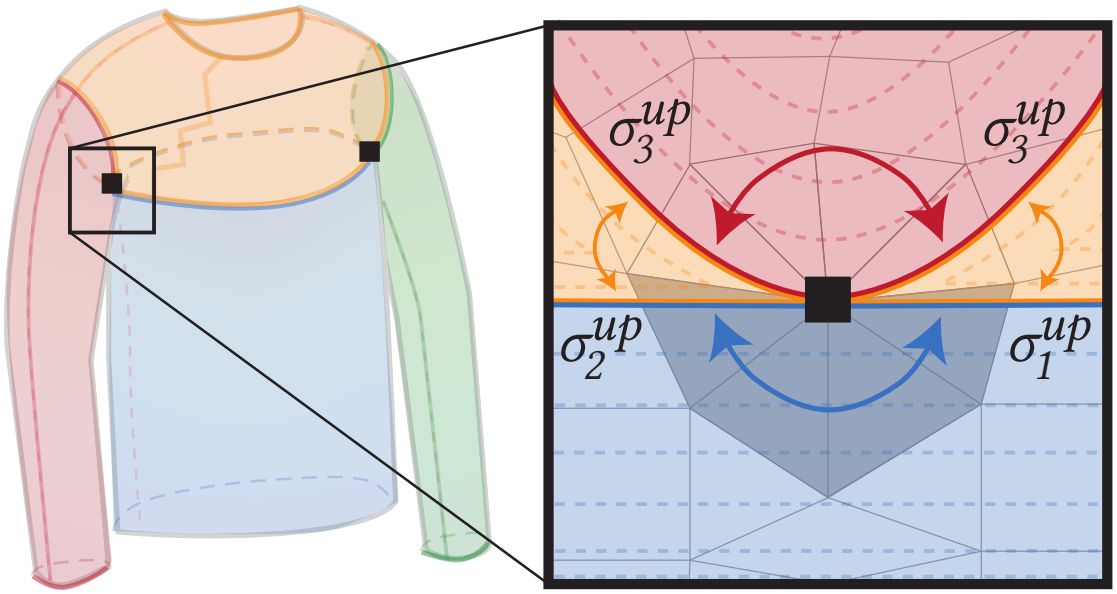


Figure 6-12: Illustration of the adjacent region merging at a separating vertex. Segments σ_1^{up} and σ_2^{up} from the trunk region can reach each others, thus the front and back span the same region. Circulating around the sleeve region has no effect since it is upper-bounded by a single segment σ_3^{up}. 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.

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. Figure 6-12 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.

6.4.3 Building the Bipartite Region Graph

Armed with the garment's topological structure, non-critical isolines are filtered out to produce the desired minimal set of simple regions. The *non-critical* isolines are those which (1) connect a single preceding region to a single subsequent region, **and** (2) have topologically identical structures on both sides (i.e., flat to flat, or circular to circular). Note that both criteria are necessary for pruning. For instance, if an isoline has a single previous and a single next region but its topology changes (from flat to circular or vice versa), the isoline is considered critical. Once a non-critical isoline is removed, its preceding and subsequent regions are merged.

After resolving the minimal set of regions, we construct a bipartite *region graph* that represents the final garment decomposition. This graph has a node set $\mathcal{I}$ to represent interfaces (critical isolines), another node set $\mathcal{R}$ to represent each simple region, and a *directed* edge set $\mathcal{E}$ to connect related isolines and regions. Each directed edge $\mathbf{e}_i \in \mathcal{E}$ corresponds to an isoline segment set $\mathcal{S}_i = \{\sigma_0, \sigma_1, \ldots\}$. Moreover, for a given interface node η, any incident edges in $\mathcal{E}_\eta^{\text{in}}$ originate at a preceding region/interface, and those in $\mathcal{E}_\eta^{\text{out}}$ lead to a subsequent one. This yields the final graph $G = (\{\mathcal{I}, \mathcal{R}\}, \mathcal{E})$ as shown in Figure 6-8d.

Graph Post-Processing and Δt_{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 Δt_{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

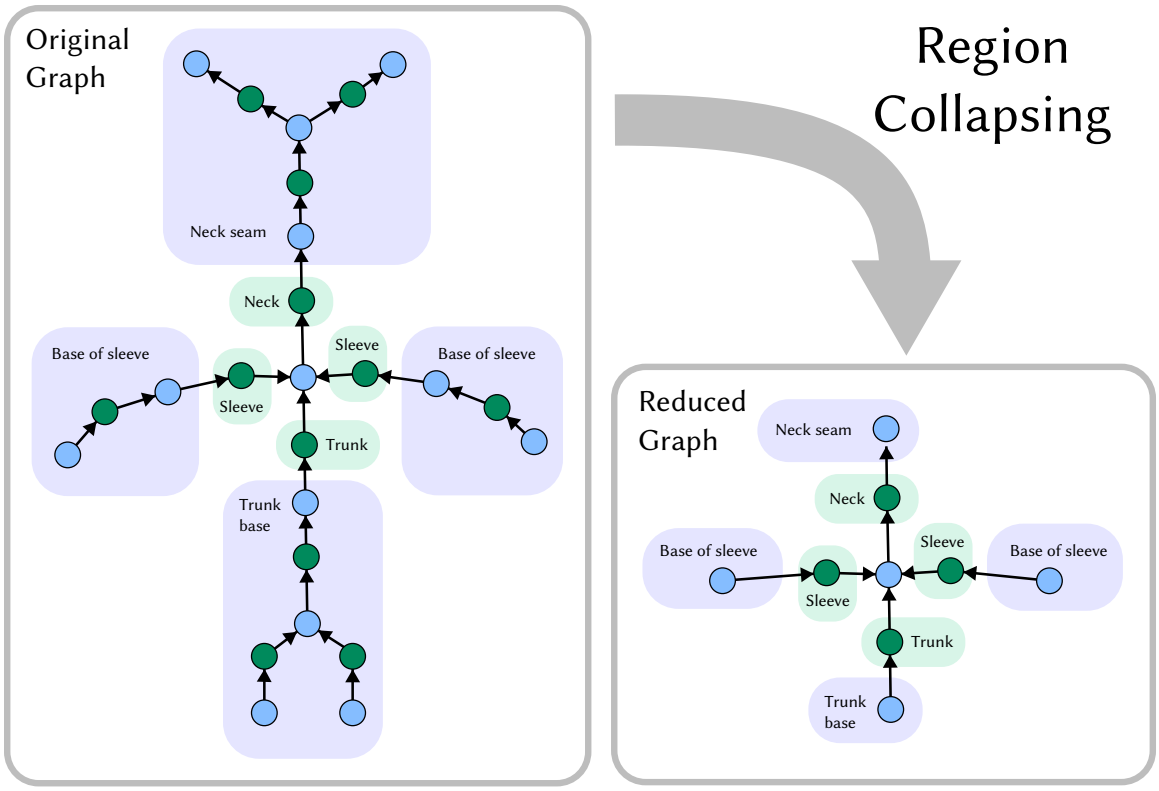


Figure 6-13: *(Left)* the original graph of the sweater example when the sources and sinks are not sufficiently constrained – e.g., no hard time isoline constraints are set on the sketch boundaries –, *(right)* its reduced graph that looks identical to the ideal one.

time extents, or there remains only one simple region. Simple regions that collapse become parts of new interface nodes.

Figure 6-13 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.

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*.

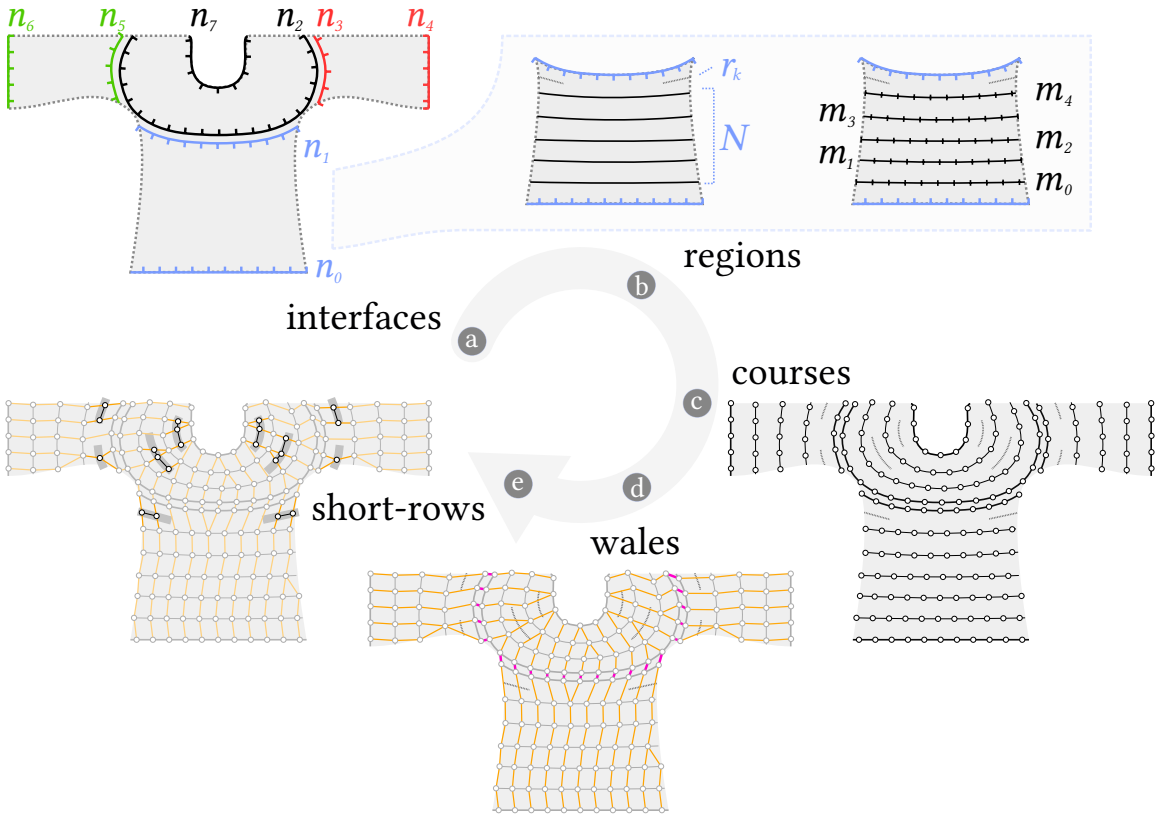


Figure 6-14: Illustration of the steps of our sampling algorithm: (a) optimizing stitch numbers at region interfaces, (b) optimizing course number, short-row densities and stitch numbers in each region, (c) creating stitch courses, (d) pairing stitches between adjacent courses across interfaces and within regions, and (e) generating short-rows.

6.5 Hierarchical Stitch Sampling

In order to generate machine knittable instructions from the region graph, a *stitch graph* must be instantiated. Our stitch graph computation is formulated as a global hierarchical optimization over the region graph and sketch atlas. Unlike previous works, each phase of our optimization accounts for both (1) the garment size accuracy and (2) its topological simplicity. These goals are fundamentally conflicting, because irregular topologies (shaping or short-rows) often improve size accuracy, but this typically happens to the detriment of a simple topology (regularity of stitches). Our system solves optimization problems that allow the user to navigate the tradeoff between garment size accuracy and topological simplicity. Moreover, our formulation enables the user to interactively control the wale alignment using seam annotations.

Our optimization approach has multiple stages illustrated in Figure 6-14. First, we optimize the number of stitches at each interface (Section 6.5.1). We then optimize the number of full courses and short-rows within each region and the number of stitches placed along each full course (Section 6.5.2). Next, we create all course stitches with their course connectivity, and optimize the wale connectivity across the interfaces and within each region, while taking the user's seam annotations into account (Section 6.5.3). Finally, we insert short-row stitches (Section 6.5.4).

6.5.1 Interface Sampling

To determine the stitch count n_i at each edge $\mathbf{e}_i \in \mathcal{E}$ within the bipartite region graph $G = (\{\mathcal{I}, \mathcal{R}\}, \mathcal{E})$, we formulate an Integer Quadratic Programming problem (IQP) with linear constraints:

$$\begin{aligned} \arg\min_{\mathbf{n}} \; & \lambda_{\text{crs}} \sum_{\mathbf{e}_i \in \mathcal{E}} E_{\text{crs}}(n_i) + \lambda_{\text{smpl}} \sum_{(\mathbf{e}_i, \mathbf{e}_j) \in \mathcal{R}} E_{\text{smpl}}(n_i, n_j) \\ & \text{s.t. } \forall\ \eta \in \mathcal{I}_{\text{internal}}, \sum_{\mathbf{e}_i \in \mathcal{E}_\eta^{\text{in}}} n_i = \sum_{\mathbf{e}_j \in \mathcal{E}_\eta^{\text{out}}} n_j. \end{aligned} \tag{6.10}$$

The first term E_{crs} measures the per-edge *course accuracy* for the stitch count n_i along $\mathbf{e}_i$:

$$E_{\text{crs}}(n_i) = \left| n_i - \frac{\omega_i}{D_{\text{crs}}} \right|^2, \tag{6.11}$$

where D_{crs} is the expected distance between the center of adjacent course-connected stitches and ω_i is the user's desired course width, as indicated by the scaled atlas.

The simplicity term E_{smpl} penalizes large differences in stitch counts (n_i, n_j) between the beginning and end of a given region $(\mathbf{e}_i, \mathbf{e}_j)$ so as to encourage simple regions with minimal shaping:

$$E_{\text{smpl}}(n_i, n_j) = |n_i - n_j|^2. \tag{6.12}$$

The constraints in Equation 6.10 ensure that courses on either side of an *internal* interface (i.e., those with $\mathcal{E}_\eta^{\text{in}}, \mathcal{E}_\eta^{\text{out}} \neq \emptyset$) have the same number of stitches. The user-

specified weights λ_{crs} and λ_{smpl} control the trade-off between course accuracy and simplicity.

6.5.2 Region Sampling

After optimizing the stitch count n for each of the interface edges, we optimize the sizing along the wale and course directions, respectively, within each region. All regions can be solved in parallel.

Sizing Along the Wale Direction

To ensure that each region has the desired measurements along the wale direction, we minimize an energy penalty for the wale size accuracy across the region. In particular, we subdivide the region by tracing N isolines uniformly along its time extents and accumulating the local wale error across those while accounting for a number of additional short-rows $\mathbf{r}$ to fill the distance in between. The subdivision produces N isoline segment sets $\mathcal{S}_i \in \mathcal{U}$ for $N+1$ sub-regions $(\mathcal{S}_i, \mathcal{S}_j) \in \mathcal{A}$. We optimize for both the number of subdivisions N and the local short-rows $\mathbf{r}$ between each sub-region:

$$\underset{N,\ \mathbf{r}}{\arg\min} \sum_{(\mathcal{S}_i,\mathcal{S}_j)\in\mathcal{A}} \big(\lambda_{\text{wale}} E_{\text{wale}}(\mathcal{S}_i,\mathcal{S}_j) + \lambda_{\text{srs}} E_{\text{srs}}(\mathcal{S}_i,\mathcal{S}_j)\big), \tag{6.13}$$

where the energy term E_{wale} measures the size accuracy along the wale direction and the short-row simplicity term E_{srs} penalizes adjacent short-row densities that change too fast.

To measure E_{wale} and E_{srs}, K sample pairs $(s_{i,k}, s_{j,k})$ are uniformly distributed along $\mathcal{S}_i$ and $\mathcal{S}_j$, respectively. We let r_k be the number of *additional* short-rows between each sample pair. For efficiency, our value of K is typically much smaller than the final number of stitches on the courses. In particular, K is computed based on the curve lengths $\ell(\cdot)$ and the distance Δ_s between adjacent grid samples at the finest mesh resolution: $K = \lceil \max(\ell(\mathcal{S}_i), \ell(\mathcal{S}_j))/\Delta_s \rceil$. Then, E_{wale} and E_{srs} can be

defined in a discretized form as follows:

$$E_{\text{wale}} = \sum_{k=1}^{K} \left| \frac{\mathrm{G}(s_{i,k}, s_{j,k})}{D_{\text{wale}}} - 1 - r_k \right|^2, \quad E_{\text{srs}} = \sum_{k=1}^{K} |r_k - r_{k-1}|^2, \tag{6.14}$$

where $\mathrm{G}(s_{i,k}, s_{j,k})$ is the geodesic distance between samples $s_{i,k}$ and $s_{j,k}$, and the -1 term accounts for the implicit wale step that happens between $\mathcal{S}_i$ and $\mathcal{S}_j$.

Full courses are preferable to short-rows wherever possible, as the latter tend to increase knitting complexity. To enforce this, we require that at least one sample pair from every sub-region ends up with no intermediate short-row density– i.e., $\exists\ k,\ r_k = 0$ between each $(\mathcal{S}_i, \mathcal{S}_j)$. By optimizing Equation 6.13 subject to this constraint, we bias the solution toward full-course isolines (large N, small r_k) rather than relying on short-rows (lower N, large r_k).

Sizing Along the Course Direction

Given the best value of N, we optimize for the number of stitches m_i along each $\mathcal{S}_i \in \mathcal{U}$. This is formulated as a similar constrained IQP problem to that of Equation 6.10, with a tradeoff between course accuracy and simplicity:

$$\begin{aligned} \arg\min_{m}\ & \lambda_{\text{crs}} \sum_{\mathcal{S}_i \in \mathcal{U}} E_{\text{crs}}(m_i) + \lambda_{\text{smpl}} \sum_{(\mathcal{S}_i, \mathcal{S}_j) \in \mathcal{A}} E_{\text{smpl}}(m_i, m_j) \\ & \text{s.t. } \forall\ (\mathcal{S}_i, \mathcal{S}_j) \in \mathcal{A}, \quad \lceil m_j / F_{\max} \rceil \leq m_i \leq \lfloor m_j\ F_{\max} \rfloor. \end{aligned} \tag{6.15}$$

The constraint enforces a user-defined maximum shaping factor $F_{\max} \in (1, 2]$, which limits the rate at which stitch counts can change between adjacent courses. The bounds on $F_{\max}$ ensure that stitch counts can be instantiated into a valid stitch graph, where each stitch has at most two next wales, and at most two previous wales. Because the stitch counts n_i, n_j at the extents of each region have already been fixed by the interface sampling step, the value $F_{\max}$ also implies a minimum value of N that must be respected for a given region: $N_{\min} = \left\lceil \log_{F_{\max}} [\max(\frac{n_i}{n_j}, \frac{n_j}{n_i})] \right\rceil - 1$.

6.5.3 Stitch Connectivity

After determining the number of courses and stitches in each region, stitches are sampled uniformly along their corresponding isoline. Then, course and wale connections between them are computed to form an initial stitch graph.

Course Connectivity

Adjacent stitches on the same isoline segment set are connected first. The process is trivial for singleton sets, as the sequence of neighboring stitches is clear. For multi-segment sets, it is necessary to determine a *course path* over the segments first, to ensure that the stitch sequence is well-defined. The course path traces a consistently-oriented Eulerian path over the isoline segments, where the *orientation* is defined as the sign of the cross-product between the local displacement and the local time direction field between two subsequent locations in the same sketch. The arrows in Figure 6-8b illustrate the default positive orientation.

Connectivity across Interfaces

After connecting stitches on each course within the regions, all regions are connected together by computing a 1-1 wale assignment between the stitches on either side of an interface. Our system optimizes for the alignment between the paired stitches, while enforcing that the adjacent regions have a valid layout on the final needle bed for scheduling.

A greedy wale distribution approach is used to ensure that any circular structures sandwiched between other structures end up split evenly across both knitting beds. For the general case, our system binds N lower courses to M upper courses. We reduce this to a pair of simpler interfaces (an N-to-1 interface followed by a 1-to-M interface), both of which can be solved in a symmetric manner. Our base case is a course that needs binding to M courses, for which we greedily search the best 1-to-1 stitch alignment by

1. selecting an ordering $(\pi)_{k=1}^{M}$ of the M upper courses, then

2. sequentially searching for the best layout of the course π_k, which minimizes the geodesic distance between existing stitches after left-to-right packing of courses π_1 to π_k, and

3. using the overall best ordering π and its wale assignments.

The left-to-right packing assumes that intermediate circular courses get split evenly between front and back. If more than one intermediate course is circular, it may end up with irregular odd packing as described in Narayanan et al. [122]. To avoid this, our system enforces the optimization in Section 6.5.1 to produce even-parity stitch counts for any interface of $M > 3$ courses.

This approach enables a wide array of practical garment topologies. However, scheduling constraints can be arbitrarily complex for intricate garments, and the general case remains an open problem.

Wale Connectivity

To assign the remaining wale connections between stitches in the region interiors, we extend the *Dynamic Time Warping* strategy of Narayanan et al. [122] with a modified penalty function E_{penalty} and apply it between each pair of adjacent courses independently. The modified penalty between a source stitch Ω^{src} and a target stitch Ω^{trg} is defined as follows:

$$\begin{aligned} E_{\text{penalty}} = \lambda_{\text{dist}}\ & E_{\text{dist}}(\Omega^{\text{src}}, \Omega^{\text{trg}}) \\ & + \lambda_{\text{seam}} \sum_{\Omega \in \{\Omega^{\text{src}}, \Omega^{\text{trg}}\}} \chi(\Omega)\ E_{\text{seam}}(\Omega)\ , \end{aligned} \tag{6.16}$$

where $\chi(\cdot)$ is an indicator of the stitch's irregularity: $\chi(\Omega) = 1$ if Ω is the source of a 1-2 connection, and $\chi(\Omega) = 1$ if Ω is the target of a 2-1 connection; otherwise, $\chi(\Omega) = 0$.

The first term E_{dist} is the normalized squared geodesic distance between Ω^{src} and

Ω^{trg} on the garment manifold:

$$E_{\text{dist}}(\Omega^{\text{src}}, \Omega^{\text{trg}}) = \left(\frac{\text{G}(\Omega^{\text{src}}, \Omega^{\text{trg}})}{D_{\text{wale}}} \right)^2 . \tag{6.17}$$

The second term E_{seam} is introduced to gather irregular wale connections around the user-specified seam annotations, by penalizing irregular wales that occur far away from any seam location:

$$E_{\text{seam}}(\Omega) = \min \left(\alpha_{\text{seam}}, \frac{\Delta_{\text{seam}}(\Omega)}{D_{\text{crs}}} \right) , \tag{6.18}$$

where $\alpha_{\text{seam}} = \Delta_s \sqrt{2}$ is the interaction support of any seam annotations, Δ_s is the distance between adjacent grid samples at the finest mesh resolution, and $\Delta_{\text{seam}}(\Omega)$ is the Euclidean distance between stitch Ω and the closest seam location in its 2D chart.

After computing the wale connection, users are allowed to further edit their seam annotations interactively. To incorporate the new annotations, the wale distribution optimization described above has to be repeated. To expedites this process, our system preemptively caches the geodesic distances between each stitch pair Ω^{src} and Ω^{trg} during the initial pass of the wale connection optimization. This dramatically reduces the evaluation time for $E_{\text{dist}}(\Omega^{\text{src}}, \Omega^{\text{trg}})$. Note that $E_{\text{seam}}(\Omega)$ cannot be cached because the seam distances must be recomputed with respect to the new annotations, but the Euclidean distance evaluations are fast enough to support interactive editing.

6.5.4 Short-row Insertion

After connecting all stitches along full courses, short-row stitches are inserted according to r_k from Equation 6.13, which indicates the number of short-rows to be instantiated between the sampled pair $s_{i,k}$ and $s_{j,k}$. Our system considers each wale connection between full course stitches $(\Omega_u^{\text{src}}, \Omega_u^{\text{trg}})$, and subdivides the wale into r_u stitches, as shown in Figure 6-15. Since the number of stitch pairs generally exceeds the number of sample pairs, the density r_u between $(\Omega_u^{\text{src}}, \Omega_u^{\text{trg}})$ takes on the value r_k

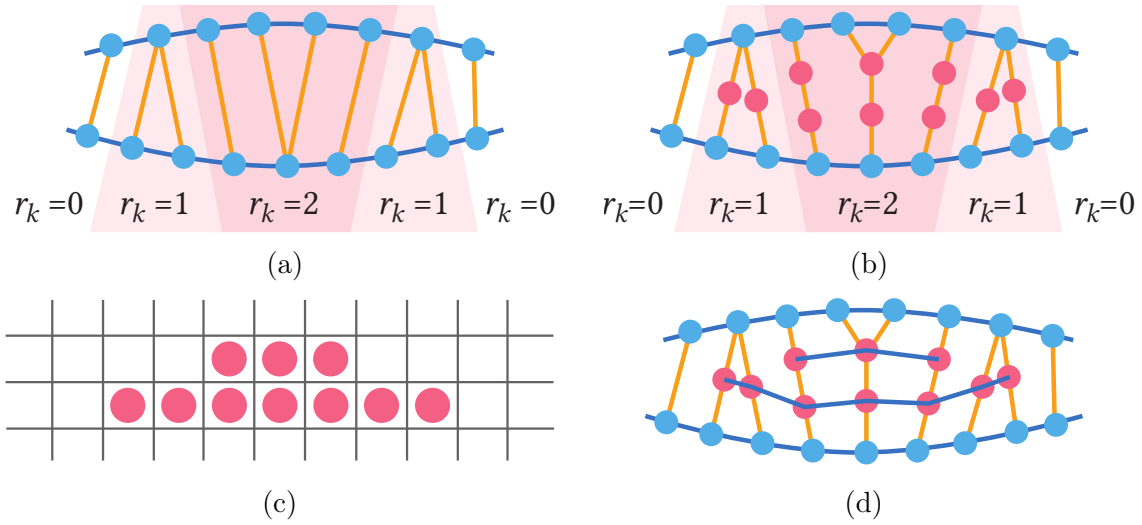


Figure 6-15: Short-row formation by splitting wales: (a) setup with initial wales and short-row densities, (b) uniform distribution of stitches over wales, (c) short-row stitch grid given user alignment (bottom), and (d) the final short-row connectivity.

from the closest sample pair $(s_{i,k}, s_{j,k})$.

For a 1-1 wale connection, the wale is subdivided into r_u uniformly-spaced stitches. The same process applies for 2-1 wale connections, except stitches are added to *both* wales in this manner. For 1-2 wales, short-row stitches are uniformly placed along the wale path between Ω^{src} and the average location $\overline{\Omega}^{\text{trg}}$ of the two target stitches. The wale connections from the source up to the upper-most short-row stitch are 1-1; only the upper-most short-row stitch has a 1-2 wale connection to the original target stitches.

After the short-row stitches have been inserted within the wales, they are connected into courses based on a user-defined vertical alignment in a virtual grid. The contiguous stitches in each row of the grid get course-connected, forming the final short-row topology. Our system supports three different vertical alignments (*bottom*, *middle*, and *top*), as illustrated in Figure 6-16.

This step concludes the generation of our stitch graph. Some implementation details related to sampling can be found in Appendix B.

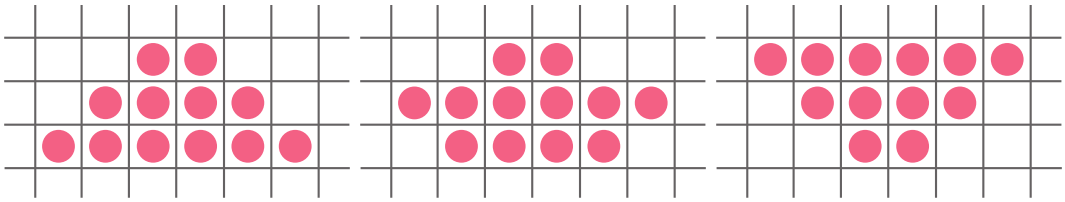


Figure 6-16: Different vertical alignments: from left to right, *bottom*, *middle* (biased towards bottom) and *top*.

6.6 Yarn Tracing

The previously sampled stitch graph is technically a *Knit Graph* as per the nomenclature of Narayanan et al. [122]. Namely, courses form disconnected rows and wales are their columns. Similarly to our yarn *tracing* step in the previous chapter (see Section 5.4), we must trace the yarn over this data structure to create the actual yarn path for which we can generate a needle schedule. We do so with the rule-based procedure of Narayanan et al. [122]. Its main idea is to trace over each stitch *twice*. This provides us with a guarantee that we can trace over short-rows with a single yarn carrier, whereas tracing only once may require the use of multiple carriers. This section briefly describes the procedure.

Figure 6-17 illustrates a tracing example across three tubular courses and a pair of short-row stitches between the last two courses. As the tracing proceeds, stitches get marked as *knit once* (●) and *knit twice* (●), whereas stitches that have yet to be knit are either *ready* (●) – when all column-wise predecessors of stitches in their row have been knit twice –, or *pending* (○) otherwise. Furthermore, the traced yarn keeps an orientation state (CCW or CW) that gets updated with the tracing rules. By default, the orientation starts counter-clockwise (CCW). The conjunction of double-tracing and the structure of our stitch graph guarantee that the first stitch being traced in any full course always starts in CCW orientation.

The *tracing rules* are attempted in order, until one is found. The tracing then restarts searching for a matching rule until all stitches have been traced twice.

R1 – Start Yarn: If no yarn is active, start a new yarn by knitting a stitch that is either ready or knit once. When choosing the starting stitch, we pick in priority:

(1) endpoint stitches (only one course neighbor), (2) stitches with a course neighbor that is knit twice, (3) stitches with no previous wale or no next wale, and (4) stitches on seams or sketch boundaries.

R2 – Next Row: If the previous step updated a set of stitches to *ready* by completing the last of their predecessors, then move to the newly readied group of stitches. In the tubular case, use the stitch that follows in the current orientation; otherwise, use the stitch directly above (next wale of) the last knit stitch, in the opposite direction. In the short-row case, further mark the base stitch as having a *next tuck*.

R3 – Continue: If the course neighbor of the last stitch in the current orientation has not been knit twice, knit it.

R4 – Tuck and Turn: Upon reaching the end of a row (i.e., no course neighbor in the current orientation), if the current stitch has not been knit twice, mark the current stitch as having a *next tuck*, switch to the opposite orientation and knit the last stitch.

R5 – End Short-row: Upon reaching the end of a row, if the current stitch has been knit twice, knit the next stitch beyond the row if any is available in one of the previous rows.

R6 – End Yarn: If no other rule applies (i.e., no row-wise or column-wise adjacent ready node exists, and all adjacent nodes have been knit twice already), then stop the current yarn.

Next tucks

A *next tuck* is a hint to tuck the next stitch in the current orientation within the final bed layout. Whether a tuck is actually generated is dependent on user settings and the safety of the actual tuck in the final knitting program (e.g., are there already multiple loops in the needle?).

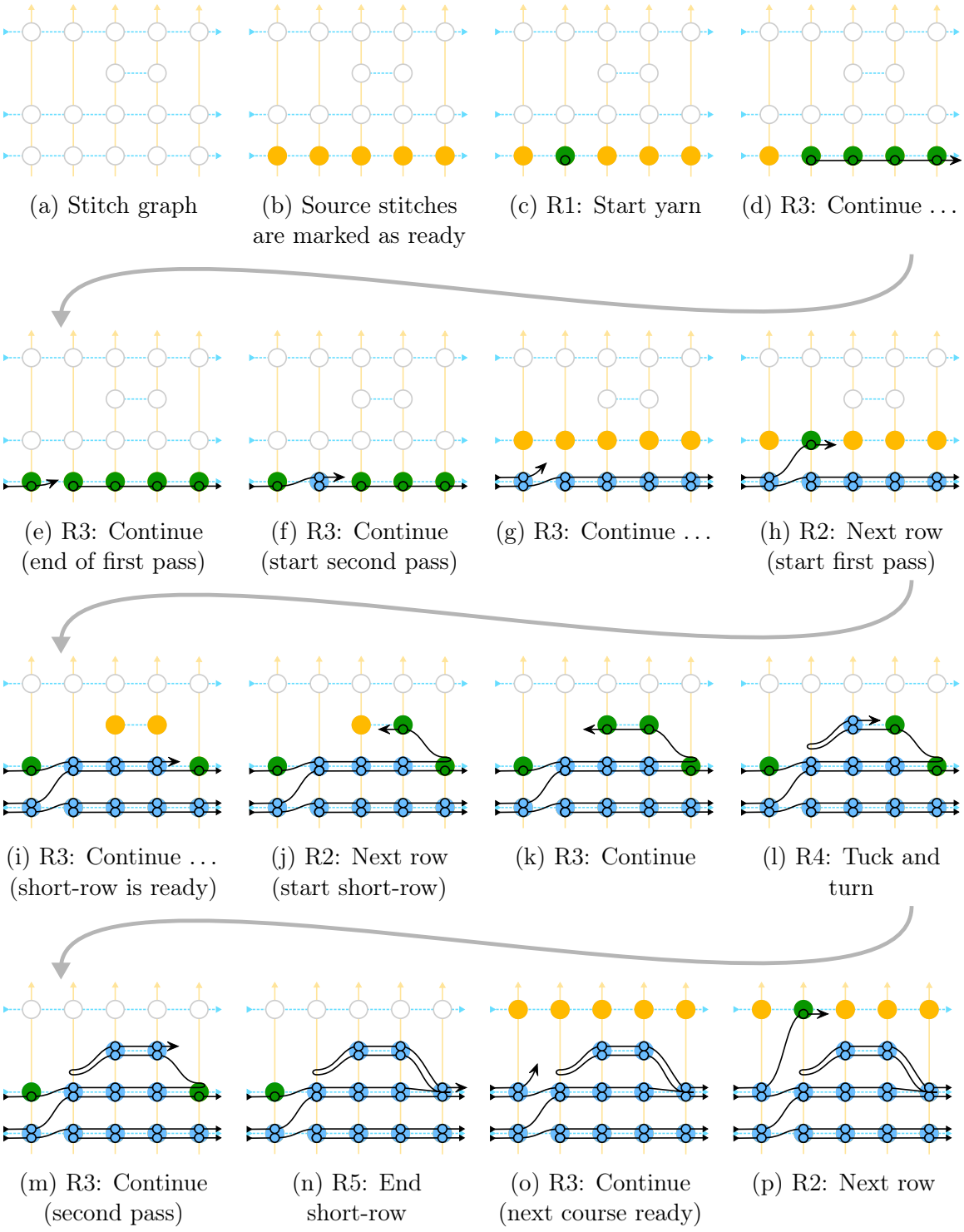


Figure 6-17: Tracing example of a tubular structure: three full courses and two short-row stitches between the last two courses. "Continue ..." corresponds to a sequence of application of rule *R3 – Continue*.

6.7 Scheduling Stitches onto Needles

The scheduling stage transforms the yarn trace into a sequence of needle bed *slices* and then optimizes the layout of the stitches on these needle bed slices.

6.7.1 Slicing

Needle bed *slices* represent local snapshots of the yarn path as it gets created on the machine. They are *local* as we only consider the yarn within its current local branch. When multiple branches are on the bed, we thus have multiple slices next to each other. Importantly, the only slices that interact with slices outside of their region are slices at region interfaces.

We represent a slice with:

- A list of stitches forming a continuous CCW-oriented cycle on the bed, and
- A mapping from stitch to state (`Expected`, `Active` or `Suspended`)

Slicing serves two purposes: (1) it provides an abstract representation of cycles of stitches on the needle bed and (2) it enables us to divide the time sequence into steps for code generation.

The latter purpose provides a few constraints that drive the slicing procedure so as to match the interpretation during code generation:

1. There should be only one active yarn within a slice,
2. Two wale-connected stitches should not appear in the same slice (because they must always happen at different times),
3. The cast-on / off needs of active stitches must be the same,
4. The active trace orientation should be the same within a slice, and
5. The number of active increases / decreases should not be too large.

The last constraint is tunable by the user: allowing more shaping at once may lead to fewer passes to the detriment of more dangerous operations. By default, we restrict ourselves to 2 increases or decreases per flat slice, and 4 per tubular slice.

Fresh Slice

Given a stitch, we create a *fresh slice* with all the knit-once stitches of its row in stitch graph before tracing. Fresh slices always contain full-course stitches and never short-row stitches. By default, the state of the stitches is set to `Expected`.

Slicing

The slicing procedure successively goes over the stitches of the yarn trace and either (1) generates a fresh new slice, (2) derives a new slice from the last one, or (3) updates the last slice. The iterated stitch gets marked as `Active` and the process goes on until all stitches of the yarn trace have been visited.

Fresh new slices (1) are generated when visiting a stitch of a new region.

New slices get derived (2) upon encountering a conflicting constraint (e.g., a stitch that is wale-connected to a stitch in the current slice, a change of trace orientation, shaping levels beyond the user threshold, etc.).

The last slice gets updated (3) when encountering a decrease stitch that merges two currently `Suspended` stitches. In such case, the two decreasing stitches are replaced by their common target stitch.

Slice Derivation

Slice derivation consists in a per-stitch and state remapping:

- `Expected` stitches stay as-is,
- *Decreasing* stitches[3] (either `Active` or `Suspended`) stay as `Suspended`, and
- Other `Active` stitches get replaced by the (0, 1 or 2) stitches of their next wale connections, as `Expected`.

[3] A stitch with a single next wale neighbor that has two previous wale connections

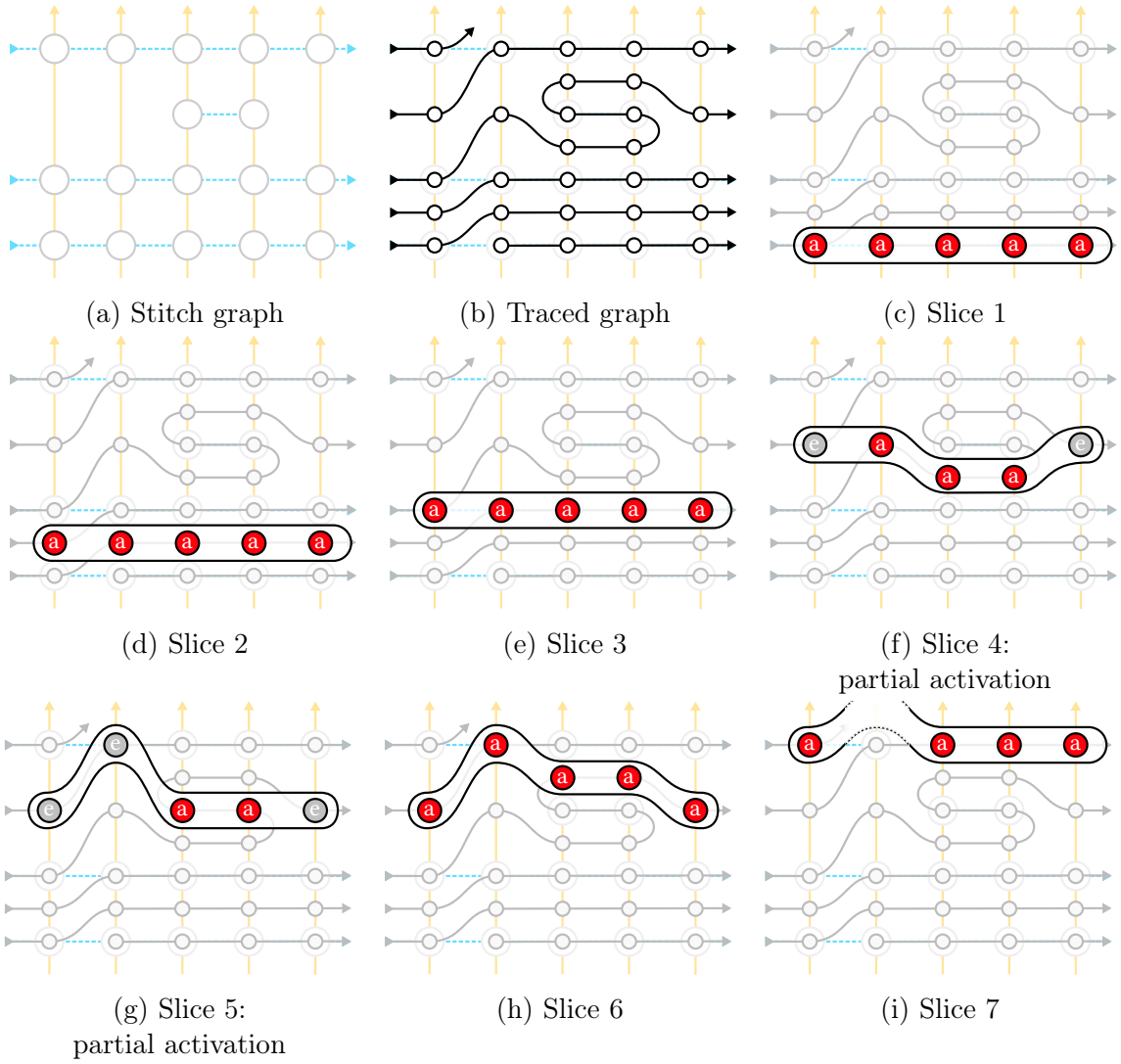


Figure 6-18: Slicing steps for the example traced in Figure 6-17. The **red stitches** are `Active` (label "a") whereas **gray stitches** are `Expected` (label "e").

Increase vs Decrease

In the *increase* case, activating a stitch directly results in two new stitches in the next slices (the next wale connections of the splitting stitch). In the *decrease* case, activating stitches that are decreasing does not always directly result in the merged stitch in the next slice. Notably, if only one decreasing stitch is activated, we must keep it on the needle bed until the resulting merged stitch gets activated. The `Suspended` state serves to keep decreasing stitches on the needle bed until the merge action effectively happens. This is notably important with short-rows as the decreasing stitches can

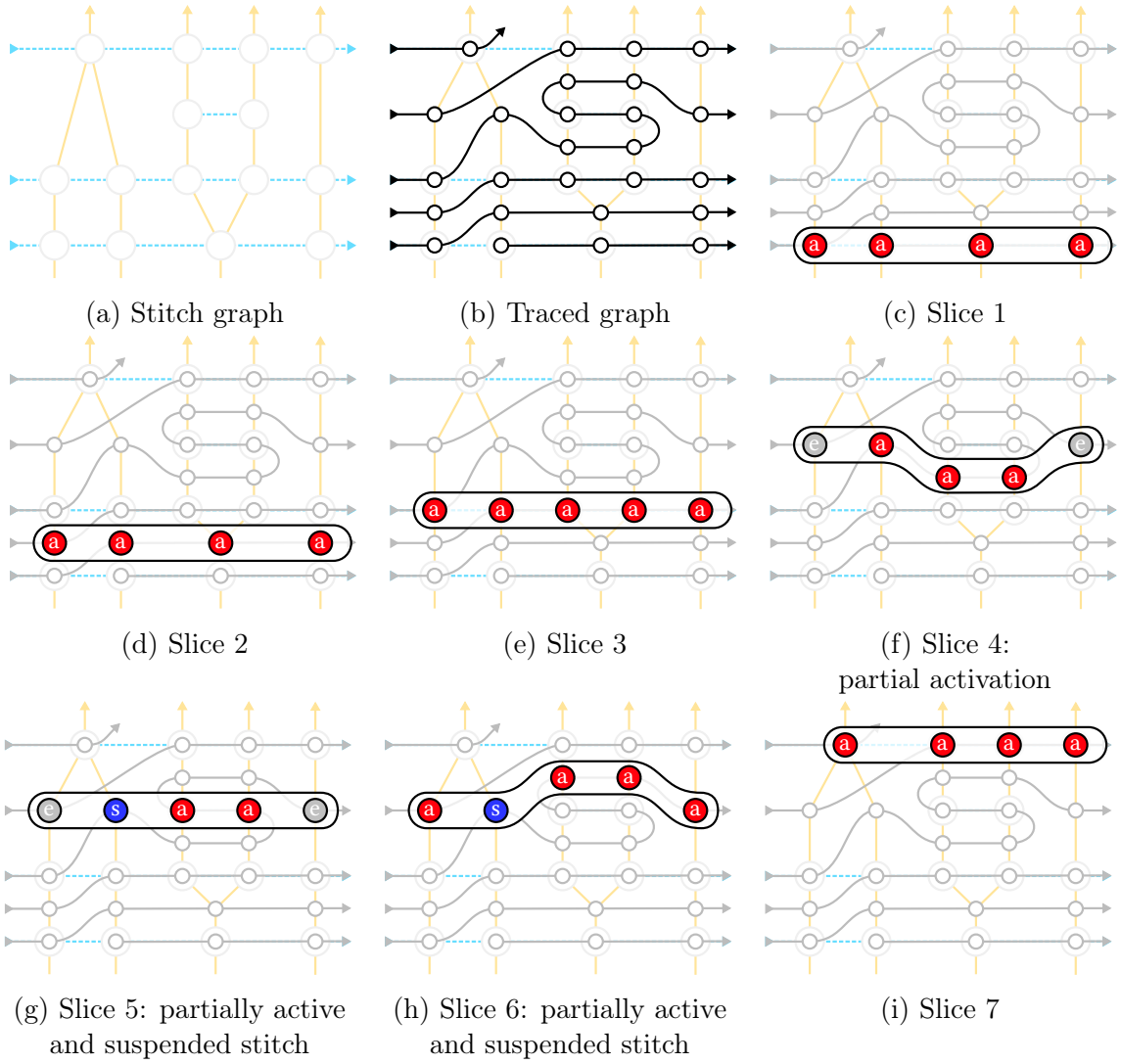


Figure 6-19: Slicing steps for the an example variation that includes shaping (increase at bottom and decrease between the two last full courses). The **red stitches** are `Active` (label "a"), the **gray stitches** are `Expected` (label "e") and the **blue stitches** are `Suspended` (label "s"). Compared to Figure 6-18, the slices differ notably in the short-row region between the last two full courses. The difference is due to the stitch decrease that keeps the decreasing stitches suspended until they effectively merge.

stay *suspended* for a few slices before actually getting merged into the decrease stitch. Figure 6-18 illustrates the basic slicing mechanism whereas Figure 6-19 highlights the impact of shaping and notably the suspended state. Notice the wale transformation between stitch graph and traced graph in Figure 6-19: the (1-2) wale pair is *after the second pass* of the splitting stitch, whereas the (2-1) pair is directly *before the first pass* of the merging stitch.

6.7.2 Layout Representations

Each slice can be laid out on the needle bed in different ways. A reasonable *slack* constraint is to require that successive stitches be reasonably close – i.e., their respective needles should have offsets $o_1 \dots o_N$ such that

$$\forall i, \quad |o_{i+1} - o_i| \leq 1. \tag{6.19}$$

In the following, we consider different types of layout parameterizations that satisfy Equation 6.19. In general, we assume the layout parameterization to deal with the relative layout of stitches. This notably excludes a global needle offset that is represented separately in the scheduling problem.

Circular Layout

Similarly to Narayanan et al. [122], we parameterize the layout of circular stitch cycles with two components:

- A *roll* parameter that corresponds to the rotation of the cycle, and
- A *nibble* parameter that decides the layout of the corners of the cycle.

Then, assuming the offset constraint of Equation 6.19, for N stitches, we have either $5N$ or $4N$ possible circular layouts for N even, respectively N odd, as illustrated in Figure 6-20. Rolls are visualized in Figure 6-21.

Nibbles: If we consider the *nibble* parameter to represent the corner defects (i.e., lack of stitch at a given corner from the default fully aligned cycle), and name the corners (FL - front-left, FR - front-right, BL - back-left, and BR - back-right), then the *even* case has 5 possible defects: none, FL+FR, BL+BR, FL+BR and FR+BL. The *odd* case has only 4 possible defects: FL, FR, BL and BR. Note that some combinations are irrelevant (e.g., FL+BL is the same as no defect with a global offset to the right, and similarly FR+BR is equivalent to no defect).

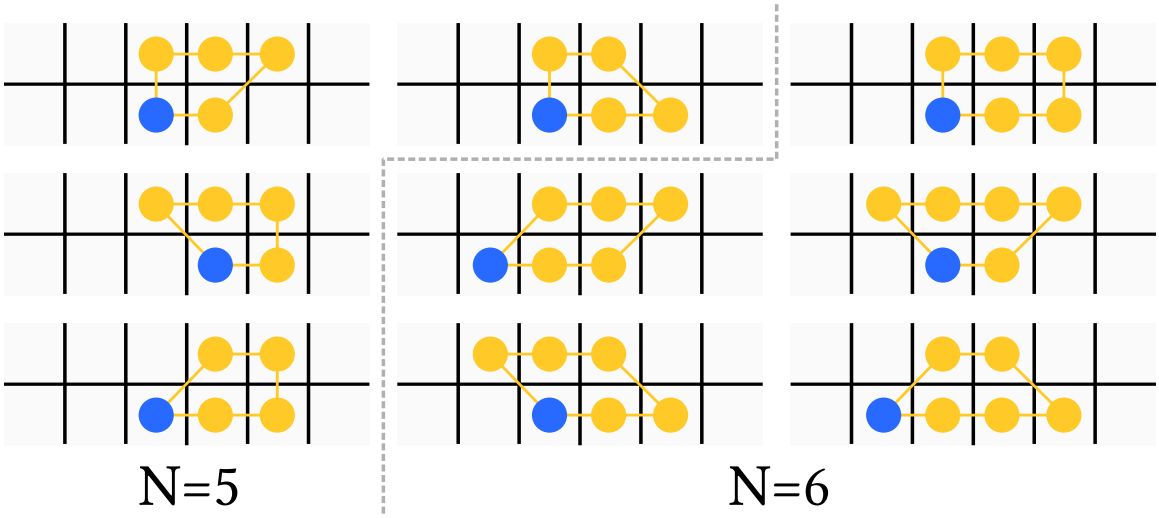


Figure 6-20: All possible circular layouts shapes for $N = 5$ and $N = 6$ without considering the roll parameter (kept constant as `roll` $= 0$). The blue circle ● represents the first stitch.

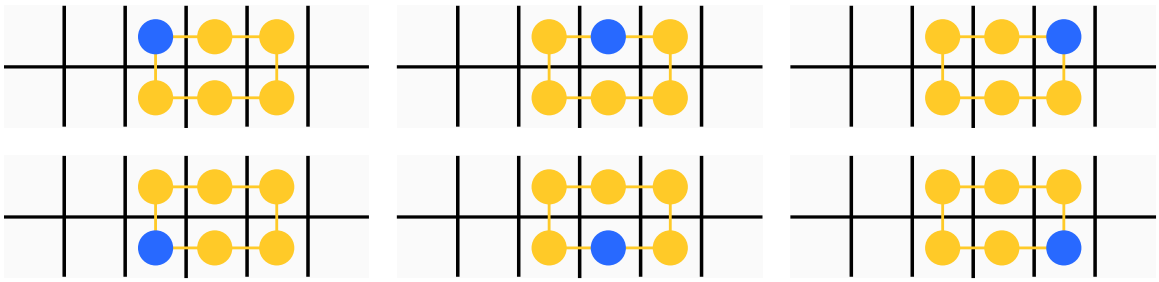


Figure 6-21: The regular (no-nibble) layout for $N = 6$ and all its roll variants. Starting at the bottom-left, going counter-clockwise, the `roll` goes from 0 to $N - 1 = 5$. All cycles have stitches in counter-clockwise order and the blue circle ● represents the first stitch.

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 one complication is that scheduling doesn't require only the layout 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

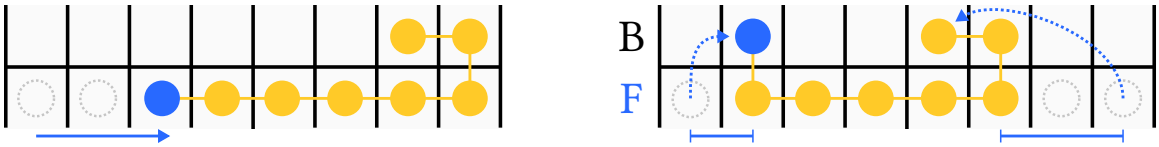


Figure 6-22: Examples of flat layouts for $N = 8$ stitches: *single-fold* with $r = 2$ (left), and *c-shaped* with $s = F$, $l = 1$, $r = 2$ and $m = 5$ (right).

the layout – i.e., the rotation doesn't matter. In the flat case, however, the rotation matters when packing cycles next to each others. This increases the search space substantially.

C-Shape Layout

The C-shaped layout 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 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

$$r = N - m - l. \tag{6.20}$$

Obviously, one can change the parameterization through that substitution as is done in Figure 6-22 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.

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

A 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.

Simplified C-Shape Layout

In practice, given the scalability issues of 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 first two 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 can add an additional parameter $a \in \{\text{left}, \text{right}, \text{both}\}$ that describes the secondary layout: either all packed on the *left*, *right* or spread evenly across *both* sides. This results in a flat layout that encompasses both the *single-fold* and simplified *c-shape* layouts.

6.7.3 Schedule Optimization

The schedule optimization is based on the same inter-layout cost and hierarchical formulation as Narayanan et al. [122]. The optimization consists in finding an optimal assignment of bed layout and bed offset for each slice. Note that the relative time between slices is fixed by the yarn tracing sequence.

Cost

The optimization selects layouts to minimize the number of operations needed to knit the target topology. We effectively try to minimize the number of loop movements that arise in the knitting program. The cost tuple that we minimize is

$$\text{Cost}(B_i, B_j) = (\text{Align}(B_i) + \text{Align}(B_j), \text{Roll}(B_i, B_j), \text{Shift}(B_i, B_j)) \tag{6.21}$$

where B_i and B_j are two bed slice layouts at adjacent time steps.

The interpretation of the terms is as follows:

- Align(B) equals 0 if the bed layout B is naturally *aligned* – i.e. no nibble for a circular layout and no folding for a flat layout –, and 1 otherwise;
- Roll(B_i, B_j) counts the number of loops that must change bed between layout B_i and B_j;
- Shift(B_i, B_j) counts the number of loops that must change needle offset between layout B_i and B_j.

The first *alignment* term favors naturally aligned layouts since those tend to lead to natural layouts. The second *roll* and third *shift* terms jointly serve to align layouts over time. The first part of the schedule optimization attempts to minimize the cost sum $\sum_{(i,j)} \text{Cost}(B_i, B_j,)$ over all related adjacent layouts (B_i, B_j).

Schedule Hierarchy

Since fresh new slices are generated upon reaching new regions, each region node of the garment gets a natural sequence of slices that is associated with it. While each of the slices of a node gets an associated layout B_i, the node gets an additional post-shaping layout B^{ps} that allows us to modify the location of the stitches on the bed after the actions of the last slice have happened. This additional degree of freedom allows for some bed transformation (without any stitch action) at the end of each node that makes the full optimization simpler.

Instead of solving the schedule optimization as a branch-and-bound over the full problem, we can now divide the problem into solving for the schedule between nodes (at their interfaces), and within each node separately.

Between-Nodes Scheduling

The global *between-node* optimization is done with branch-and-bound and seeks to find both (1) the layouts of the node interfaces, and (2) their left-to-right ordering. In the fully-tubular case, this effectively enforces that there is a planar embedding of

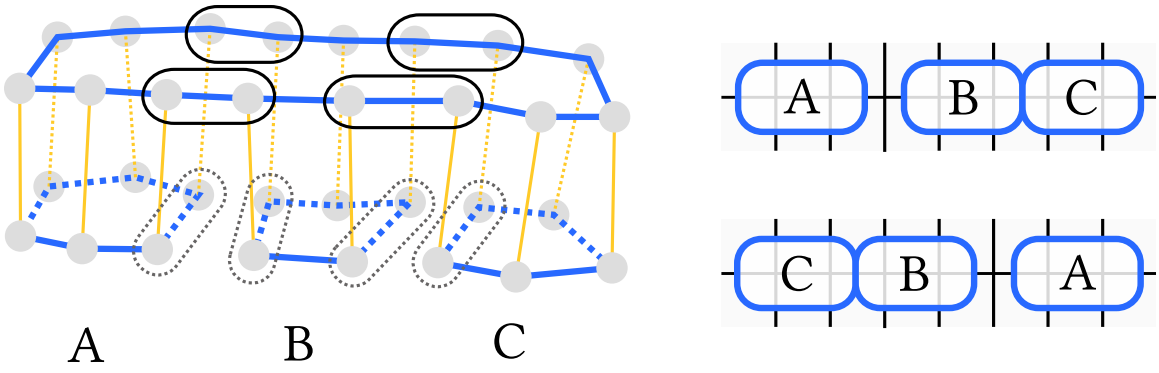


Figure 6-23: Example of bridges at a 3-1 interface. The merged cycle has 4 bridges: two for each tubular adjacency of the branches (left). The B branch has 2 bridges so that it *must* end up between both A and C. This results in only the A-B-C and C-B-A layouts being possible (right).

the region nodes as required for a valid tubular schedule [122]. When flat layouts are used, then some of the orderings may be interchangeable (for flat layouts that overlap but are across beds).

A key to efficiently search the exponential space of layouts is to quickly reject invalid pairs. When considering a given (partial) ordering of layouts and a new layout to insert, we can rule out invalid locations based on the known interaction of the current layout with neighboring layouts in the stitch graph. Notably, at a merge/split, neighboring layouts have critical adjacent stitch pairs, which we call *bridges*, illustrated in Figure 6-23. A stitch pair (Ω_i, Ω_j) in a given slice is considered a *bridge* if and only if

- Ω_i is course-connected to Ω_j and
- the wale connection of Ω_i across the interface is *not* course-connected to the corresponding wale connection of Ω_j.

The number of bridges is directly related to the number of branch separations in the region graph. This number is topology-dependent and thus does not change as the scale of the stitch layouts increase. Since a valid layout should have bridges laid out tightly, bridges serve as a direct means to reject orderings for which we know there is no possible compact bridge layout, without having to verify the entirety of the layout cost. For example, in a 3-1 circular merge interface such as in Figure 6-23, the

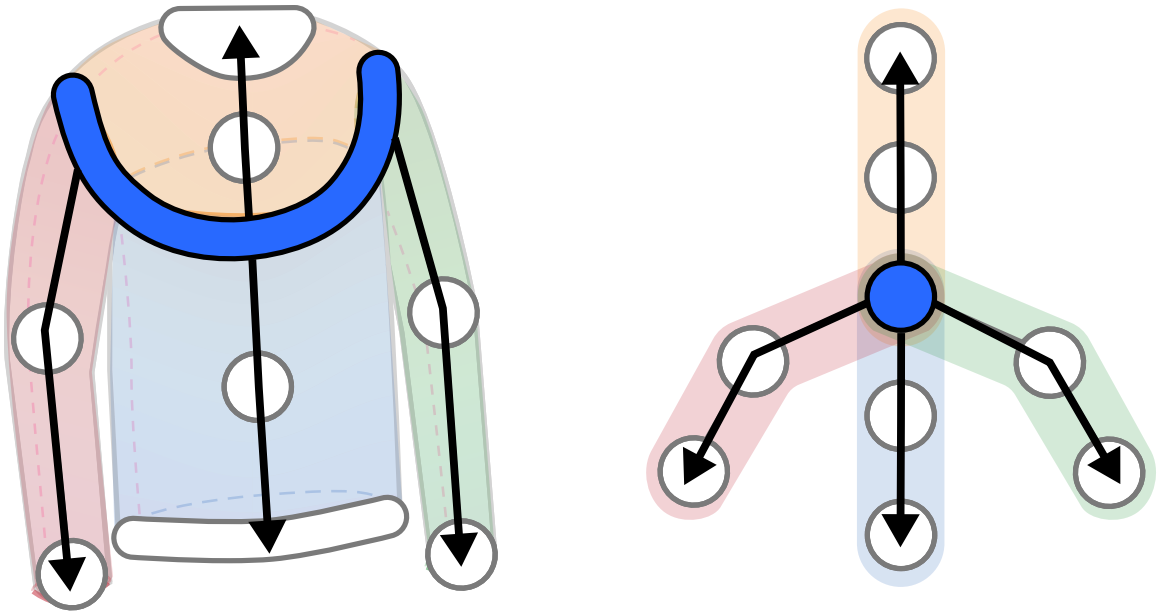


Figure 6-24: Illustration of the directionality of the greedy algorithm for each region node given the central interface having been fixed. The region nodes are only constrained from that interface and thus start their greedy layout propagation from it.

bridges can be used to pick one of the 2 possible left-to-right orderings. The remaining 4 orderings can be ruled out because their adjacency does not enable compact layouts of all bridges.

Within-Nodes Scheduling

Given the layout of the internal interfaces fixed, we optimize the layout within each node separately. This is done with a greedy, sequential strategy: (i) we pick an interface layout to start from, and then iteratively (ii) find the following layout by minimizing the cost of Equation 6.21, until all layouts in the node have been found.

If both interfaces were initially fixed (i.e., both were internal interfaces), then we start from the bottom interface. If one interface was fixed only, then we start from that one, as illustrated in Figure 6-24 with the 3-1 interface of the sweater. If no interface was fixed – i.e., we have a single node –, then we start from the bottom and consider all potential layout pairs between the two bottom layouts and pick the minimizing pair. The iterations then successively fix layouts until the whole node

is set. Note that if both interfaces were fixed, then the greedy strategy does not change the last post-shaping layout. That last layout acts as a buffer that takes care of dealing with any missing alignment. This could in practice lead to unfortunately large layout transformations.

One major shortcut to speed up common layouts concerns consecutive layouts that have the same cardinality and no shaping in between (from N stitches to N stitches through N wales). In those cases, we simply reuse the fixed layout directly since this automatically leads to an ideal transition.

Offset Optimization

After all slices have been attributed a specific layout, we build the sequence of left-to-right blocks of stitches that would eventually fill the needle bed with their fixed layouts (but currently missing offset). This notably assumes that we create *suspended* blocks after a node has been processed, by separating the stitches that are used in the next node from those that are not. Figure 6-25 visualizes a simplified schedule of a yoked sweater that contains suspended blocks for each of the nodes knitted before the last node to the 3-1 merge.

We then optimize for their offsets while taking the left-to-right ordering of nodes into account. We parameterize the offsets of a row of blocks with one left offset for the leftmost block, and one non-negative gap between each following block on its right. Furthermore, we must ensure that shaping can be properly done for the active block to its next one. The related constraints are that (1) unless the active block does not need any layout transformation to its following block, we need some spacing between it and its adjacent blocks – i.e., the corresponding gaps must be at least 1 – and (2) the stitch transformation through shaping must be possible without conflicting with the blocks on the sides – i.e., the side gaps must account for any block size increase to that of the next block.

The optimization starts by using a simple left-to-right packing. It then iteratively goes over all rows from bottom to top, and then from top to bottom, with local updates to improve the local offset and gaps of the iterated row. The iterations stop

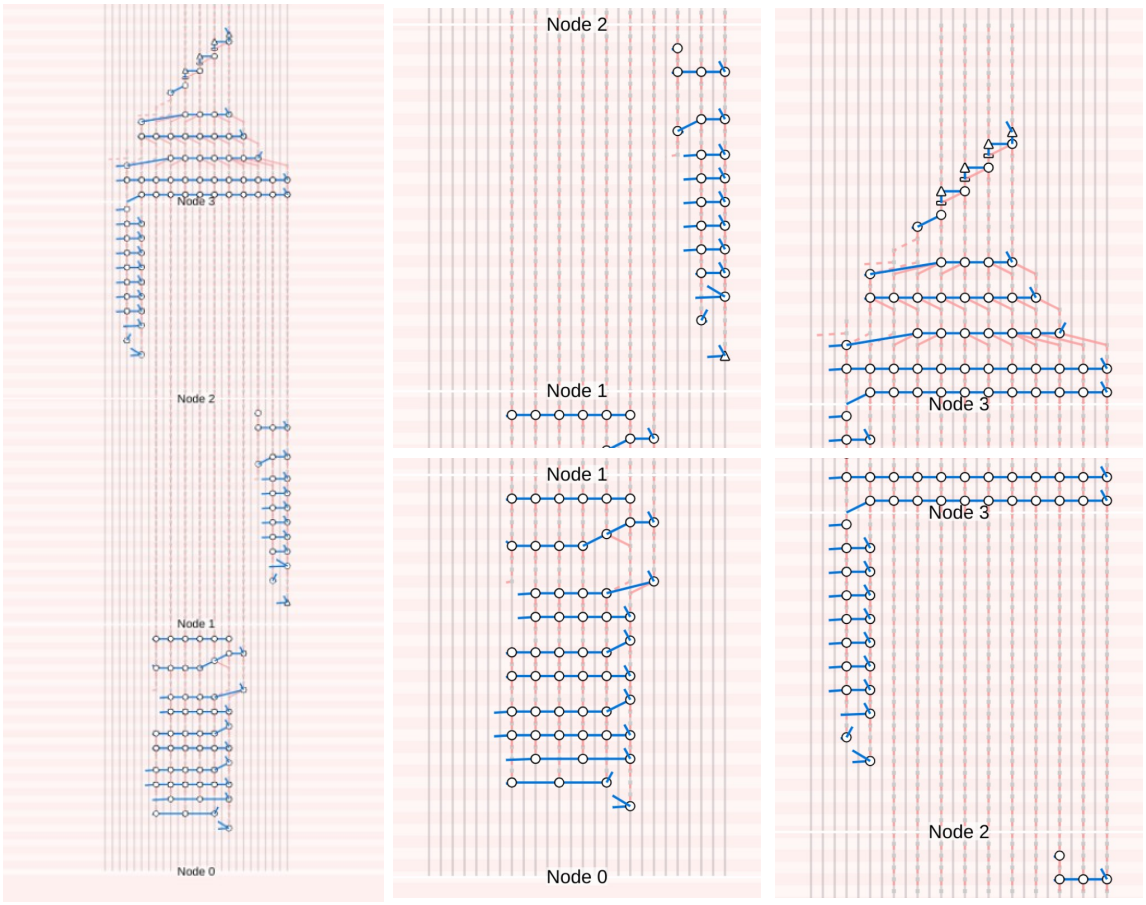


Figure 6-25: Example of simplified schedule for a yoked sweater: the full schedule (left), and closeups of each node (right). Node 0 generates a sequence of suspended block to the left of node 1. Node 1 generates yet another sequence of suspended blocks. Nodes 0, 1 and 2 merge into node 3 which takes over all suspended stitches and finishes the knitting program. This visual schedule only shows the *front* bed and further hides transfers which are implicitly visualized with the wale connectivity. It includes additional stitch blocks for yarn insertion, yarn cast-on and cast-off (i.e., zigzag section at the end).

when either a maximum number of forward and backward passes has been reached, or no udpate has improved the situation in the last forward and backward passes.

Each row update searches within some interval of the current offset and gap values of the row, and picks the ones that induce the least amount of shifts, as measured with the L2 offset error between related needles of the current row and both the previous and next rows. Needles between two rows are considered *related* if they correspond to either a *same* stitch, or two stitches that are *wale-connected*.

6.8 Code Generation

The process of transforming the nodes, their slices and needle blocks into knitting code is structured around the schedule slices. Each slice potentially triggers

- A *yarn start* pass that introduces the principal yarn carrier;
- A *cast-on* pass that casts the yarn onto the active needles of the current slice;
- Either a *cast-off* pass that casts the yarn off the active slice needles, or both an *action* pass that triggers the actions of each active stitch in the current slice, followed by a *shaping* pass that transforms the resulting stitches to match their following layout;
- An *alignment* pass that applies a sequences of transfers to move the inactive blocks on the current row to match their target location in the following row;
- A *yarn end* pass that removes the yarn of the principal yarn carrier.

After the passes of a node have been completed, an additional post-node *alignment* pass is inserted to deal with potential needle alignments that are necessary before processing the next node.

The rest of this section first details each of these passes (Section 6.8.1), and then considers the issue of the knitting gauge (Section 6.8.2). It finishes by describing two different shaping transfer algorithms (Sections 6.8.3 and 6.8.4).

6.8.1 Code Passes

We describe the different passes and provide illustrations with the schedule visualization from our sketch interface. Circles represent stitches, triangles represent tucks and flat rectangles represent explicit misses. Note that the illustrations show both front and back bed (the latter being rendered with less bright colors). We also provide example Knitout code [115] that matches the illustrations.

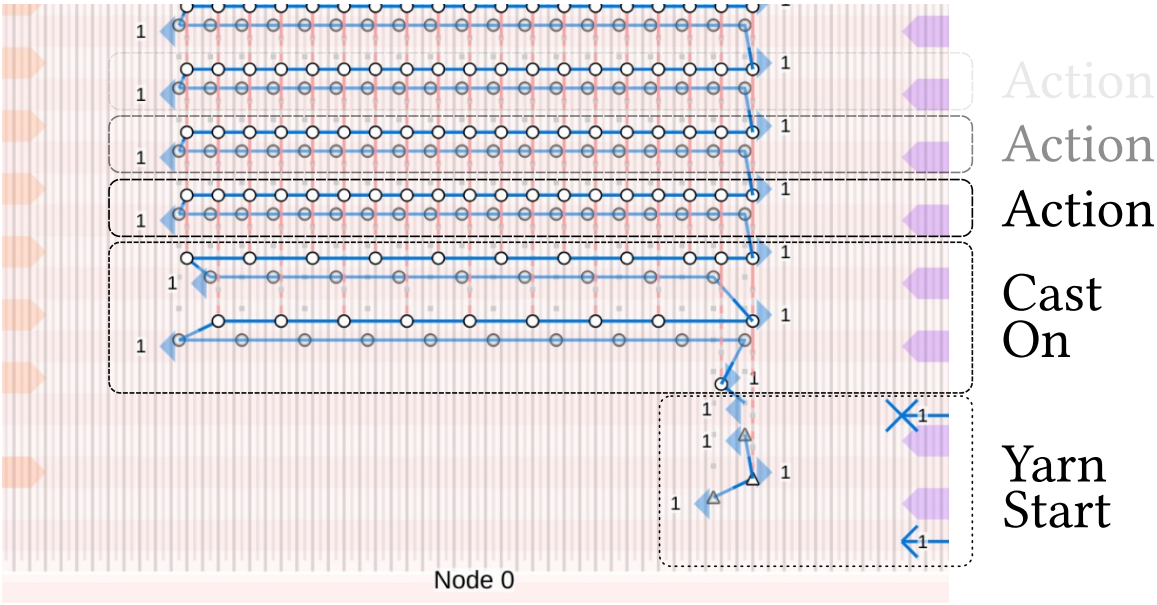


Figure 6-26: Time–needle bed layout at the introduction of a node (top), including a yarn insertion pass, a cast-on and then multiple action sequences (without visible shaping or alignments because those are empty here).

Yarn Start Pass

The yarn insertion can be done automatically, either by using the yarn insertion mechanism of the knitting machine [150] or simply drawing it and catching it with simple direction-alternating *tucks* as illustrated in Figures 6-26 and 6-27. When the yarn has caught, it can be released from the yarn holding hook device. In practice, there are machine-dependent constraints depending on the mechanism being used: e.g., the first tuck must be on the bed side opposite to the insertion unit for it to properly do insertions, and the tucks should not be in the range of the insertion mechanism before release. Furthermore, depending on the type of yarn, one may want to use more or less tucks. Lighter yarn weights may require more tucks to get proper friction and prevent it from getting pulled back by any tensioning device being used.

Cast-On Pass

Knitting a stitch on a needle can only happen if there is already a loop in the hook. Otherwise, the operation typically corresponds to a tuck since there is no yarn loop

```
3  ;Node 0
4  x-stitch-number 33 ;yarnstart
5  inhook 1
6  tuck - b33 1
7  tuck + f36 1
8  tuck - b35 1
9  releasehook 1
10 knit + f34 1 ;caston
11 knit - b35 1
12 knit - b31 1
13 knit - b27 1
14 knit - b23 1
15 knit - b19 1
16 knit - b15 1
17 knit - b11 1
18 knit - b7 1
19 knit - b3 1
```

```
43 knit + f16 1
44 knit + f20 1
45 knit + f24 1
46 knit + f28 1
47 knit + f32 1
48 x-stitch-number 6 ;action
49 knit + f34 1 ;$meta=0
50 knit + f36 1 ;$meta=1
51 knit - b35 1 ;$meta=2
52 knit - b33 1 ;$meta=3
53 knit - b31 1 ;$meta=4
54 knit - b29 1 ;$meta=5
55 knit - b27 1 ;$meta=6
56 knit - b25 1 ;$meta=7
57 knit - b23 1 ;$meta=8
58 knit - b21 1 ;$meta=9
59 knit - b19 1 ;$meta=10
```

Figure 6-27: Sections of knitting code corresponding to the introductory passes.

being knocked over the new one. Tuck loops[4] have the particularity that their location is not *stable* w.r.t. their needle. If the carriage triggers successive needles in one direction, all containing a single tuck, then the yarn can directly slide across all opened hooks and operations are likely to fail[5].

One solution to this initial instability is to only trigger knit stitches on every other needle that has a single tuck. By keeping every other needle hook closed, the yarn is caught on both sides of the needle operation and is thus more likely to succeed. Another similar option is to interlace the initial tuck passes so that the knitting pass can be done on every needles directly. The resulting yarn is not directly connected to the successive needles, which makes the initial knit stitch formation stable. This is called the *interlock* cast-on, illustrated in Figure 6-26. The cast-on procedure first alternates by casting yarn on every other needle location, and then casts it on the remaining ones only. The knitting process follows directly. In the code of Figure 6-27, note that the cast-on happens with needles offset by 4, which is every other needle in half-gauge.

[4]Sometimes referred to as *pick-up stitches* when alone in a needle hook.

[5]A related issue – although not at the heart of the cast-on problem – is that one cannot tuck on a needle and then directly tuck it again in the opposite direction. The second tuck effectively undoes the first one.

Action Pass

The action pass goes over each active stitch of the current slice and triggers the associated action with each of these. By default, these actions include *knit*, *kickback* (i.e., a *knit* in the opposite direction), *miss* and *split*, which are needed for different types of stitch increases. However, we explicitly make the action pass modular to support additional *user actions* associated with each stitch. Examples of different user actions and passes are developed in Section 6.9.

Given a slice, we get the action associated with each active stitch in sequence, and check whether the sequence needs to be *divided*. This is the case for actions that happen on the opposite bed (e.g., purls). In such case, we divide the sequence into the largest sub-sequences that use needles from the same bed side.

Each stitch action can be made of a sequence of different steps motivated by the "pre-main-post" stitch face actions from Narayanan et al. [123]:

- *Pre*-steps for introducing any new local yarn, or transferring stitches to their necessary bed (e.g. purl patterns that use the opposite bed);
- *Main* steps that trigger the main knitting actions; and
- *Post*-steps for finishing transfers and potentially remove local yarns.

The action pass goes over each sub-sequence in order, starting with each of the pre-steps (covering all stitches of a sub-sequence, for each step index at a time), then continues with all main steps, and further post-steps. Figure 6-28 illustrates a course using purl continuously. This gets divided per bed-side, and we get three steps per action block: a pre-transfer, the main knit action, and the post-transfer back to the initial bed. Figure 6-29 illustrates a small 1×1 rib structure over two courses, which highlights the alternating process between front and back stitches for each pass.

The *main* step that triggers the base yarn also potentially adds tucks on its sides depending on the tracing result (e.g., for short-rows, or potential intarsia tucks) and the user settings.

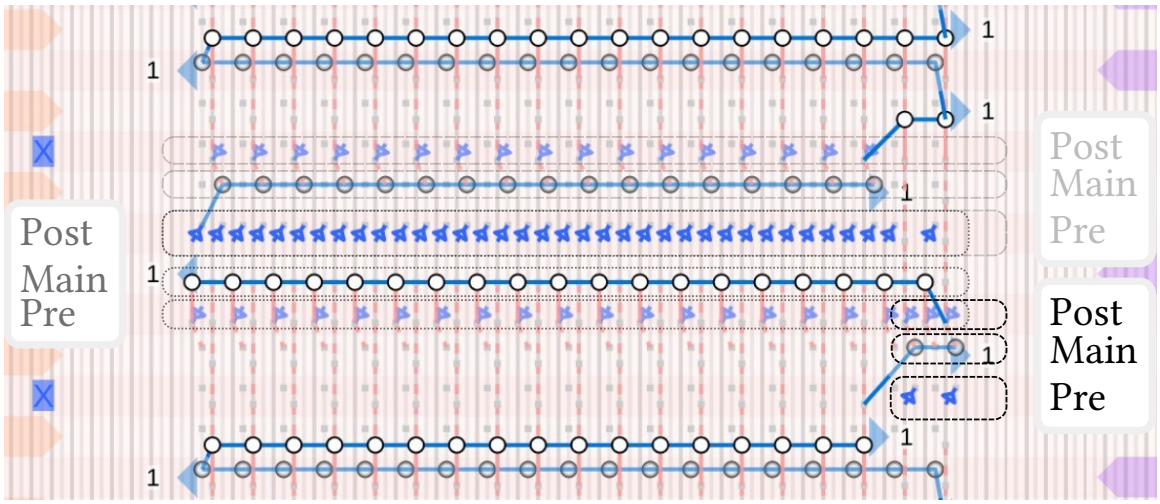


Figure 6-28: Time–needle bed layout for two consecutive purl courses. Note that the layout is slightly rotated so each slice gets split into three: one for the small back bed on the right (two stitches), one for the full front bed, and one for the remaining back bed. The post-transfer and pre-transfer steps across beds automatically get consolidated into a single pass.

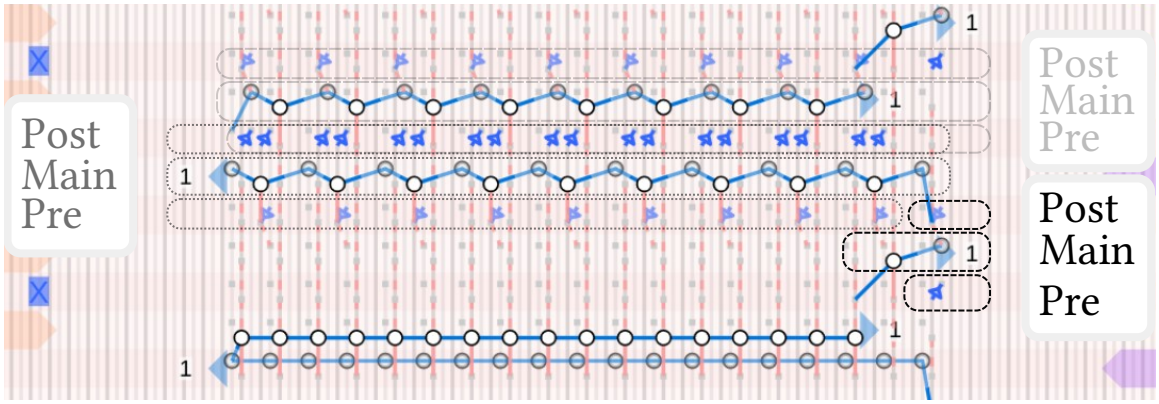


Figure 6-29: Time–needle bed layout for one 1×1 rib course.

Shaping Pass

The shaping pass computes a sequence of needle transfer operations that transform the current layout into the next one. The needle transfer is supposed to be always possible because the offset optimization took into account this shaping step and introduced necessary gaps for complex layout transformations that require space for rotating stitches. We use one of two different transfer algorithms described in Sections 6.8.3 and 6.8.4, illustrated with a simple decrease shaping example in Figure 6-31.

```
124 knit + f28 1 ;$meta=73
125 knit + f30 1 ;$meta=74
126 knit + f32 1 ;$meta=75
127 x-stitch-number 0 ;shaping
128 x-stitch-number 6 ;action
129 xfer f34 b34
130 xfer f36 b36
131 knit + b34 1 ;$meta=76
132 knit + b36 1 ;$meta=77
133 xfer b34 f34
134 xfer b36 f36
135 xfer b35 f35
136 xfer b33 f33
137 xfer b31 f31
138 xfer b29 f29
139 xfer b27 f27
140 xfer b25 f25
```

```
128 x-stitch-number 6 ;action
129 xfer f36 b36
130 knit + f34 1 ;$meta=76
131 knit + b36 1 ;$meta=77
132 xfer b36 f36
133 xfer b33 f33
134 xfer b29 f29
135 xfer b25 f25
136 xfer b21 f21
137 xfer b17 f17
138 xfer b13 f13
139 xfer b9 f9
140 xfer b5 f5
141 xfer b1 f1
142 knit - b35 1 ;$meta=78
143 knit - f33 1 ;$meta=79
144 knit - b31 1 ;$meta=80
```

Figure 6-30: Sections of knitting code corresponding to the purl and rib sections.

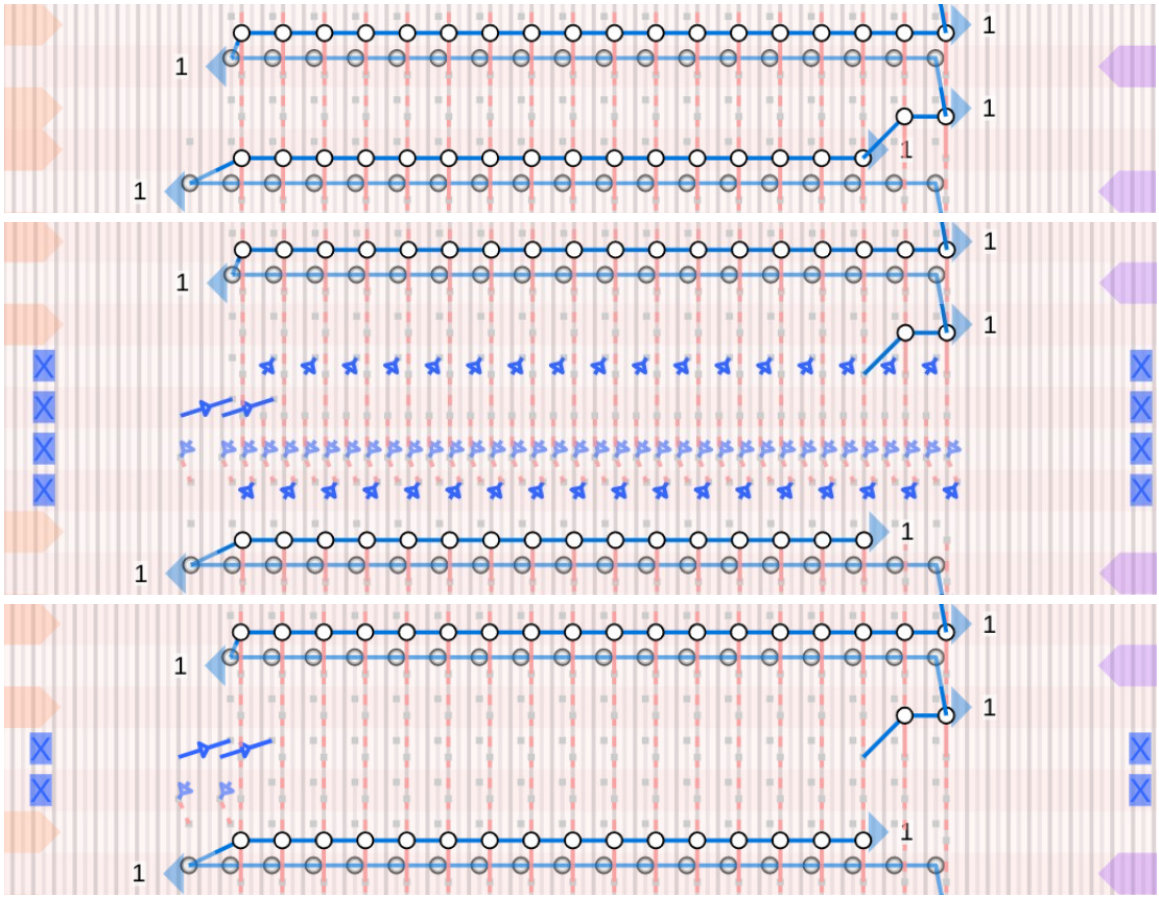


Figure 6-31: The layout decrease with implicit shaping (top), the corresponding transfer passes with the *Collapse-Shift-Expand* algorithm (middle) and with the *Rotate-Shift* algorithm (bottom).

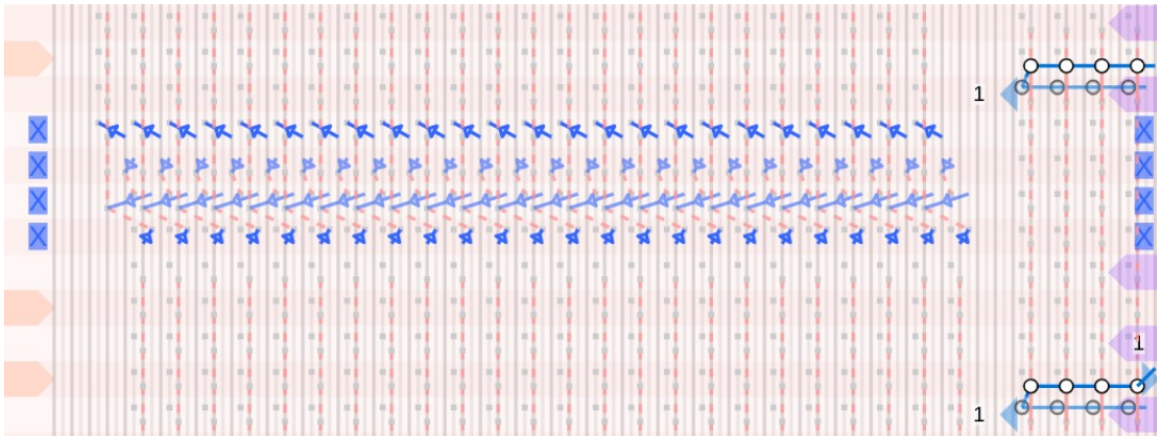


Figure 6-32: An alignment pass shifting a suspended block by two needles to its left so as to make space for future actions of the active block on the right.

Alignment Pass

The alignment pass produces a sequence of transfer operations that shift entire blocks of stitches on the bed. The algorithm is less involved than the more general shaping ones: (1) we first measure the offset that each current stitch should translate by, (2) we then alternate between applying grouped lateral moves on the front and back beds with a limited distance (i.e., between -2 and $+2$ needles), (3) we update the necessary shift of the stitches that have moved given their last translation. The process continue until all stitches have reached their location. Lateral moves are done by transferring all moving stitches to the opposite bed, then racking (from -2 to $+2$ depending on the target moves), and transferring back to the original bed. The procedure eventually terminates since any moving stitch in an iteration ends up decreasing its distance.

Cast-Off Pass

The cast-off pass deals with active slices that have no next stitches: i.e., the stitches are terminal and must be closed. Dropping stitches from the needle hook is not a good solution for the finishing of any knit topology because the loops in the hook are not connected to any other loop and would thus unravel if left unattended. Casting off deals with automatic closure of those loops. The typical procedure sequentially creates a stitch in the sequence to close, moves it to its next course-neighbor, and goes

```
338 x-stitch-number 24 ;castoff
339 knit + f2 1
340 miss - f2 1 ;castoff kickback
341 xfer f2 b2
342 rack 2
343 xfer b2 f4
344 rack 0
345 knit + f4 1
346 miss - f4 1 ;castoff kickback
347 xfer f4 b4
348 rack 2
349 xfer b4 f6
350 rack 0
351 knit + f6 1
352 miss - f6 1 ;castoff kickback
353 xfer f6 b6
354 rack 2
```

```
338 x-stitch-number 24 ;castoff
339 knit + f2 1
340 miss - f2 1 ;castoff kickback
341 xfer f2 b2
342 rack 2
343 xfer b2 f4
344 rack 0
345 tuck + f2 1
346 knit + f4 1
347 miss - f4 1 ;castoff kickback
348 xfer f4 b4
349 rack 2
350 xfer b4 f6
351 rack 0
352 tuck + f4 1
353 knit + f6 1
354 miss - f6 1 ;castoff kickback
```

Figure 6-33: Sections of knitting code corresponding to the default castoff pass (left) and with pick-up stitch (right).

on from that neighbor, as illustrated with the code of Figure 6-33. This effectively links each stitch to its neighbor, except for the last one that requires some manual closure outside of the machine (similarly to the first stitch during cast-on). We further provide the option to add an additional pick-up stitch inserted at each closing stitch, which makes the closing course looser.

Yarn End Pass

Upon ending the yarn, a dedicated pass triggers the yarn removal, typically done automatically with the yarn insertion unit to cut the yarn locally before bringing it back to the yarn holding hook. If this pass happens after a cast-off pass, then the yarn is closed and we add a small *tail* of yarn to simplify the manual closing of the last stitch as having more yarn is helpful in that scenario.

6.8.2 Half-Gauge vs Full-Gauge

Similarly to Narayanan et al. [122], the scheduling is done without taking into account whether the output is meant to be knit in full gauge or half gauge: i.e., as if done in full gauge. However, their implementation relies on the *Collapse-Shift-Expand* algorithm from McCann et al. [116] to do shaping. Unfortunately, that algorithm

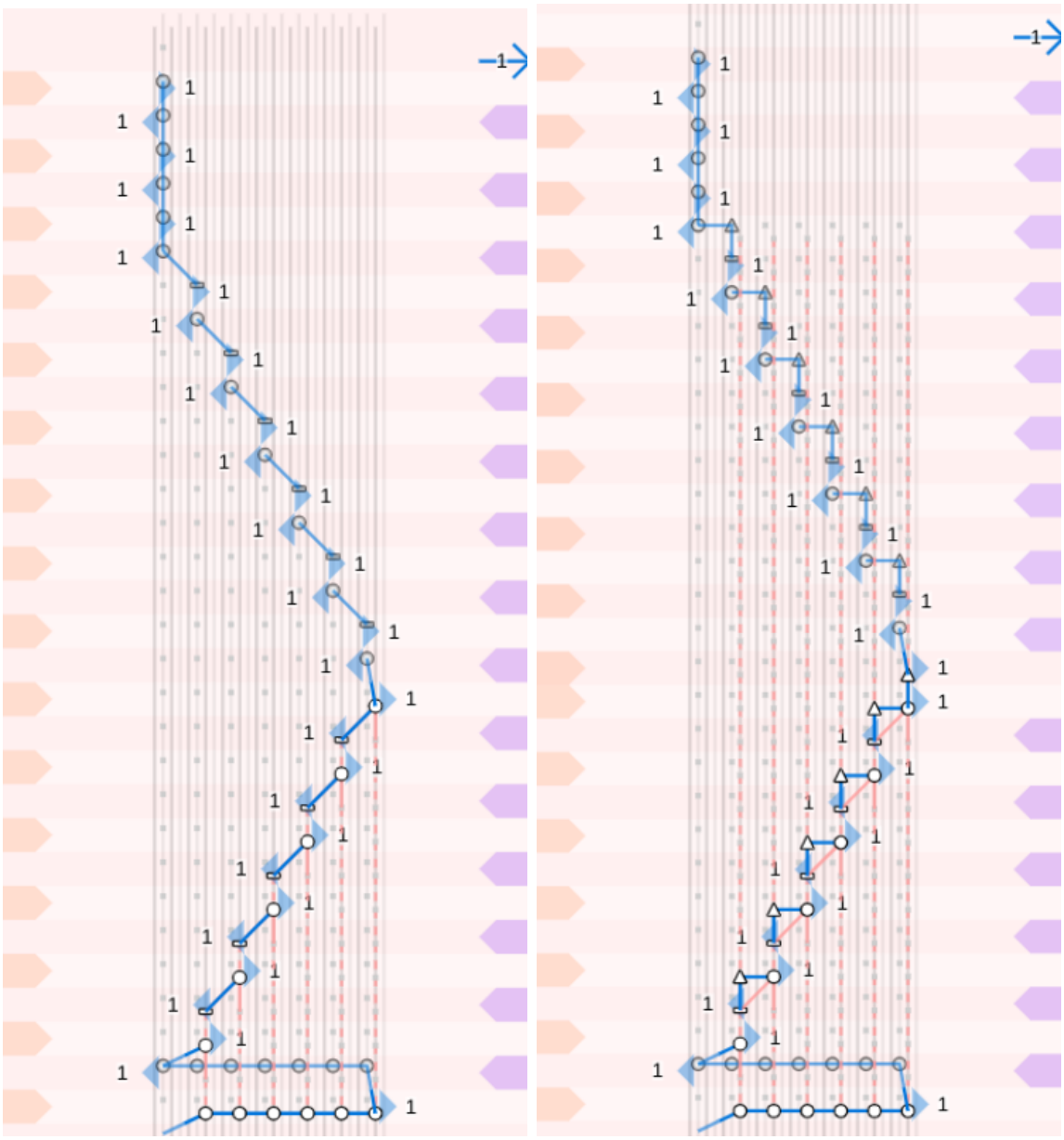


Figure 6-34: Two yarn ending procedures including the castoff pass and the yarn removal with added tail for easy manual closing. The left variant is the simplest bind-off procedure whereas the right one adds additional pick-up stitches to loosen the ending edge of yarn.

makes assumptions that prevent it from working for full-gauge knitting if the carrier ends up being inside of the shaping transfer region. In such scenario, the transfer algorithm should explicitly take care of moving the carrier around, whereas Narayanan et al. [122] rely on the implicit carrier movements taken care of by their basic Knitout compiler [115]. Implicit carrier moves greatly simplify the algorithm implementation,

but they are unfortunately not possible during full-gauge shaping transfers because the machine must use sliders. To our knowledge, the SWG091N2 machine does not support carrier movements while stitches are on needle sliders. This is mainly a hardware limitation and it can be bypassed in software by using a different algorithm that explicitly deals with explicit carrier movements, and which we call *Rotate-Shift*.

By default, we assume a half-gauge output since it is looser, supports tubular purl operations and generates larger pieces for a smaller computational cost (see Section 5.3.4). In such case, we rely on the *Collapse-Shift-Expand* algorithm for shaping transfers, and the needle locations are expanded to half-gauge after the transfers have been computed [116]. If the user requests a full-gauge version of the garment, our shaping passes are instead switched to rely on the *Rotate-Shift* algorithm.

6.8.3 Shaping with Collapse-Shift-Expand

The *Collapse-Shift-Expand* (CSE) algorithm [116] provides a solution to the *transfer planning problem*. Namely, given a cycle of stitches on the needle bed and a set of constraints, the optimization problem seeks to find a sequence of transfer and racking operations that transforms the stitch cycle from its initial configuration into a new one while satisfying the given constraints. The constraints typically include: (1) a free needle range, (2) a set of slack values describing the maximum allowed distance between successive stitches, and (3) a maximum racking offset.

The CSE algorithm works by sequentially applying three types of transformations:

- **Collapse** transfers all stitches from one bed side to the other side – which is always possible thanks to the slider locations; while doing so, it will typically focus on either the frontmost or rightmost location on the initial bed and ensure the remaining slack constraints;
- **Shift** optionally applies a bed shift while transferring all stitches back to the starting bed (and the associated slider locations) without changing the relative locations;
- **Expand** then distributes the stitches on the sliders back to the opposite bed.

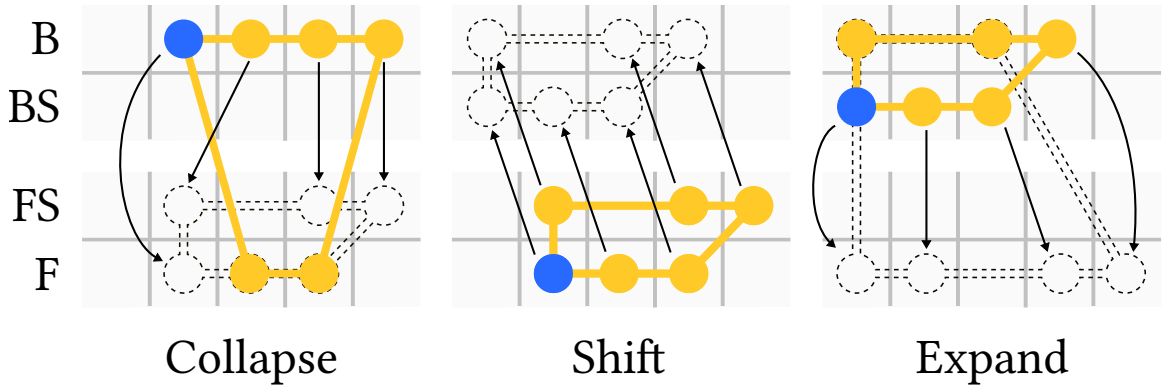


Figure 6-35: Example of cycle transformation with the *Collapse-Shift-Expand* procedure. B / BS / FS / F refer to the needle bed types: back, back-sliders, front-sliders, front.

Figure 6-35 shows an example of CSE round that enables a rotation of a stitch cycle.

The optimization part consists in deciding what moves to select during the collapse and expand stages, as well as any useful shift to apply in between. This is done with a recursive penalty function that describes the total circulation cost of stitches around the bed until reaching their target, given minimum and maximum free needles.

What prevents the application of this strategy to full-gauge knitting is the fact that the procedure heavily relies on the storage of stitches on the slider locations. In practice, this is not an issue with half-gauge knitting since those locations get translated as needle hooks, and thus no slider is necessarily involved.

6.8.4 Shaping with Rotate-Shift

Our *Rotate-Shift* (RS) strategy for full-gauge knitting is a simpler transfer procedure, specifically dedicated to the *shaping* pass. It consists in composing two intuitive, basic operations needed for shaping a stitch cycle, illustrated in Figure 6-36:

- **Rotate** collapses a corner of the current cycle to improve a *winding number*–based penalty, possibly using a few instances of *Shift* to enable the collapse;
- **Shift** applies lateral moves to the current layout to move current stitches left or right while taking into account the available free needle range, as well as carrier locations.

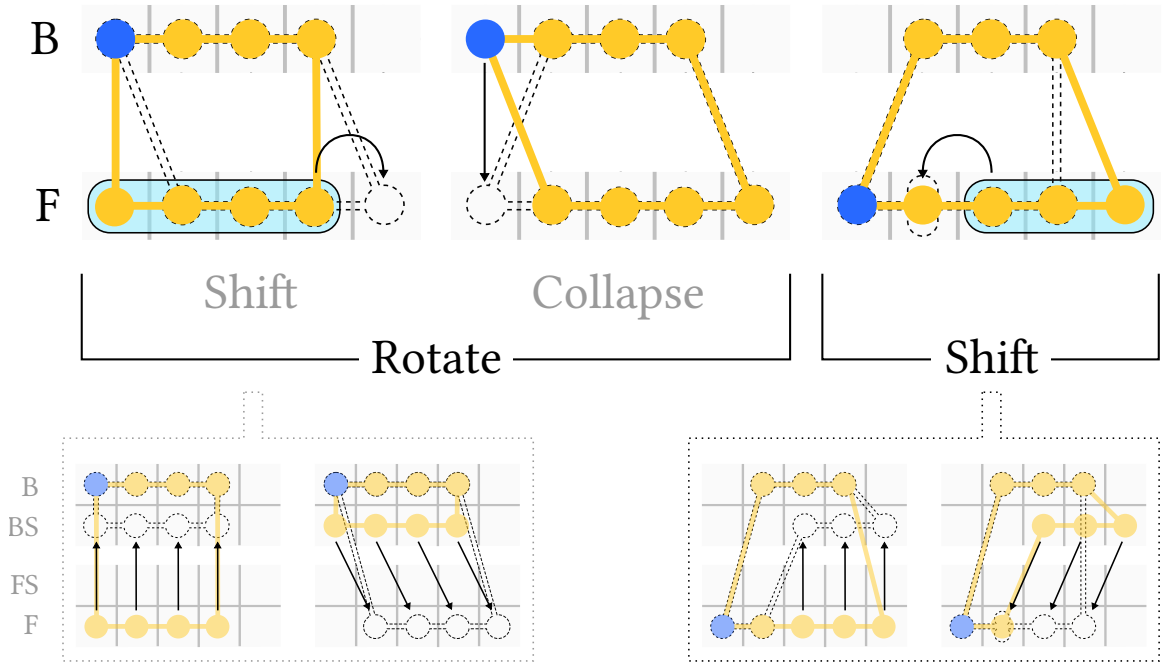


Figure 6-36: Example of cycle transformation with the *Rotate-Shift* procedure. The *Rotate* step uses *Shift* to prepare the bed, before apply a *Collapse* operation. The last *Shift* step deals with increase/decrease shaping. In this example, the cycle rotates once and then applies a stitch decrease. Both shifts are shown as group moves (top section) and as developed two-step transfers (bottom section).

To easily deal with carrier movements, our main simplification is to assume that the slack of each stitch is *at least* 2. This is a necessary relaxation of the algorithm to allow us to move stitches by themselves locally, which is necessary when a carrier is within the cycle and acts as a *barrier* that stitches need to cross one-by-one.

Rotate Procedure

We choose the rotation to pick (which corner to collapse) based on which one decreases the absolute sum of minimum winding numbers the most. We use the winding numbers defined as in the work of McCann et al. [116] – i.e. as the number of counter-clockwise crossing across beds for a stitch to reach its target location.

The corner collapsing basically consists in *shifting* the bed to have the needle opposite to the stitch corner free, and then transferring that stitch to the other bed.

When all stitches are on their target bed, we apply a last *shift* step to distribute them to their target locations, effectively applying stitch increases and decreases.

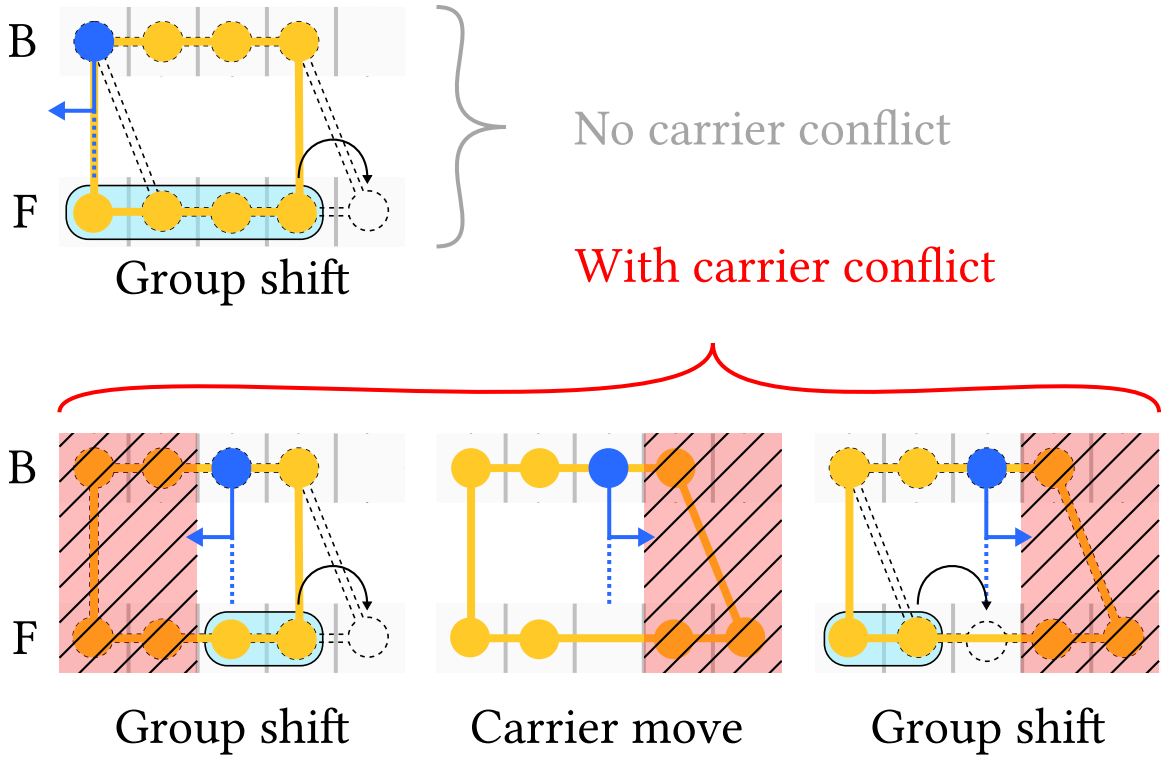


Figure 6-37: Example of simple lateral shift similar to the first one in Figure 6-36 where the four front stitches are moved the right by one needle. The difference is that the main stitch to which the yarn is attached is inside the cycle, marked in *blue*. The arrow below denotes the side of the carrier relative to that stitch. It starts to its left, which invalidates moves for stitches on the left of the barrier. Thus, the first shift step moves the two stitches from the barrier to the right (left). Then, the carrier is switched to the other side of the barrier (center). Finally, the last two stitches on the left can now move by one stitch to the right.

Shift Procedure

The main complication happens during this *shift procedure* in which we must provide an explicit treatment of carrier moves when they conflict with needle transfers. The base idea is to split the current bed into four quadrants: front stitches going left, right, and similarly back stitches going left, or right. The shift procedure then moves stitches within each of these quadrants so that they eventually reach their shift location. A quadrant shift must however deal with potential *barrier* locations induced by the last stitches knitted by an active carrier, as illustrated in Figure 6-37.

When a stitch forming a barrier is moving, it splits its moving group into two sub-groups that move separately, possibly in small steps to enforce slack constraints.

When other stitches must cross a barrier (i.e., on the other bed), they do so by (1) approaching up to the barrier, with the corresponding carrier being on the shift direction away from the barrier; then (2) the barrier carrier is switched to the other side, (3) which lets the stitches at the barrier needle and beyond go through by at least one stitch. The procedure is continued, until all stitches have reached their target shift location.

The stitch-by-stitch traversal of the barrier is potentially slow but its main limitation is the fact that stitches have to move one by one to cross the barrier. This requires a slack of *at least* 2 needles since the needle at the barrier offset must become empty to let the next stitch pass. While the available slack decides on how far the group ahead of the barrier can move, there is typically no advantage in moving more than one needle away as long as the barrier is present since its crossing is done one stitch at a time.

Comparison to CSE

On the positive side, RS deals explicitly with carrier conflicts, which enables its use for shaping during full-gauge knitting. In contrast to CSE, stitch merging only happens at the end of the procedure. This explicitly reduces the number of *overlapped loop transfer* operations and may thus be safer when knitting.

On the downside, the procedure is based on a greedy strategy that collapses stitches one by one. While the local shifts are grouped by quadrant, each of these are done separately. Thus, we expect the procedure to be potentially a lot slower for complex shaping operations. Although, for simple shaping, it can be as effective, if not better than CSE such as in Figure 6-31 where a single step is sufficient for a stitch decrease. Another potential issue with the RS procedure is that it requires a larger available slack between stitches and may wear the yarn faster than CSE. This is however to put in the perspective of CSE being effectively used for half-gauge knitting, and thus with double the original slack.

6.9 Layer-based Customization

All the algorithms and procedures of this chapter, up to now, mainly deal with the shape of the garment. Yet, the patterns and colorwork that may happen on a sketch are critical operations. Here, we develop a strategy based on *layers* to integrate typical garment customization such as stitch patterns and colorwork on top of sketches.

We start by describing the general method we use to program patterns (Section 6.9.1). We then consider whether a *layer* should live in sketch space or stitch space (Section 6.9.2). The rest starts with a generic layer interaction model (Section 6.9.3) before detailing different layer implementations: stitch patterns, multi-yarn patterns, Jacquard patterns and intarsia layers.

6.9.1 User Stitch Programs

Similarly to the patterning done with the parametric skeleton graph in Section 5.3, we let the user specify user actions associated with each stitch. The underlying graph is the *traced graph*, which we augment with the original *pre-tracing* stitch graph information – notably for the course and node information.

The per-stitch actions are represented by a set of passes similar to those of Narayanan et al. [123]. Each pass consists in one or a sequence of functions that take the local context as input (e.g., needle, direction, bed and machine states) and output *Knitout* code [115]. Listing 6.1 illustrates the definition of the *purl* pattern action, as well as different actions related to fair-isle colorwork. Note that the `front`, `back` and `tuck` actions have multiple stages for their *main* pass. Listings 6.2 and 6.3 illustrate user programs to apply stitch patterns and fair-isle colorwork respectively.

One of the major differences with the patterning from Section 5.3 lies in how we generate local grid structures. In the skeleton graph, the primitives provide an inherent structure that allows for regular patterning automatically. Since we cannot rely on a fixed primitive-based structure, we instead rely on a new method for inserting the structure locally: namely, we use two new constructions `waleGrid` and `stitchGrid` to produce some form of regularity that is instantiated from a stitch selection.

```
// basic purl
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
});
// main colorwork yarn actions
const cs2 = ['2'];
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
});
// floating tuck connections
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)
});
```

Listing 6.1: Section of user program defining user actions

`waleGrid(waleRange, steps)` extends a current stitch selection by extruding it in the wale direction by a specified number of stitches. The initial selection is considered

```
1  const twoInches = prog.lengthToWaleStitches('2 in');
2
3  // neck edge
4  const neckLen = twoInches + twoInches % 4;
5  const neckEdge = prog.filter(s => s.countNextWales() === 0);
6  const neck = neckEdge.waleGrid(0:end, -neckLen);
7  neck.tile(0b10, 2).prog(purl);
8  // sub-ribs
9  for(let i = 1, ci = 0, pass = 1; i < neckLen - 1; ++i){
10   const crs = prog.courses(-2 - ci).pass(pass);
11   const cl = crs.left();
12   const cr = crs.right();
13   // go over wales
14   const seeds = prog.withIndices([]);
15   for(let si = 0; si < crs.indices.length; ++si){
16     const s = crs.stitches[crs.indices[si]];
17     const l = cl.stitches[cl.indices[si]];
18     const r = cr.stitches[cr.indices[si]];
19     if(!s.getProgram()
20     && !l.getProgram()
21     && !r.getProgram()){
22       prog.withIndices([s.index]).prog(purl);
23       seeds.indices.push(s.index);
24     }
25   }
26   seeds.waleGrid(0:end, -(neckLen - i)).prog(purl);
27   // seeds.waleGrid(0:end, neckLen - i).prog(purl);
28   pass = 1 - pass;
29   if(pass) ++ci;
30 }
```

Listing 6.2: Section of user program that associates pattern actions to stitches.

as a sequence of stitches for an abstract *X-axis* whereas the extrusion forms an explicit *Y-axis* of the final wale-based grid. This abstract grid can then be used to apply scaled and tiled patterns, as naturally done with a regular $X - Y$ grid. Listing 6.2 shows a few examples.

For example, line 6 (`neckEdge.waleGrid(0:end, -neckLen)`) extrudes all the wales (`0:end`) along the edge at the neck boundary by `neckLen` stitches below ("-" sign). It then uses a simple 1×1 rib tiling (`neck.tile(0b10, 2).prog(purl);`) to create ribs that go radially from the neck. The *sub-ribs* code from lines 9 to 30 completes the base ribs by spawning additional ribs when the neck shaping makes it possible so that we end up with a complete radial extrusion in spite of the large decreases happening before the neck edge.

```
1  const caturl = '<dataurl...>';
2
3  // center stitch on front
4  const cs = prog.sketch(0).nearPosition({ x: 0, y: 150 });
5  const bl = cs.down(32).left(66);
6
7  const rows = bl.stitchGrid(Infinity, 100);
8  const img = prog.parseImage(caturl);
9  rows.prog(purl)
10 const cat = rows.tileMap(img, {
11   0: front,
12   255: back
13 }, 46, 3, 0);
14
15 // replace tiling remainder with a simpler one
16 if(rows.maxWidth){
17   const rw = rows.maxWidth % 46;
18   if(rw){
19     const rem = bl.stitchGrid(Infinity, 1).waleGrid([-rw-1,-2], 100);
20     rem.tileMap(0b1001, {
21       0: front, 1: back
22     }, 2);
23     rem.filter(s => noise.simplex2(s.index * 2, 0) > 0).prog(front);
24     rem.filter(s => noise.simplex2(s.index * 2, 0) <= 0).prog(back);
25
26     // borders
27     bl.stitchGrid(Infinity, 1).waleGrid([-rw-1,-rw+1], 100).tileMap([[0,
        1, 0]], { 0: front, 1: back });
28     bl.stitchGrid(Infinity, 1).waleGrid([-4,-1], 100).tileMap([[0, 1,
       0]], { 0: front, 1: back });
29   }
30 }
31
32 // yarn handling
33 rows.first().prog(yarnIn);
34 // rows.first().up().prog(yarnRelease);
35 rows.last().prog(yarnOut);
```

Listing 6.3: Section of user program that associates colorwork actions to stitches.

`stitchGrid(w, h, opts)` instantiates a grid from the current *single stitch* selection. Given a number of stitches along the course (w) and wale (h) directions, it instantiates a grid by first creating a sequence of stitches along the first axis (*course* or *wale*), and then extrudes each of these along the secondary axis. The options include the grid alignment w.r.t. to the stitch, as well as the primary axis of the grid. Listing 6.3 provides a few examples.

Notably, line 7 (`bl.stitchGrid(Infinity, 100);`) instantiates a grid from its bottom-

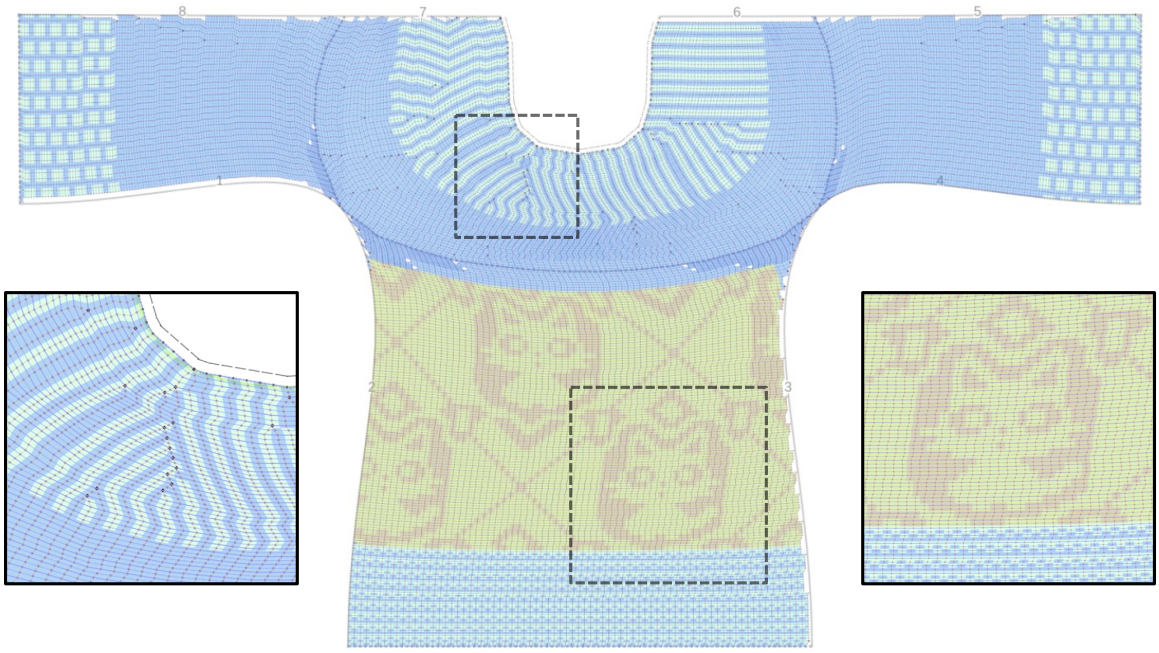


Figure 6-38: Example color-coding of programs including those from Listings 6.1 to 6.3 as applied on the front of a sweater sketch. The left highlight shows the radial ribs from the neck. The right highlight shows the fair-isle colorwork.

left corner stitch given an unlimited width (i.e., all the stitches available over each courses), and a height of 100 stitches. This is used to tile a small cat image of 46 pixels in width over that region. The tiling width may not match the image width so that lines 16 to 30 replace the remainder section that does not complete a full tiling by some random simplex-based noise pattern. The main fair-isle actions (`front` and `back`) from Listing 6.1 correspond to the front and back yarns that have two main passes: one that knits its corresponding yarn, and the other missing the other yarn. Finally, note that the secondary yarn is explicitly inserted and removed with the actions of lines 33 and 35.

6.9.2 Screen-space vs. Stitch-space Layers

While using layers is an intuitive approach to customization, we have to deal with an obvious conflict which is that our patterns are applied on the stitch graph, whereas visual layers in vector graphics are typically in the same sketch space so that they interact appropriately.

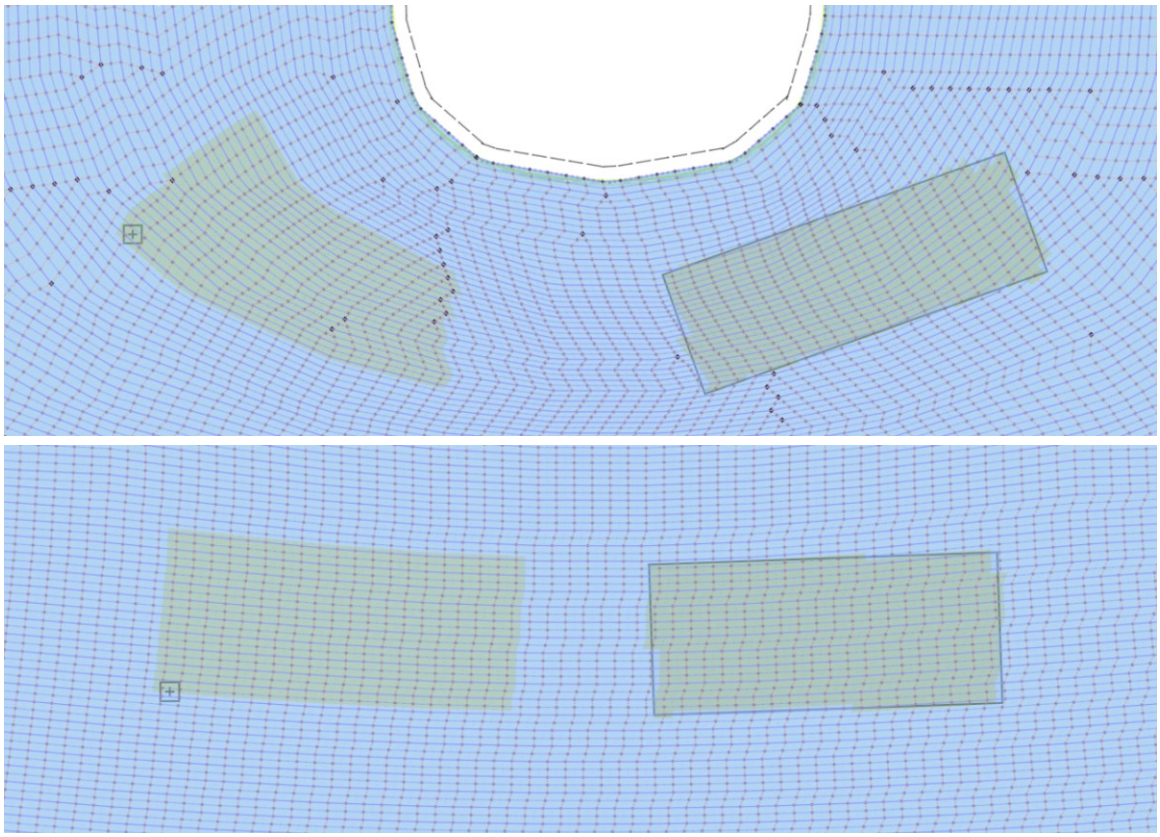


Figure 6-39: The stitch covering of a 20×20 anchored grid in stitch space (left) and a rectangular sketch grid (right). The upper section shows a *high-curvature* region and its impact in terms of stitch coverage and regularity. The lower section shows a *low-curvature* region in which both options are quite similar, notwithstanding some recurring alignment issues with the boundaries of the sketch-space rectangle.

This brings multiple options for representing layers such as

- Using children sketches – in sketch space;
- Using dedicated *rectangle* objects – in sketch space; or
- Using pointwise *anchors* that span regular *grids* – in stitch space.

The first two options (general children sketches and specific rectangle objects) are in sketch space, whereas the last one uses a pointwise location in sketch space, together with a grid specification similar to that of `stitchGrid` in the previous section. Figure 6-39 illustrates both the anchored grid and the rectangular grid.

Practically speaking, geometric shapes in sketch space are great as *masks* since they provide a trivial way to select regions to apply patterns on (i.e., we just have to

check whether the position of a stitch is within the corresponding polygon). Unfortunately, the underlying stitch structure may typically not align well with arbitrary layers so that these will often require tuning by the user when the underlying stitch graph changes. As a result, we should not expect to use them for complex local structures such as with lace patterns. We allow the use of general children sketches (any geometry shape) as *masks* to clip the impact of other layers, whereas specialized *rectangle* shapes can be associated with an image that serves as a stencil for basic scalable stitch patterns or colorwork.

The stitch-based layer anchor serves to sample the closest stitch to its location in sketch space. Then, this stitch is used to instantiate a grid with `stitchGrid` and its associated parameters (width, height, main axis and alignment).

6.9.3 Layer Interactions

In vector and raster graphic applications [14, 98], layers typically don't exist in complete separation. We thus need a means to compose them on top of the sketches. The main idea is to apply their logic in different steps that allow for information to be combined appropriately. Our system applies layers by: (1) *selecting* the associated stitches (typically based on the layer container – an anchored grid or a specialized rectangle); (2) *marking* the stitches with information corresponding to that layer (i.e., the stitch type, the yarn stack, or the base yarn); and finally (3) *unifying* the stitch information into an *action* similar to programmatic *user actions*.

In our implementation, the unification step is basically a single procedure that looks at the *yarn stack* at a stitch, together with its *stitch type*, and the relative yarn stacks at the stitch neighbors to decide what the action should be at the given stitch.

The order of layers has an impact on their application and the result. For example, with color patterns, later, overlapping patterns may overwrite the information from preceding ones. The user can eventually change that order. And we envision that one could even decide how this information "merging" happens with modes similar to graphics blend modes [98].

In the remaining sections, we describe the different layers we implemented, their

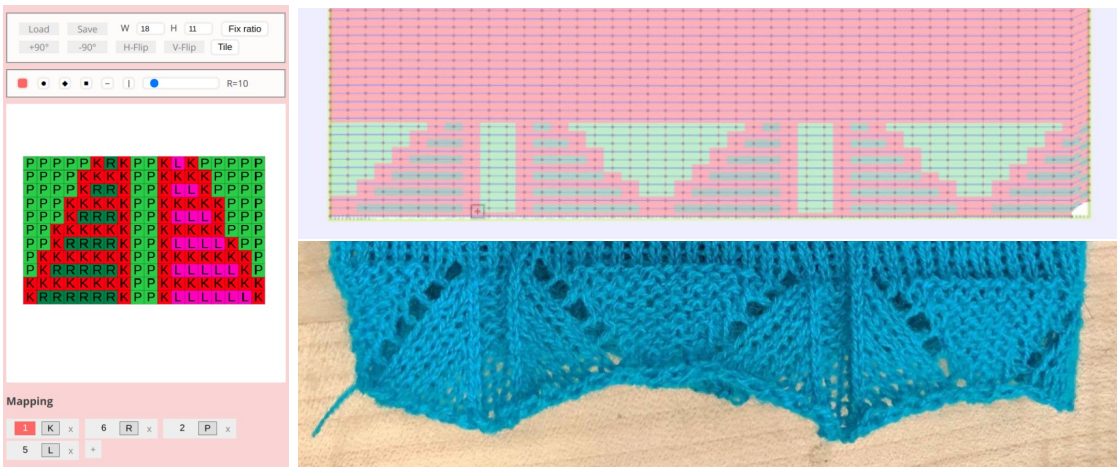


Figure 6-40: Example of lace pattern in the editor (left), its corresponding color-coded program visualization as the hem of a tubular sketch (top-right), and the corresponding knitted artifact (bottom-right).

intents and provide relevant implementations details. All are applied on top of the *traced graph*, with the exception of the last *intarsia* layers that work with the initial *stitch graph* and lead to re-*tracing*. In *layer editing* mode, after the sampling has been done, the user can modify the layers and get direct visual feedback (i.e., stitch type and yarn) until they are satisfied. At that point, the rest of the pipeline is re-enacted to get the final schedule and code.

Two common properties of all layers are: (1) a single-channel *pattern image* that represents the pattern data as a grid of pixels, together with (2) a *mapping* from pixel data to layer representation (e.g., stitch type or yarn index). The image can typically be input either as a text file (i.e., for stitch patterns), as an image file, or directly drawn as part of our interface.

6.9.4 Stitch Pattern Layers

Stitch pattern layers are basically the equivalent of those in the primitive-based design of Section 5.3. We use the same pattern instructions to describe the individual *stitch type* during the *marking* step, and allow the user to select either a *scaling* or *tiling* mode of the layer pattern. Figure 6-40 shows our pattern editor with a lacy hem pattern that is tiled along the bottom boundary of a tubular sketch.

Front Yarn	Back Yarn	Description
True	*None*	A basic front yarn stitch (based on the *stitch type*)
True	*Miss*	↿
True	*Tuck*	↿
True	*Knit*	A front-back knit operation for double-sided fabric
False	*None*	The yarn is not present
False	*Tuck*	The yarn tucks to the front side to maintain connectivity during floating (i.e., wide fair-isle)
False	*Miss*	The yarn floats in the back (i.e., general fair-isle)
False	*Knit*	A basic knit stitch for the opposite side of a double-sided fabric (i.e., Jacquard knitting)

Table 6.1: The potential states for each yarn and the corresponding interpretation

6.9.5 Multi-Yarn Pattern Layers

Multi-yarn patterns modify a stack of yarns per stitch, which is composed of two parts: (1) a front yarn and (2) a set of back yarns with attributes. Here, the terms *front* and *back* refer to the side of the fabric, respectively the *outer* and the *inner* sides. Notably, they do *not* refer to the needle bed so that a front yarn may happen on the back needle bed and vice versa.

The *Front Yarn* corresponds to the yarn we want to appear on the outer side of the fabric being knitted (i.e., one of the available carriers). By default, this is the base yarn being used in the trace (which may be modified by the *intarsia layers*).

The set of *Back Yarns* represents the additional yarns that pass in the inner side of the fabric. Each yarn is associated with a mode that can be one of: *None*, *Miss*, *Tuck* or *Knit*. The default is for a yarn to be absent: i.e., its mode is *None*.

Multi-yarn patterns can modify both the front yarn and the back stack. The meaning of the possibles combinations for a given yarn Y are listed in Table 6.1.

Marking. A multi-yarn pattern effectively allocates the back yarns not only for the region it covers, but for its full lateral extents, up to intarsia boundaries.

Unification. By ensuring that we cover the full lateral extents of the local region, then the multi-yarn actions at each stitch become very simple: we can assume that all the stitches we encounter in the active slices are covered by all the yarns we have allocated, so that the per-stitch actions have as many *main* steps as there are locally allocated yarns – i.e., one step per yarn, in a given fixed order.

Special *pre* and *end* steps are added to the first stitch of such region if the previous (respectively next) stitch has a different yarn stack so that the yarn difference can be inserted (respectively removed).

Float Patterns

Float patterns essentially specify the front yarn and allocate back yarns as floating in the back – i.e., the non-front yarns that are part of the pattern at a different location. Figure 6-41 illustrates a simple checkerboard pattern that is tiled from the trunk region across the neck interface of a yoked sweater.

Tuck Patterns

Tuck patterns describe the location of tucks when floating yarn in the back over large number of needles. They are an important component of complex fair-isle colowork.

Fair-isle colorwork typically knits on the front side to show the pattern and floats in the back otherwise. If the pattern changes often between front and back, then the floats end up well-connected to the front fabric. In the general pattern case, we can have yarn that floats for long ranges of needles and this leads to dangerous operations. Large floats are not stable – i.e., the yarn tends to be loose – and may get caught by needle actions or prevent proper needle operations. By inserting tucks to the front side, we can introduce local connectivity (e.g., every N stitches) to ensure that the floating yarn is properly connected and stable. Figure 6-42 shows a fair-isle pattern that covers a small section of a tube. It includes a simple *float pattern* layer in the center, whereas the rest is covered by a tiled *tuck pattern* (we show two different variants of the tuck pattern). The *tuck pattern* ensures proper connectivity for the floats outside of the base local pattern.

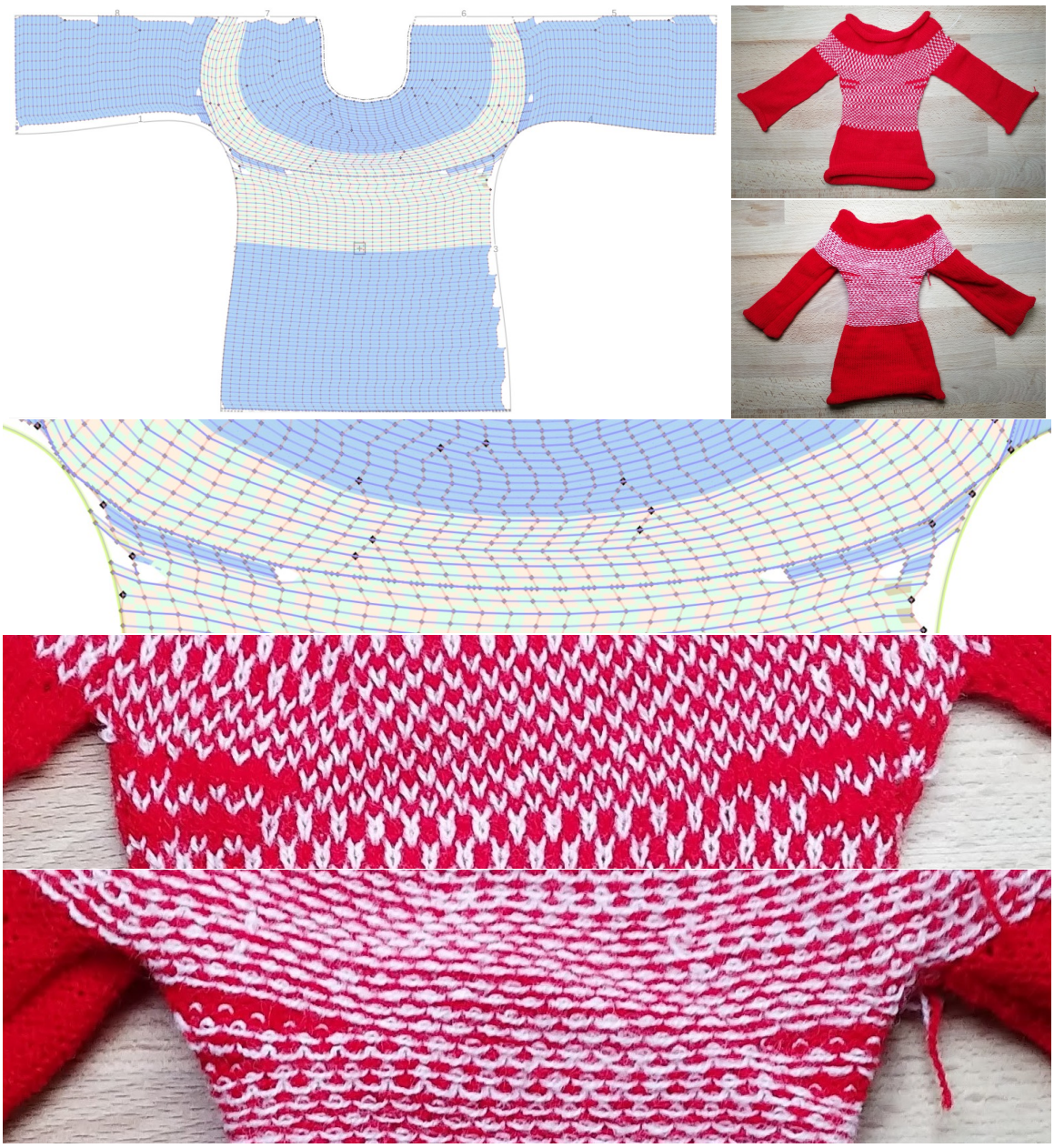


Figure 6-41: Example of simple float pattern that does not require any tuck pattern because the tiled checkerboard pattern ensures that floats are tightly connected to the main fabric. The close-ups from top to bottom: program, knitted sample, and inside-out version. The anchored grid is aligned to the bottom center and its primary axis is a wale, with 100% course width. The primary wale does not cross any short-row so that these end up excluded from the pattern.

Figure 6-43 shows a failure case due to the lack of negative *tuck pattern* inside of the main float pattern. Because the main pattern has wide sections of the same color, we end up with long floats and these lead to issues at their boundaries. The

Figure 6-42: Local float patterns typically need accompanying tuck patterns to ensure floats are properly connected to the fabric at regular intervals.

two potential fixes are: (1) changing the float pattern so that it does not include such long floats, or (2) introducing a tuck pattern on top of the float pattern for proper connectivity within the long floats.

One user option concerns the interpretation of *floats*. They can be *implicit* or *explicit*. In the former case, no action is actually triggered, and the float is implicitly happening through the carriage movement to the next operation. This implies that if there is no *next front stitch* or *next tuck* in the current pass of the code generation, then there is no float. It enables minimal float regions but may create incomplete float regions if the layer extents are over a whole tubular structure. The latter *explicit* case uses an explicit `miss` operation.

Figure 6-43: A failure example whose main float pattern ends up with too wide floats that lead to failure at their boundaries.

Jacquard Patterns

The term "Jacquard knitting" typically refers to the process used for creating two-sided knit fabric. Section 3.2 includes multiple Jacquard packages from Shima and their corresponding knitted results. Our Jacquard patterns emulate the same process.

Figure 6-44: Jacquard patterns with 2 colors: their fronts and their backs. In clockwise order, from the top-right: floating, horizontal, tubular, pique, vertical. Note that the sample with *horizontal* backing (bottom-right) looks taller. There are as many front stitch as for the other, but the backing generates twice the density, which stretches the fabric vertically and introduces some bending.

The principle can be considered as a generalization of fair-isle patterns, but one which loses the ability to work with tubular structures. In our system, a Jacquard pattern ends up acting like a *float pattern* in that it enforces that the underlying image ends up visible in the front of the fabric. The main difference is in how we treat the *backing* of the fabric.

Importantly, Jacquard fabric is restricted to flat sheets of fabric. This restriction enables us to have a larger degree of freedom in terms of the operations that happen in the back of the fabric because we can effectively use the opposite needle bed without restrictions – i.e., we can knit the fabric backing there.

Backings. The *Float* backings act like in fair-isle patterns, with fixed tuck spacing to enforce proper connectivity to the front of the fabric. The *Tubular* backings knit the back yarn directly on the back bed – while partially floating in between when using more than 2 colors. The *Alternate* and *Pique* backings alternate the yarn that knits in the back to get a mostly uniform noisy checkerboard pattern. The *Horizontal* and *Vertical* backings form correspondingly oriented stripes of the different yarns. Figure 6-44 shows the front and backs of a same 2-colors pattern with different backings, whereas Figure 6-45 includes 3-colors patterns.

Figure 6-45: The front of the CSAIL logo with a tubular backing (top) and the front and back of a 3-colors cat with an alternate backing (bottom).

6.9.6 Intarsia Layers

The term "intarsia" generally refers to artistic techniques used to insert decorative elements inside of a more general matrix – notably with wood inlaying [79]. In the knitting world, it refers to a form of local decomposition of the fabric into smaller sections that act similarly to parts being *inserted*.

Two important aspects of intarsia are: (1) it makes use of separate yarns in different sections of the knit structure, and (2) it requires proper connectivity between each of the regions, notably across the course boundaries.

We deal with the first part by changing our *tracing* algorithm to take into account new *sub-regions* of the stitch graph that only let specific yarns pass. This effectively forces the yarn to switch direction and requires a new rule to *introduce yarn locally*. The connectivity issue is dealt with by our tuck annotations during tracing.

Marking. The intarsia layers deal with a *yarn mask* that is specified on each of the initial stitches of the base *stitch graph* – before tracing. No unification is needed since no action is actually generated for those stitches.

Tracing with Intarsia

The first modification is that the tracing algorithm keeps information about *pending yarns* and their tracing state (stitch, pass, orientation).

Then, while all previous rules are kept, they must now check whether the next stitch they are leading to can be reached with the current yarn. This is done by checking whether its corresponding *yarn mask* includes the current yarn – in which case we can proceed – else the rule is rejected.

When all rules are rejected, we then attempt to switch to a locally *pending yarn*. We repeat this at least once for each of the pending yarns. Assuming all pending yarns have failed – i.e., we are in a form of locking scenario where no yarn can proceed without cutting –, then we try to *introduce* a new local yarn. This is an additional rule on top of those of Narayanan et al. [122].

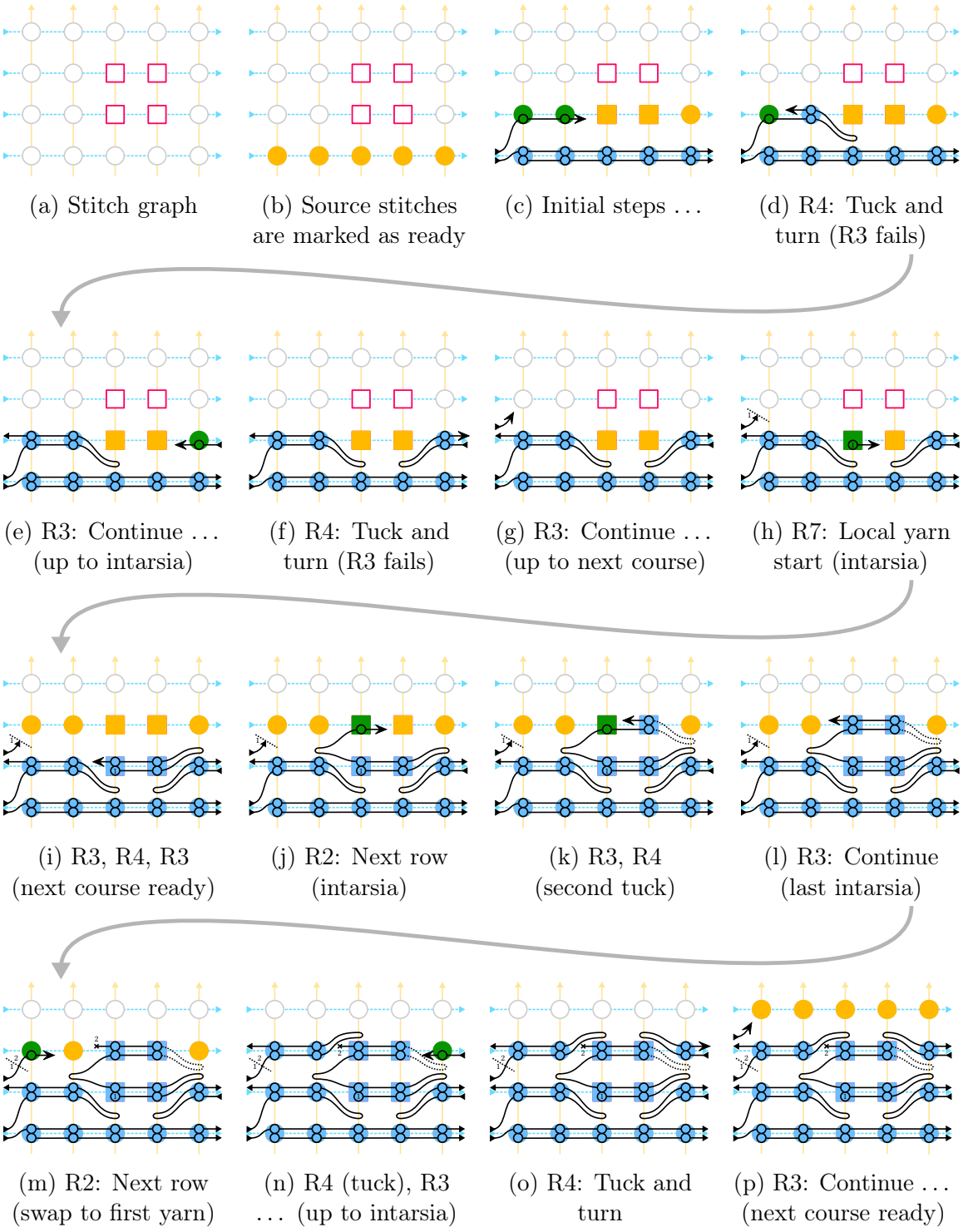


Figure 6-46: Tracing example of a tubular structure with a block of intarsia (square nodes with distinct yarn mask). The step (d) happens because the first yarn does not match the mask of the intarsia block. Step (l) is the last within the intarsia block. Step (m) switches to the pending first yarn.

R7 – Start Local Yarn: Search for an available yarn that is not currently active and can start from one of the *ready* stitches available – i.e., it matches the corresponding *yarn mask*. If a yarn is available and can start from a *ready* stitch, then switch the tracing state to the new yarn – storing the previous one as pending – and knit the corresponding stitch.

If none is available, then the rule is rejected and the tracing iteration ends locally. This happens in two scenarios: (1) when we've completed a region of the region graph that is either the last one, or requires us to restart from a different region; and (2) when we're unable to complete a current region node without cutting one of our pending yarns. In both scenario, we basically end up triggering rule **R6 - End Yarn** with the current yarn, before restarting the tracing iterations. Figure 6-46 illustrates the new tracing with a basic block of intarsia inside a tubular structure.

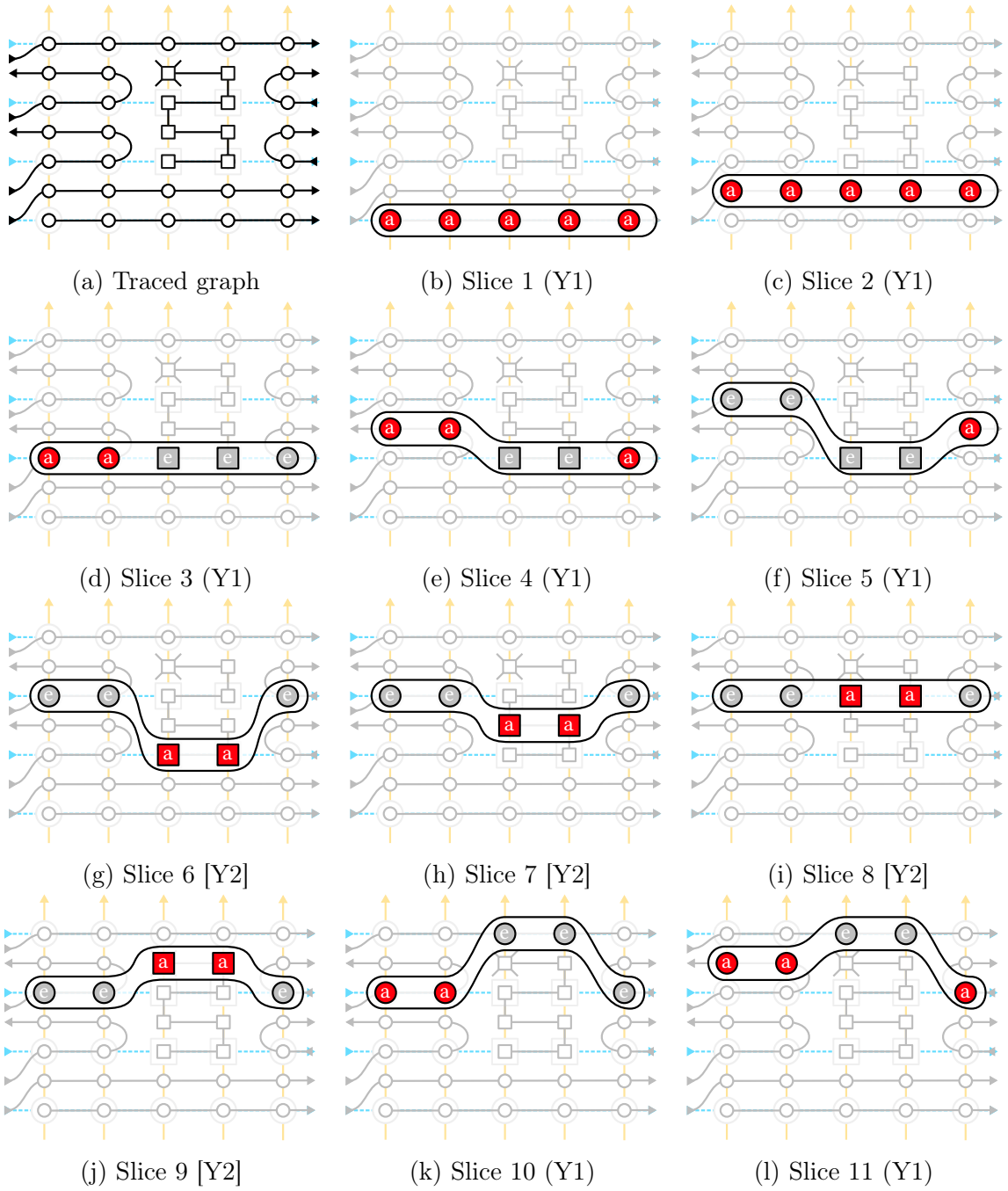
Note that a pending yarn that cannot do any action and has its upper stitches all completed can be stopped automatically so that it can be reused before the end of the current node. Furthermore, to avoid that each node gets a new yarn, we use a restrictive yarn mask by default: i.e., the default yarn mask is set to the first yarn only, which requires cutting at the end of a region node if another disconnected node must be completed.

Slicing and Scheduling

We modify slicing by simply requiring a new slice whenever the yarn changes. The rest works without modification. Especially, scheduling does not require any change as it did not make any assumption regarding the underlying yarn. Figure 6-47 illustrates slicing given the tracing of Figure 6-46.

Examples and Limitations

Figure 6-48 illustrates the use of intarsia at the center of the trunk region of a small sweater prototype. Figure 6-49 illustrates the combination of an intarsia layer with a float pattern to create local colorwork. This relies on the fact that the extents of the multi-yarn patterns end at the boundary of intarsia regions.



(a) Traced graph (b) Slice 1 (Y1) (c) Slice 2 (Y1)

(d) Slice 3 (Y1) (e) Slice 4 (Y1) (f) Slice 5 (Y1)

(g) Slice 6 [Y2] (h) Slice 7 [Y2] (i) Slice 8 [Y2]

(j) Slice 9 [Y2] (k) Slice 10 (Y1) (l) Slice 11 (Y1)

Figure 6-47: Slices of the trace shown in Figure 6-46. The active yarn is indicated with (Yi) in the caption of each sub-figure.

Tuck Connectivity. Tuck annotations happen during tracing when one of the rules that changes the yarn direction is triggered. In the base tracing without intarsia, this happens whenever we reach the end of a short-row (or the end of a course in flat fabric) and have to change direction. With intarsia layers, this effectively happens

Figure 6-48: A small intarsia sample illustrating a single intarsia layer that carves a section with a distinct yarn.

also when we reach an inaccessible intarsia region and have to turn back. The real tuck is generated during the action pass based on the tuck annotations and its safety considering the needle bed state.

Ideally, the algorithm should be distributing the side tucks uniformly. A corollary is that we should not be tucking twice on a same stitch with the same yarn, as doing so implies that we are lacking a tuck somewhere else. Our basic algorithm extension does unfortunately not necessarily prevent double tucks. Because of double-tracing, we may end up doing two passes against a barrier until we must stop, as illustrated in step (k) of Figure 6-46. One basic improvement consists in switching yarn if there is any pending yarn, before or after turning back from an intarsia barrier, which we use for Figures 6-48 and 6-49. Another strategy would be to synchronize the pending yarn traces so that they move all in the same direction: e.g., disable a yarn if it needs to change direction but another pending yarn is going in the current direction. A full solution would optimize for the switching by searching across the different options.

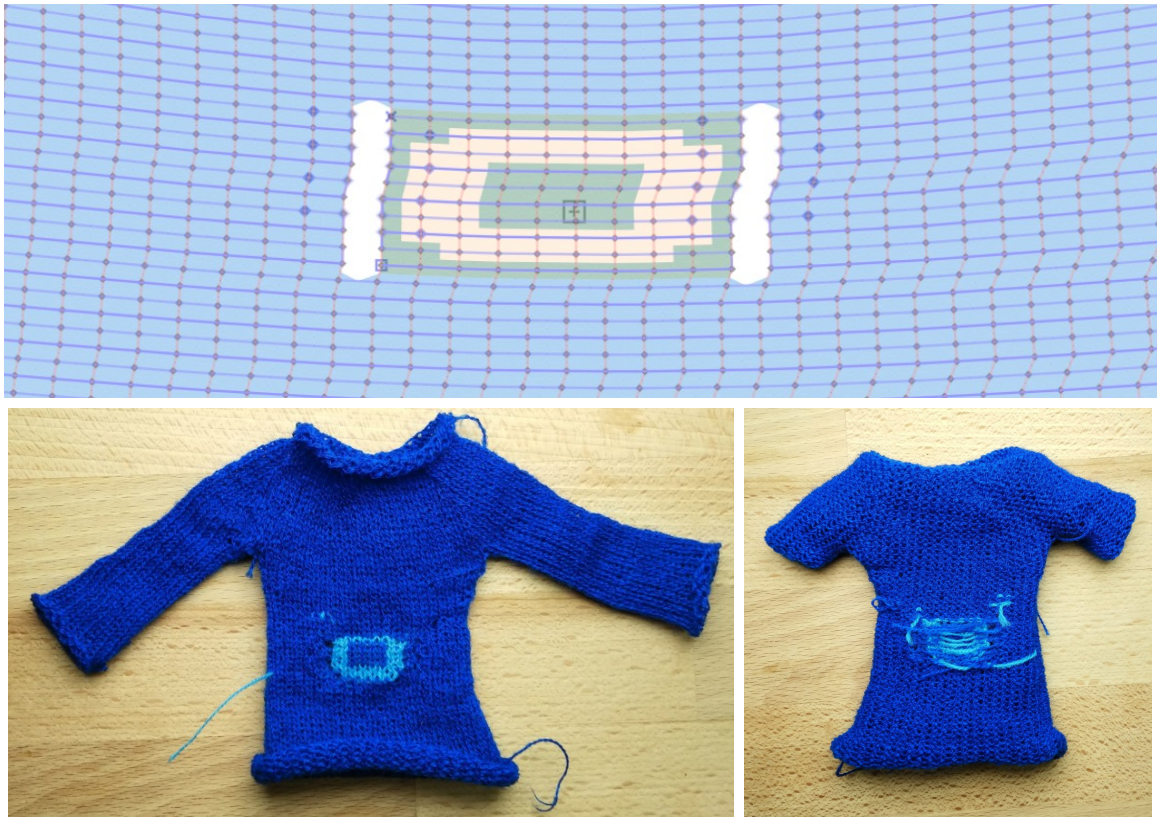


Figure 6-49: An intarsia sample that uses an additional float pattern over the intarsia layer. The pattern consists in a sample ring. The remaining yarn is chosen so that its color is the same as the main sweater yarn. The inside-out picture (right) shows that, as a result, the float is only local.

6.10 Results and Discussions

6.10.1 Knitted Garment Samples

All the following results were knit with 2/30 1-ply acrylic yarn. They are knit in half-gauge, with the following size measurements on a tubular swatch: $D_{\text{crs}} = 300$ mm/100 stitches and $D_{\text{wale}} = 135$ mm/100 stitches.

Most of our garment patterns are created by manually redrawing on top of original patterns selected from BurdaStyle. The only exception are: the first *sweater*, which we drew from scratch to showcase the capabilities of our system and to serve as a simple introductory design, and the *beanie*, which is based on the *Joyful baby bear hat* from Joy Kelley at howjoyful.com.

Figure 6-50: Our larger examples on a 4-foot boy mannequin, together with top-down views of the individual garment pieces and a zoom on one of the inseam pockets of the trousers which are knit as inside-out tubular structures merging with the body.

Young boy garments Figure 6-50 shows the larger-scale examples we knitted for a 4-foot-tall boy mannequin, including three garment pieces. These results verify our pipelines' ability to scale to human-sized garments. The primary constraint preventing a full adult-scale garment is our knitting machine target. Keeping in mind that we knit in half-gauge, our largest example, the sweater, takes over 309 needles of our knitting machine bed, out of 541 available.

The *beanie* with earflaps showcases a mixed flat/tubular structure. Both earflaps use a garter pattern over their entire structure to avoid curling and folding, which

is particularly pronounced with flat Jersey fabric. The upper section uses a fair-isle pattern that is tiled horizontally and floats the background yarn inside.

The *sweater* includes partial rib patterns at the wrists, a waffle pattern at the base of the trunk, and a radial rib pattern for the neck. It also includes fair-isle colorwork in the center section.

The pair of *trousers* uses ribs around the waist and garter patterns on the ankles. The original trousers pattern did not have any pockets. The inseam pockets on the side of our trousers were added by cutting and pasting the segmentation of a pair of pockets from a different garment. This illustrates that multiple existing sketches can be reused to build more complex ones.

Both the trousers and the shirt exhibit yarn breakage in the armpit regions unless short-rows are used.

Smaller mannequin garments Figure 6-51 shows different garment pairs (uppers and bottoms), whereas Figures 6-52 and 6-53 show the top-down views of the complex upper garment patterns, together with a visualization of their sketch atlas with linking. They are scaled to fit on a 16-inch wooden mannequin. All garment patterns use patterning at their extremities, typically ribs or garter stitch, to ensure that they don't curl or fold.

The *cardigan* is knitted flat from top to bottom, to avoid having to split the yarn between three sections (front left, front right, and the back). Splitting can lead to yarn breakage unless each section is knit in parallel; our scheduler is sequential, so we do not support this. For the same reason, we do not link the top section, but bind it manually instead. Since the whole structure is flat, we use a global garter stitch pattern to prevent it from curling and folding.

The *hoodie* and *jacket* examples both showcase c-shaped knitting layouts for which one side of the panels are not linked. The *turtleneck dress* was originally opened at the top of the back to make it easier to put on. However, we closed the opening to simplify its manipulation given its physical scale.

The *princess dress* is knit in two variants. The first version in Figure 6-53 is knit as

Figure 6-51: Examples of dresses on 16-inch mannequins

Figure 6-52: Top-down views of upper garments (left) and their corresponding sketch atlas (right): the cardigan (top), the hoodie (middle) and the jacket (bottom).

a single piece, showcasing one of the potential advantages of whole-garment knitting. The pleats found in the original pattern are non-trivial to knit automatically, so we substitute a series of darts at the interface between the skirt and the body. We attract irregular stitches to the dart edges via seam annotations, and use rib patterns

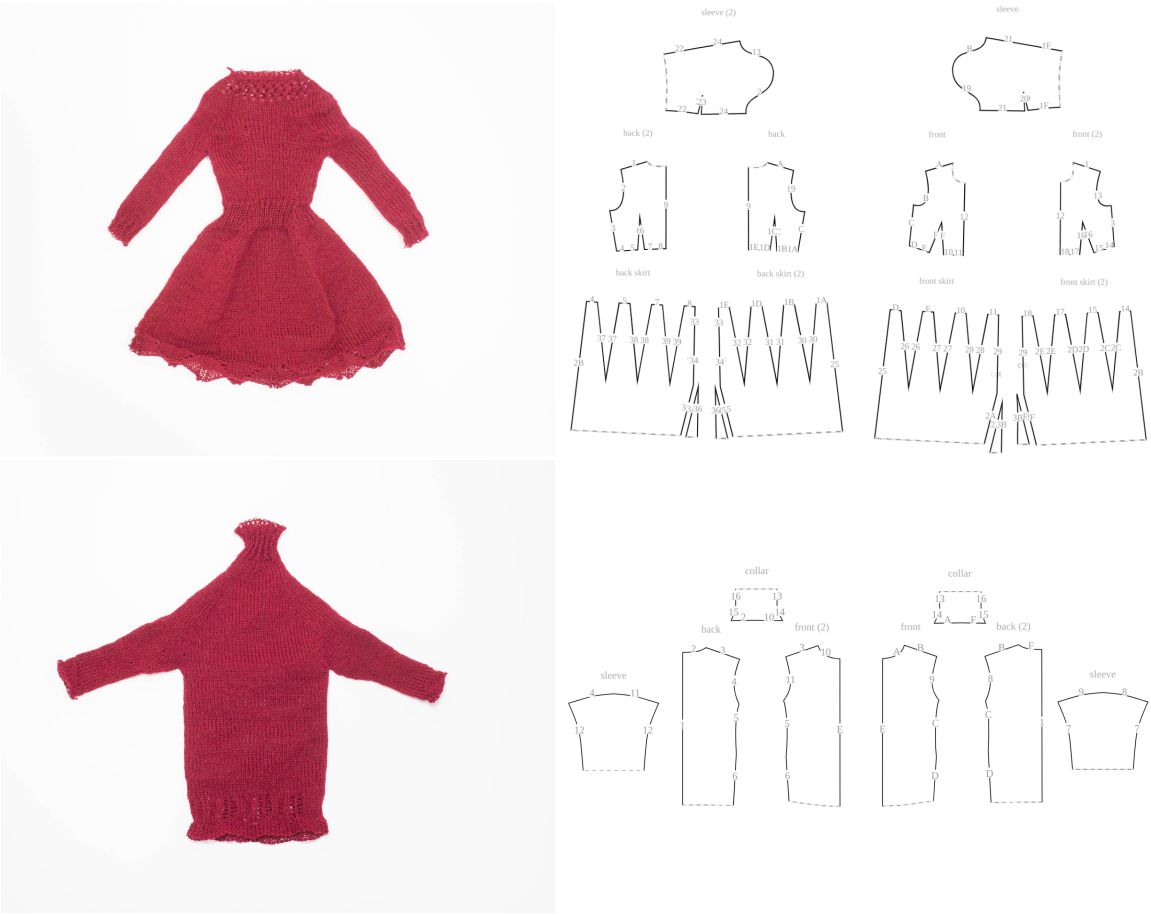


Figure 6-53: Continuation of Figure 6-52: the princess dress (middle) and the turtleneck dress (bottom).

above that waist interface to strengthen the visual impact of the folds. Near the top of the neck and the bottom of the skirt, we showcase different tiled lace patterns. The second version of the dress, shown in Figure 6-54, features the original pleated pattern, which can be knit in two sections and bound manually.

6.10.2 Scheduling Algorithms

Existing scheduling algorithms [103, 122, 188] either work with tubular or flat fabric, but not both. To support the scheduling for some of our mixed flat and tubular designs, we extended the work of Narayanan et al. [122] with *single-fold* and *c-shaped* layouts. The main take-away is that scheduling becomes, perhaps counter-intuitively, harder. Flat structures can be folded in different ways, and their parameter-varying

Figure 6-54: Two-parts version of the princess dress, with manual binding done with box pleats.

extents substantially increase the search space. While some of the structures may appear simpler locally, their interactions become more complex.

One major issue we encountered with existing schedulers is that they rely on the assumption that transferring stitches around is fine as long as excessive slack and unwanted loop overlaps are avoided. Our experience seems to indicate that large stitch cycle transformations typically lead to some form of failure (due to transfers). Similarly, the current general-purpose transfer procedure *Collapse-Shift-Expand* [116] enforces slack and overlap constraints, but allows unrestricted *overlapping loop transfers* for loops that have the same target needle. While having a same target needle is necessary (i.e., for decrease shaping), overlapping loop transfers are a common source of failure. Figure 6-55 shows failure cases caused by both issues.

We envision that part of the scheduling should be guided by the user similarly to how our workflow allows control of the directions and isolines of the knitting time process. Current schedulers have enabled many applications, but they would be more practical if the user could interactively manipulate their process.

Figure 6-55: Example of knitting failures due to failing needle transfers: the *left* example failed at large decreases above the crotch due to non-ideal schedule alignments; the *right* example had catastrophic failures due to overlapping loop transfers during shaping transfers.

6.10.3 The Importance of Details

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

Seams

Figure 6-56 illustrates the impact of the irregular stitches and how our wale penalty deals with their specific placement. While the location of the singular stitches is

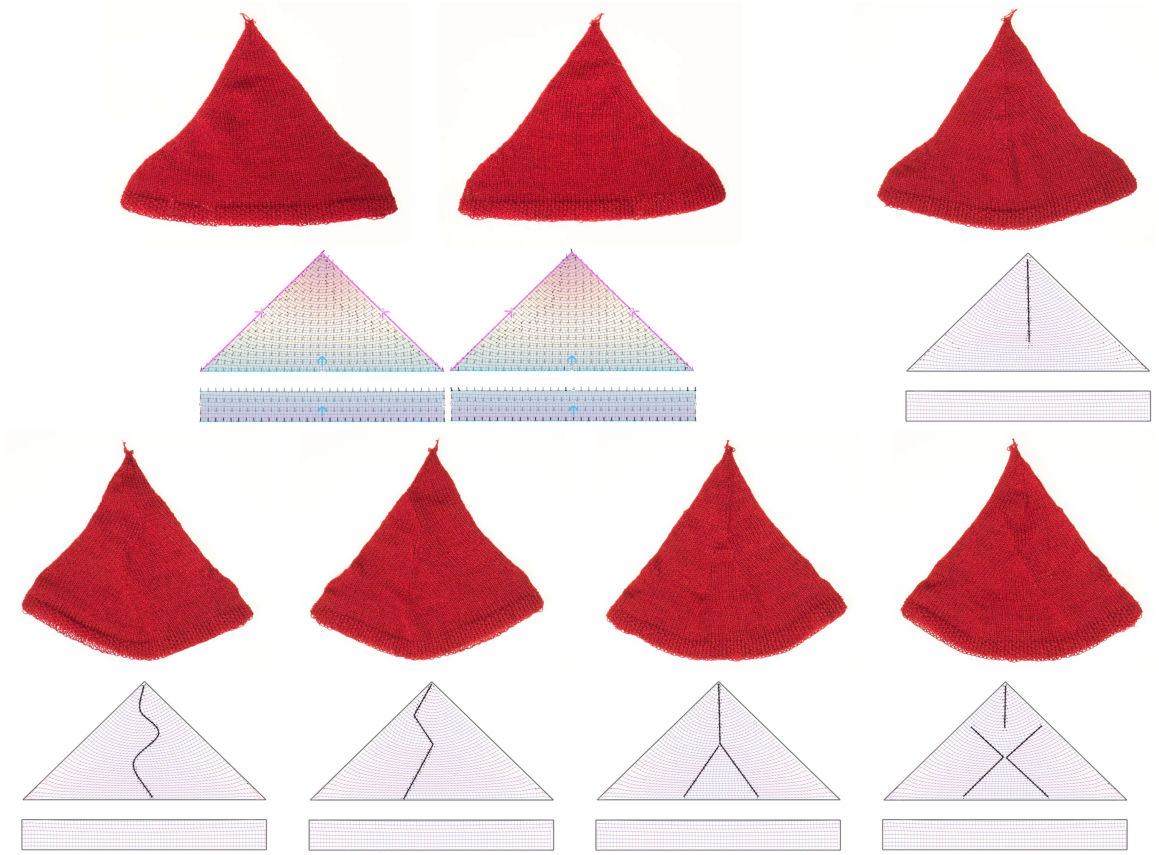


Figure 6-56: Illustration of the impact of seam annotations with the corresponding irregular stitch placement.

reasonably clear in such samples, one limitation of our penalty-based editing is that the wale distribution is done independently per course pair. Thus, we do not have any notion of the alignment of irregular stitches across subsequent courses. Although this global alignment is important in practice, it is not fully controllable in our system. Another limitation comes from double-tracing of the stitch graph: irregular stitches of a kind (increase vs decrease) only happen on every other course. This results in seams that are less prominent from those we could generate in Chapter 5 (see notably Section 5.2.5). 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.

Figure 6-57: The addition of color work and stitch patterns can highly improve the final appearance, which calls for dedicated means to specify those.

Customizing Stitch Patterns

Figure 6-57 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 is critical.

Garment Sizing and Preview

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 Figure 6-58, by slightly changing the scale, we go from a shirt that looks pretty tight, to one that is appropriately loose, if not too loose.

Our stitch sampling strategy makes the simplifying assumption that the number of stitches along courses and wales are sufficient to describe the garment size through two constants D_{crs} and D_{wale}. This is a gross approximation and does not account for the impact of the underlying garment curvature or the impact of different stitch operations and surface textures. From a design perspective, we are missing two components: (1) a more accurate simulation of the size, which would inform the sampling strategy (e.g., through fast simulation [97, 179]), and (2) a means to adjust

Figure 6-58: Two slight scale variations of a same shirt input showing the importance of proper sizing.

the desired size along specific target curves directly (either by optimizing the sketch, time function or the stitch graph), rather than searching for it iteratively as in the current workflow.

Finally, our system only tackles the intrinsic aspect of knitted fabric, whereas a full system would benefit from a full 3D garment preview. Flattened shape editing requires experience with the traditional cut & sew workflow and an intuition for how local pattern editing influences the final shape. A clear next step is to provide an interactive 3D preview and manipulation alongside the 2D pattern editing capabilities, as is already common in professional garment authoring software [39, 113].

Local Knitting Procedures

Figure 6-59 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.

Similarly, *binding* the yarn *on* and *off* the needles can be done in various ways that change the tightness and appeal of the garment boundary edges. In general, this calls for a more general framework that can explore those customization capabil-

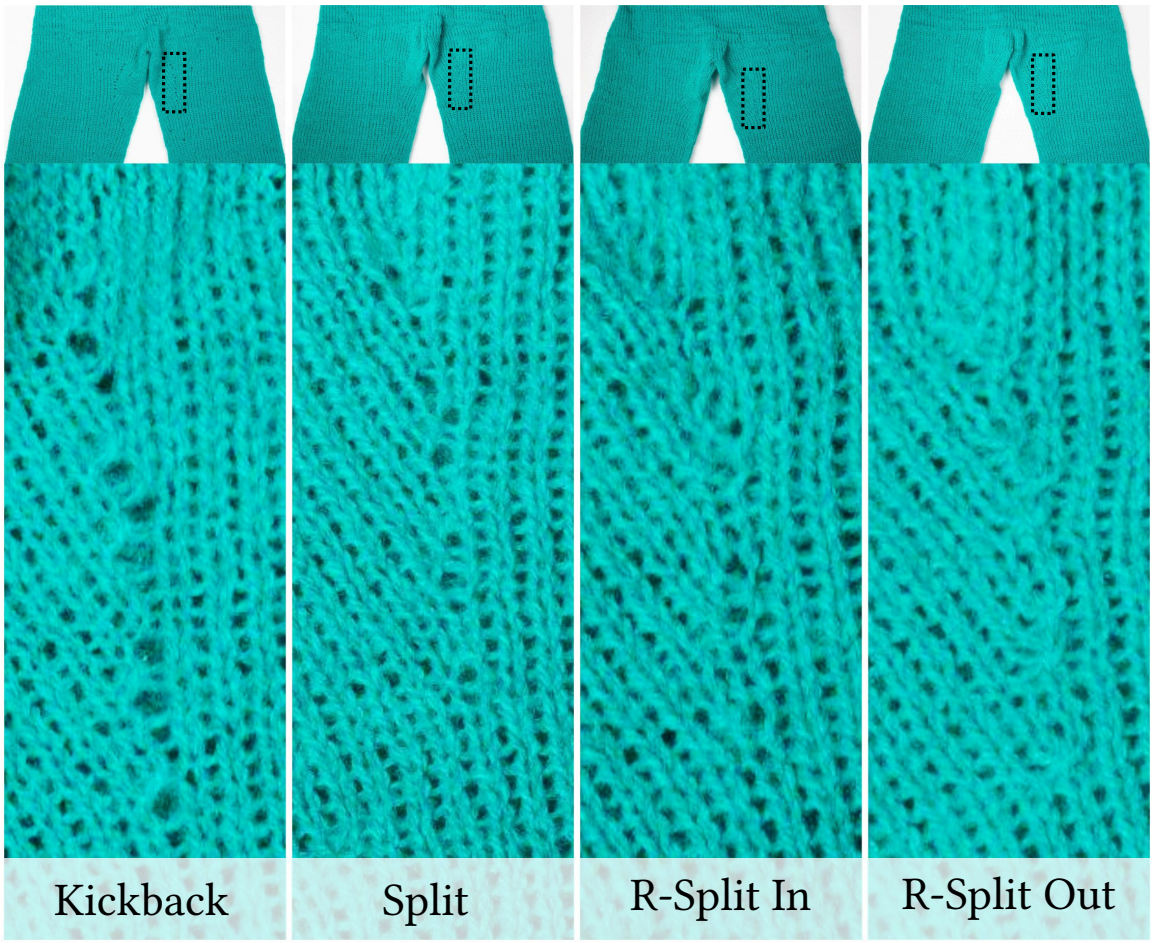


Figure 6-59: Local appearance of different stitch increase procedures: kickback, split, reverse-split inward, reverse-split outward.

ities intuitively and possibly select them locally given functional specifications from the user (e.g., yarn looseness, tension, structure strength and durability at critical interfaces).

6.10.4 Binding Fabric

In this work, we consider the binding of garment panels as a direct one-to-one continuation of the fabric. This provides a very simple and intuitive support for *darts*, which our system simply considers as direct links from one side of the fabric to another (no fabric is actually cut or folded). However, cut & sew supports various other means of manually binding pieces of fabric together, including pleats, ruffles, zippers, or other non-manifold bindings of multiple fabric layers together.

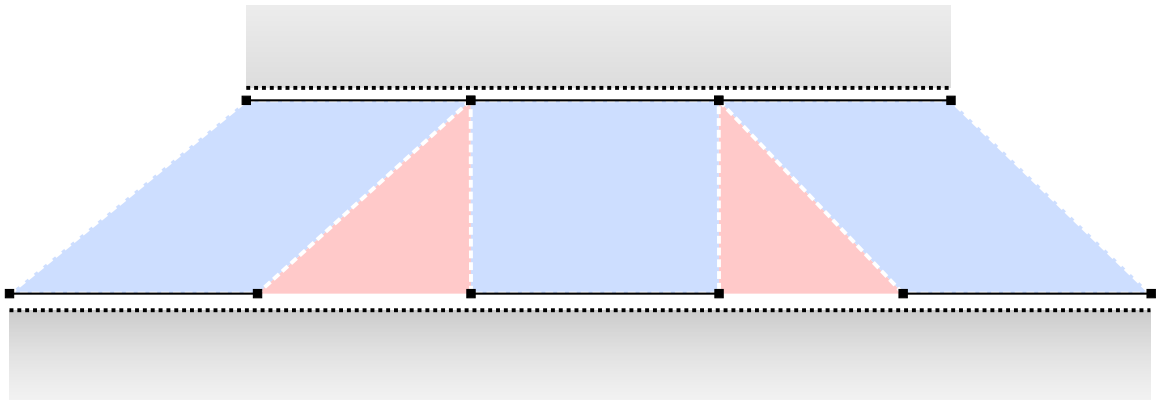


Figure 6-60: Pleat binding: blue regions are links between the two panels (in gray), red regions are the intermediate regions to fold / bind off.

Of those, *pleats* are likely the next, most amenable to automation. The current workflow is theoretically able to deal with pleats at least partially: users can bind a larger interface to a small one by splitting the large one into pairs of linked and unlinked sections that cover the binding of the smaller one (see Figure 6-60). With dedicated schedules or knitting procedures, one may be able to automate the folded binding of the intermediate regions. A partially manual solution is to bind off the intermediate section, which the user can then fold and link. The main bottleneck is that large changes to the number of stitches without coordinated increases/decreases lead to excessive stitch rotations that are hard to knit successfully.

Another related issue is that of the fabric purpose. In our system, all sketch charts have the same purpose: composing the apparent garment shape. However, cut & sew includes various types of fabric panels, such as *lining* or *facing*. Each typically serves a distinct purpose such as to modify the fabric's appearance, structure, or rigidity. When interpreting a garment pattern purely from the shape perspective, our system would typically discard the additional fabric panels. By contrast, it would be ideal to account for their intended function using compatible weft-knitting techniques. For example, *inlay* interlocks thread in between wales without creating loops, which restricts the stretch of the fabric; similarly, *stitch patterns* can modify the appearance, texture, and tactile feel of the knit fabric. Ideally, those would all be customizable properties of the garment representation.

Table 6.2: Statistics about the result samples shown in this chapter. 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.

Sample		Complexity				
Target Size	**Sketch**	**Charts**	**Constr.**	**Regions**	**Stitches**	**Instr.**
4 feet mannequin	`beanie`	2	8	3	13184	34892
	`sweater`	2	19	4	47624	97987
	`trousers`	12	22	6	57254	120364
16 inch mannequin	`cardigan`	4	12	4	12290	31351
	`dress`	14	26	4	17238	41704
	`hoodie`	6	18	5	12874	29136
	`jacket`	5	17	4	11252	31184
	`turtleneck`	8	24	4	13426	24752
	`shorts`	4	23	3	2842	7673
	`l trousers`	12	22	6	11104	25226
	`w trousers`	6	14	3	14804	25014

6.11 Scalability and Performance

Our system is implemented as a client web browser application in Javascript and WebAssembly. We first look at the complexity behind the examples shown in the previous section. Then, we list the existing parameters and their ranges, highlighting the dynamics behind the simplicity terms within our sampling algorithm. Finally, we measure the performance of our system

6.11.1 Complexity

Table 6.2 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).

6.11.2 Parameters

Table 6.3 lists the 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

Table 6.3: The list of parameters used for the result samples shown in this chapter.

Sample		Parameters		
Target Size	**Sketch**	$\Delta t_{\min}$	λ_{srs}	**Options**
4 feet mannequin	`beanie`	0.25	0.0	Uniform branching
	`sweater`	0.25	0.1	—
	`trousers`	0.5	0.1	—
16 inch mannequin	`cardigan`	0.25	0.1	Reverse time
	`dress`	0.5	0.1	Reverse split
	`hoodie`	0.5	0.3	—
	`jacket`	1.0	0.1	Reverse split
	`turtleneck`	0.25	0.1	—
	`shorts`	0.25	0.1	Reverse split
	`l trousers`	0.5	0.1	—
	`w trousers`	0.25	0.1	Reverse split

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 6.3 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

Table 6.4: 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*.

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

default, simpler *kickback increase*. For those results, we used the *reverse split inward* variant shown in Figure 6-59.

6.11.3 Interactivity

Table 6.4 lists runtimes of different sections of our system, computed using an Intel Xeon i7 CPU with 32GB of RAM, as a single-threaded web worker computation beside the UI thread. The main take-aways are that time and regions computations achieve both interactive frame rates, whereas sampling and scheduling do not, unless done at a small scale. However, because sampling results are cached, the user can edit seams and modify layers at interactive frame rate after the first pass of sampling has completed. The scheduling does not benefit from caching, but it is typically not required for user feedback. The following paragraphs provide an interpretation of the runtimes.

Mesh-based Timings The timings of the *left* group (*time, segment, geo*) are mainly dependent on the mesh levels. 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 visual feedback typically comes in even less time given the coarse-to-fine, iterative nature of our computations. In practice, we throttle the user feedback to some fixed frame rate (i.e., 60FPS) as web-worker transfers together with asynchronous scheduling induce a noticeable overhead on the total computation.

The *Geo* column considers the geodesic distance precomputation, which is not triggered until sampling. Section B.2 presents our default strategy based on the Heat Method [42] from Geometry Central [152]. 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, although we do not need to compute it again for different sampling parameters, or while editing seams and layers.

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 across regions. Similarly, the binding computation could be parallelized in theory, 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 in practice. 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 Table 6.4 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). For example, the cardigan is scheduled completely flat and allows for a trivial initial solution that helps the branch and bound exploration finish very quickly.

6.11.4 Convergence of the Optimizations

The *time function* computation is prone to local extrema. As discussed with the notion of *curvature*, 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, namely 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 behave 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.

Global Sampling

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

Local Sampling

The local steps have 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 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.

6.11.5 Subdivision Strategies

Geometric applications typically get performance improvements when modeling complex geometries by performing the user interaction on a lower-resolution mesh that is then subdivided to provide a higher-resolution result (e.g., for finer appearance modeling, or simulation). In this section, we propose a similar subdivision mechanism for our system. Because the user has a specific knitted resolution in mind, subdivision is not done with the intent of getting a higher-resolution knitted artifact. Instead, the goal is to allow for costly sampling computations to be done at a lower resolution, and then subdivided to the desired final resolution that represents the physical scale of the knitted artifact.

We initially considered two different subdivision strategies:

- *Geometry*-based – relying on the stitch graph having a geometric representation – i.e., its dual stitch mesh;

- *Sampling*-based – relying on our highly structured stitch graph to go from N to kN stitches in both course and wale directions.

While geometric subdivision has been extensively studied [32, 45, 149], it brings two issues: (1) we need to generate an appropriate dual stitch mesh on-the-fly, which may be complicated because our short-rows can have arbitrary heights and profiles, and (2) we would need to properly support non-watertight surfaces that arise from flat or mixed flat-tubular garment sketches. This latter problem becomes tricky at region interfaces since we can end up with different number of stitches that require specific care (i.e., when merging two flat sheets onto the interface of a tubular structure).

Sample-based Subdivision

Our subdivision mechanism is based on sample numbers: we multiply the stitch results from *global* and *local* sampling in Sections 6.5.1 and 6.5.2 by the subdivision factor K_{subdiv} (e.g., 2, 4 or 8). Note that we are not restricted to powers of 2.

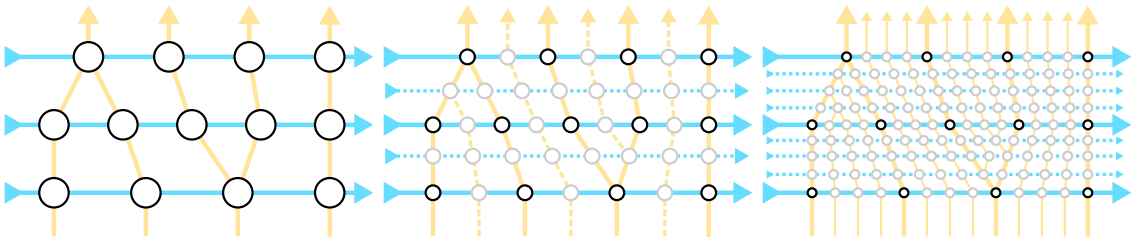


Figure 6-61: Example of graph subdivision: coarse graph (left), division by 2 (center) and division by 4 (right).

Subdividing Courses Given a course that is expected to have N stitches in the coarse version, the finer, subdivided version simply samples $K_{\text{subdiv}}N$ stitches – whether the course is circular or not.

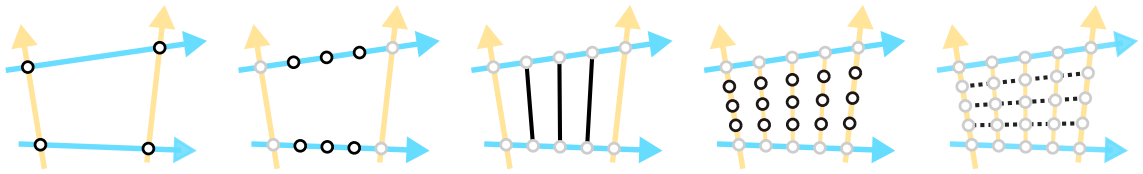
Subdividing Interfaces Internal interfaces consist in two sets of courses that need to be bound together. They are located in time on the sketch domain, and as such do not require any subdivision across the wales. To keep things simple, we use our original interface binding as is – see Section 6.5.3. We only apply it between the subdivided courses of the interfaces – i.e., we do not do coarse interface binding.

Subdividing Regions Within regions, we first go over the coarse course sampling and wale distribution. Notably, we want to be able to reuse the result of the coarse wale distribution since this one of the major interactivity bottlenecks we're facing with large stitch counts.

Given the coarse solution, we transform it into a *subdivided* version. The stitches of the coarse course pair can trivially be mapped to stitches of the subdivided course pair given their indices: from i to $K_{\text{subdiv}}i$.

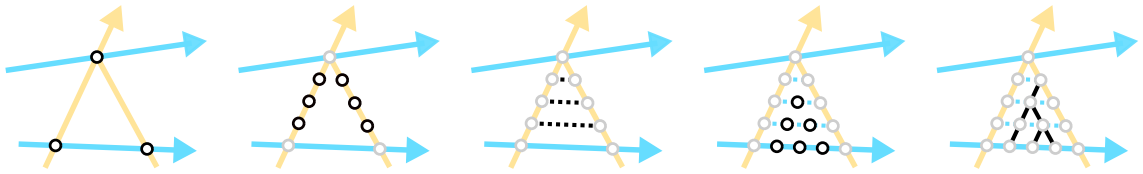
Now, we consider every adjacent coarse wale pairs. We have different cases depending on whether the wale pairs are disconnected – i.e., they form an intrinsic *quad* – or not – i.e., they form an intrinsic *triangle*. Figure 6-61 illustrates a coarse graph and its subdivided version at two different refinement scales.

If the wales are *disconnected*, then the intrinsic *quad* structure can be subdivided into $(K_{\text{subdiv}} - 1) \times (K_{\text{subdiv}} - 1)$ quads, as illustrated in Figure 6-62: (1) we map each successive subdivision stitch from the source course to that of the target course, (2)



(a) Initial coarse quad | (b) Course subdivision | (c) Geodesic wale paths | (d) Wale splitting | (e) Course connectivity

Figure 6-62: Subdivision of an intrinsic *quad*.



(a) Initial coarse triangle | (b) Wale subdivision | (c) Geodesic course paths | (d) Course splitting | (e) Wale connectivity

Figure 6-63: Subdivision of an intrinsic *triangle*.

we trace geodesic *wale* paths between these pairs, and (3) we subdivide the geodesic paths uniformly to get $K_{\text{subdiv}} - 1$ new stitches in between. The intermediate stitches form new intermediate subdivided courses.

If the wales are *connected*, then we either have a 2-1 or 1-2 wale pair corresponding to a stitch decrease, respectively decrease. Both cases represent intrinsic *triangles* and their subdivision is illustrated in Figure 6-63: (1) we subdivide each of the two coarse wales to get $K_{\text{subdiv}} - 1$ intermediate stitches, (2) we trace geodesic *course* paths for each of the pairs of intermediate course levels, (3) we subdivide the geodesic paths uniformly to get an adaptive number of new stitches along each intermediate course, and (4) the intermediate wales are distributed. The number of subdivision is linearly increased or decreased depending on whether the wale pair is an increase (1-2) or decrease (2-1). This effectively produces one irregular wale pair at each of the intermediate levels and enables us to get gradual shaping. The location of the irregular wales in each intermediate course pair can be decided with the same penalty as in the coarse wale distribution – except that the computation is purely local and can be done by exhaustive search.

Table 6.5: Evolution of the computation timings with the sketch complexity and subdivision levels. The number of stitches and instructions are provided to highlight that the subdivision does not change the final topology much for a given scale. All time measurements are in *seconds*.

Sketch	**Scale**	K_{sd}	***Stitches***	***Instr.***	***Local***	***Binding***	***Wales***	***Subdiv.***
Sweater	1mm/2px	1	19106	54163	**3.9**	**4.2**	**8.3**	—
		2	19044	52231	3.3	3.9	2.5	0.7
		4	19008	50767	2.7	3.9	0.6	0.5
	1mm/1px	1	74350	186400	**5.2**	**26.0**	**26.7**	—
		2	75222	183642	4.1	23.2	8.4	1.3
		4	74994	171942	3.2	23.1	2.5	1.2
Trousers	2mm/1px	1	36898	79283	**7.1**	**3.9**	**15.3**	—
		2	37440	77530	6.2	3.2	4.8	1.3
		4	37704	79975	5.3	3.2	1.3	1.1
	4mm/1px	1	146148	298352	**9.9**	**22.2**	**53.9**	—
		2	146220	285578	7.7	17.6	16.0	2.9
		4	148436	288007	6.1	16.6	4.8	2.7

Short-row Densities After the subdivision is done, wale splitting resumes with the subdivided stitch graph. We can either keep the short-row densities fixed for the first course (and empty for the later subdivisions), or we can diffuse them across the subdivision to get smoother short-rows – depending on a user setting.

Performance Improvements

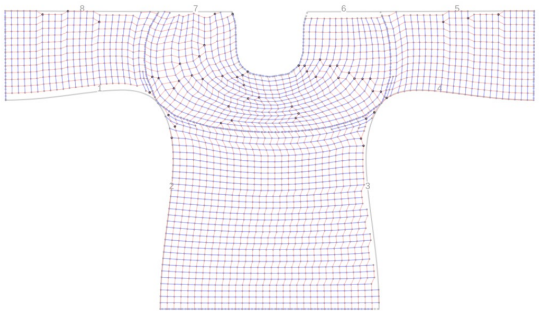
Table 6.5 provides the evolution of runtimes through finer subdivisions. Both stitch and instruction numbers are very similar across subdivisions, which suggests roughly similar sizes. Local region computations are slightly faster with subdivision, but there is no clear improvement. Interface binding is as complex, but may become slightly faster thanks to the evenness of branches. Wale distribution is the clear winner of subdivision: the operation has a seemingly quadratic computation time, and the subdivision factor thus allows quadratic time decreases, with a minimal overhead for the additional subdivision computation (last column).

Limitations

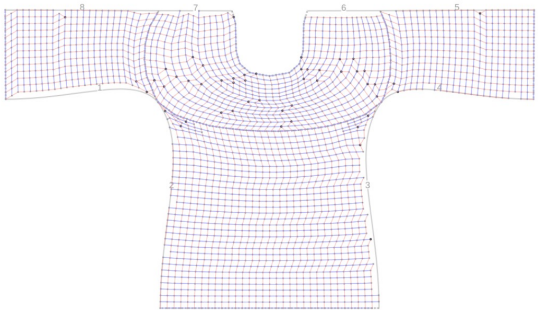
There are three notable limitations of this subdivision strategy: (1) it does not completely solve the interactivity problem, (2) it can produce geometric artifacts in high-curvature regions, and (3) it reduces the resolution of the garment in terms of sizing accuracy. Figure 6-64 illustrates a subdivison on a full sweater sample with different levels of subdivision and an example of seams to showcase that seam placement does not necessarily suffer from the resolution loss.

Interactivity: while we dealt with the issue of wale distribution – which could further be improved with parallelization –, now we have a new bottleneck with the interface binding. While we envisioned a strategy to do interface binding on the coarse graph, its translation into a subdivided version has yet to be developed.

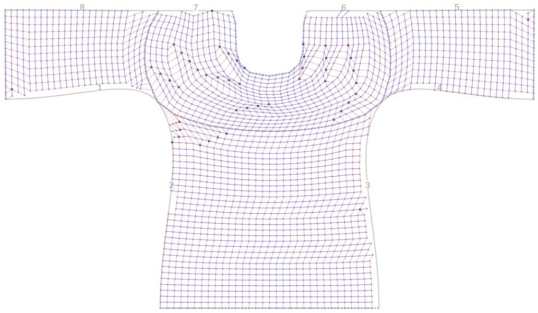
Geometric Artifacts: these are not important when we consider the shape only since the stitch embedding is not important, but they become an issue when we consider layers that are embedded on top of the sketch. The artifacts are due to the subdivision procedure for intrinsic triangles being sub-optimal when dealing with stitch irregularities in high-curvature regions.



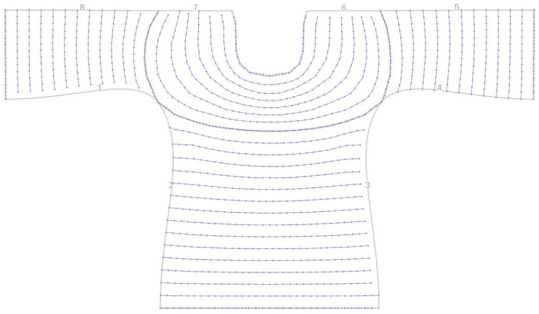
(a) No subdivision



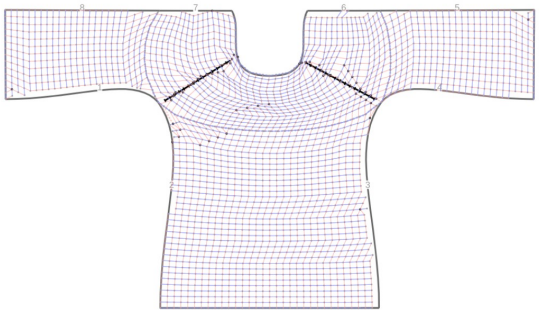
(b) $K_{\text{subdiv}} = 2$



(c) $K_{\text{subdiv}} = 4$



(d) Coarse graph ($K_{\text{subdiv}} = 2$)



(e) Seams ($K_{\text{subdiv}} = 4$)

Figure 6-64: Example of subdivision on a sweater. For all subdivision examples, the short-rows are diffused. The cases with $K_{\text{subdiv}} = 4$ show artifacts in regions of large curvature where the irregular structure (i.e., intrinsic triangles). The seam version showcases our ability to control seam placement within the subdivided irregular structures.

Chapter 7

Conclusion

This thesis presented three different high-level design tools for programming flat-bed, weft knitting machines. It proposed both CAD and CAM solutions for the corresponding three design spaces: (1) pattern programming by using image examples, (2) a parametric graph composition of shape primitives, and (3) a sketch-based workflow to translate existing cut & sew designs into knitting programs. Each of these spaces corresponds to a high-level design space. By moving from low-level programming to higher-level design abstraction, we showed that we can simplify the user interaction and enable more accessible workflows for customizing garments.

In the learning-based approach, we developed a domain-specific language for patterning, and a large dataset of real and simulated patterns with their pattern instructions. We formulated a mathematical learning framework that enabled us to effectively harness both sources of data, complementing the real – but scarce – with the synthetic – and plentiful. Our system implementation showcases high knitting program retrieval rates (around 94% accuracy on our test set). These results illustrate that images and other simple examples of garments are potent sources of knitting designs, and they do not require any design experience for the user.

Our parametric, primitive-based workflow introduced bidirectional design capability that decouple shape and patterns so that one can edit both simultaneously. Although we only considered three shape primitives, their composition permitted many interesting garment results. The digital customization was the highlight of this

workflow both from an expert perspective and through our user examples that showed unexpectedly high levels of customizability for first-time, inexperienced users.

Finally, the sketch-based workflow enabled us to reuse existing professional garment patterns we acquired from BurdaStyle and reproduce the corresponding garments in a knitted form. By augmenting the traditional sketches with knitting-specific annotations, we introduced new customization capabilities including the specification of the time process, the direction of the wale flow, and the location of shaping seams. Each of these editing capabilities were achieved at interactive framerates by relying on specific, low-dimensional representations and hierarchical, iterative optimizations.

7.1 Impact Summary

By using higher-level user inputs, we were able to (1) showcase automatic knitting program recovery, (2) enable parametric, digital customization, and (3) translate existing professional garment patterns for weft knitting, automatically. While those system instantiations were mere prototypes that have several limitations, they each enabled non-expert designers-to-be to get a step closer in the design and manufacturing of garments with weft knitting machines. Practically speaking, most of the examples in this thesis required some low-level fine-tuning before getting knitted on the machine, but they provided a large, initial amount of work that made it possible for people without knitting expertise to produce knitting programs that an expert knitting technicians would then be able to quickly fix and knit on the machine.

7.2 Future Work

The projects presented in this thesis were all research prototypes. As such, there is a large amount of work that is needed toward creating proper design tools, or integrating them into existing systems. Figure 7-1 lists some of the relevant domains involved across the general pipeline from design to fabrication – i.e., from CAD to CAM and finally the fabrication itself.

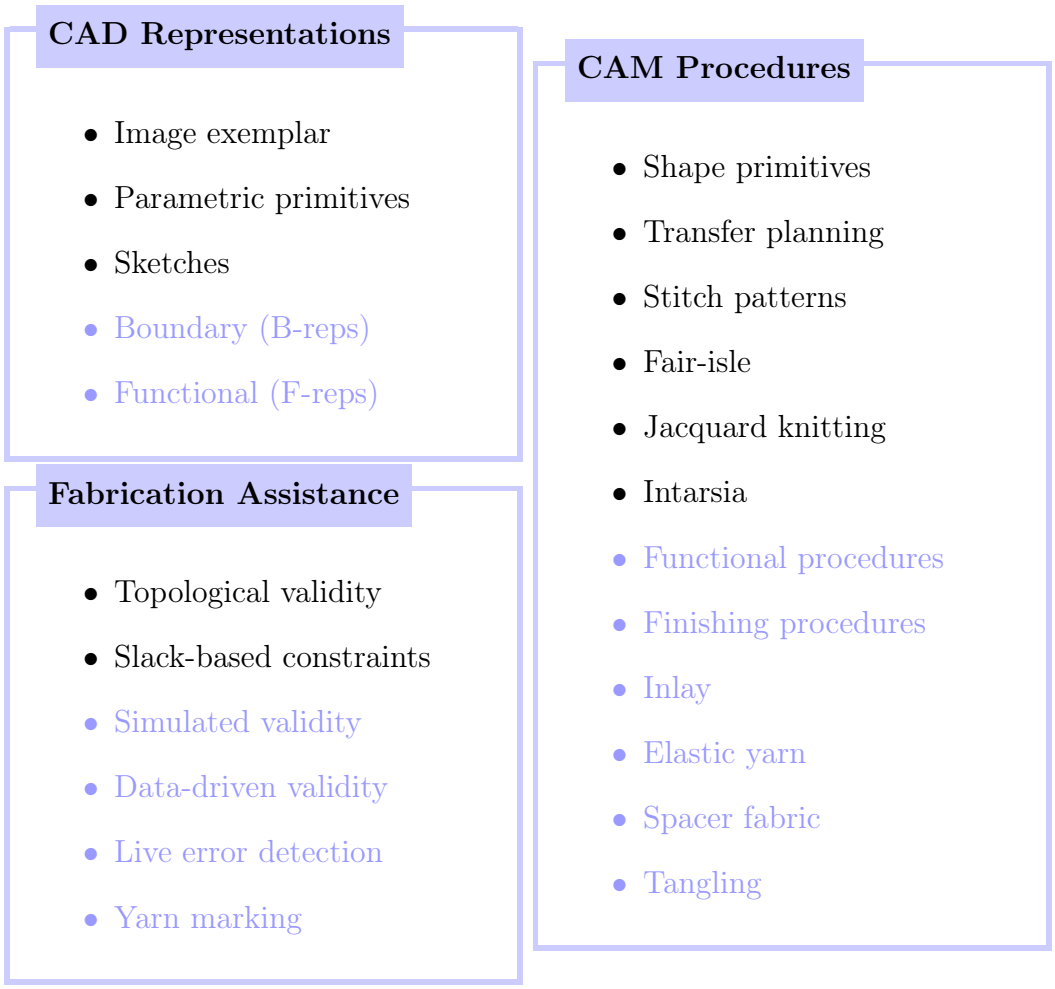


Figure 7-1: Important domains and concepts related to this thesis. The concepts we did not consider are shaded in light blue.

Design Representations: we only consider here high-level, parametric representations. This excludes string-rewriting systems and stitch meshes as explained in Section 3.3: their design spaces are too low-level and both lack the parametric aspect. This does not mean that they are not relevant, especially as long as the result of the system requires low-level fine-tuning before knitting. As of yet, the work of Narayanan et al. [123] is still one of the most complete while working with a visual representation that makes it more accessible than typical low-level knit programming. In the higher-level parametric world, *boundary representations* (B-reps) may appear as a good replacement over stitch meshes given that they are fully parametric and provide more intuitive 3D representations. However, machine knitting is typically

quite restricted in its ability to merge wale flows with sharp geometric features – e.g., the corners of a cube – and these naturally occur with boundary representations. There may be space for dedicated 3D geometric constructions targeted at knitting, but those would likely not be the ones from traditional CAD software [11, 108]. As for *functional representations* (F-reps), they model volumetric data whereas weft knitting deals mostly with geometric shells – e.g. garments on a human body.

Manufacturing Procedures: this thesis only considered basic weft knitting procedures. It did not cover more involved functional procedures that happen at the interfaces of garments parts or within specific stitch pattern for improving their mechanical properties and aesthetics: e.g., between the fingers of a glove, or before cable patterns. Similarly, we only implemented basic cast-on and cast-off procedures that do not necessarily produce nice finishing of the yarn. Beyond their use for smart textiles, some techniques used in functional fiber applications also have typical use-cases with normal garments: yarn *inlaying* is extensively used with *elastic yarn* such as in socks, and multi-layer fabrics can be programmed for garment *pockets* as well as *spacer fabric* for cushioning. Finally, while we briefly introduced some multi-yarn interactions in the last chapter, we did not look at the issues caused by yarn *tangling* – nor the potential for new design spaces it may create.

Fabrication Assistance: the feedback and constraints imposed by our systems are based on the stitch topology, together with some slack-based heuristic constraints that prevent some unreasonable transfers. As mentioned in some of our failing cases, these constraints are not sufficient. A real "manufacturability" check would likely require more complete solutions such as based on low-level yarn simulation, or using data-driven tests to learn what is acceptable or not to knit. Another important aspect is how we can help the user figure out what fails when something goes wrong on the knitting machine. Because of the complexity of the stitch interactions, many failures are currently not easily reproducible. This makes debugging a long trial-and-error process which could be help in several ways. Two notable directions include: (1)

the development of live error detection such as with machine vision, and (2) the usage of yarn markers to facilitate backtracking from errors in the final artifact to their locations in the digital program. This could potentially be done by dying (or coating) the yarn in some programmatic way.

7.2.1 Learning-based Workflow

Machine learning and data-driven methods have recently flourished in various domains. Yet, a lot is still needed for automating manufacturing processes and making them more accessible. Related to knitting program synthesis, the main directions we foresee include (1) the integration of unsupervised learning as part of the learning framework, (2) the development of more expressive simulations, and (3) new hierarchical representations of the knit structure.

More diverse data. Our learning framework is the first to tackle the automatic inference of knitting programs from images. However, in comparison to existing large-scale learning systems [143], we rely on a knitting dataset that is several orders of magnitude smaller. Our data lacks yarn diversity and only considers single-yarn knitting patterns. Given the existence of large stitch pattern collections [49, 154], it would be very useful if we can include some form of unsupervised learning as part of our framework and harness the large amounts of images of unstructured knitting patterns in the wild. For the synthetic data supplement, we relied on a black-box knitting simulation from Shima [150]. The resulting synthetic data could be improved given novel state-of-the-art knitting simulations [86, 87, 97, 185, 191].

More complex models. The resulting knitting programs are made knittable by ad-hock post-processing rules and no modeling of the knittability or knit structure is explicitly integrated as part of the current framework. While we only consider simple 20×20 knitting patterns, many interesting patterns span larger numbers of stitches. We envision that the knitting program structure could be augmented with hierarchical interactions to model the large-scale interaction between some of the knitting

instructions – notably moves and cables. A hierarchical representation [101, 157] may enable us to work with larger patterns, or remove the fixed pattern size constraint completely. Finally, a large body of future work is necessary to deal with the initial problem at hand: inferring large-scale knitting programs for whole garments. Both novel datasets and garment representation will be necessary. Our work may have actually started two of these: a primitive-based skeleton graph for garment composition, and a sketch-based representation.

7.2.2 Primitives-based Workflow

Parametric CAD/CAM software [11, 94, 108] has been developed over many decades and is still largely evolving. While our shape primitives showcase the potential to develop something similar for garments, future work is still plentiful. Notable critical components we envision include (1) additional shape primitives, (2) the parameterization of low-level knitting procedures, and (3) a form of user-editable scheduling.

Novel primitives. Our skeleton graph enabled an initial, parametric model of many types of garments, while relying on only three base shape primitives. There is no constraint on allowing more primitives. In fact, we envision that more complex glueing operations could be integrated as part of dedicated primitives. The complexity in designing these lies in a judicious choice of the primitives. It would be wasteful to rely merely on a collection of templates if these are not worth composing.

Higher-level customization. While we only showcase 2D patterns for customization, the integration of existing face-based programming [123] may provide a first extension to support colorwork and other multi-yarn interactions. Tunable scheduling procedure [121] may also further enable this. One subtle design issue is that of scale. While enabling larger garment scales is likely a computational problem that can be solved with engineering, large-scale design will require different representations of customization itself. Working with individual pixels may be satisfying for simple structures but does not scale well if we need to apply it to hundreds of thousands –

if not millions of – stitches. Pattern interaction must happen at a higher level: i.e., not rely on per-stitch drawing, but work at per-pattern or sub-pattern levels. Similarly, the interaction between patterns upon editing of the shape is currently mainly focused on maintaining access to the pattern data, not in how such patterns may need to evolve. Smart knitting patterns [71] that use data-driven methods to adapt the patterns may provide an elegant solution. Finally, a critical, missing component is a form of parameterization of low-level knitting procedures. While Nader et al. [121] rely on procedures designed by experts for customizing their schedules, it is unclear how to fully parameterize the space of low-level procedures in a way that is *composable*, *verifiable* and *accessible* to users.

7.2.3 Sketch-based Workflow

Our workflow allows the translation of professional garment patterns into knitting programs. Yet, existing garment workflows provide many interaction capabilities beyond what we displayed, such as different forms of binding, the grading of patterns, and physical tangible manipulation that allows low-level customization. Similarly, we envision that our sketch workflow can be extended in many important ways: (1) using novel forms of shape representation, (2) integrating with two-view garment design tools, and (3) parameterizing sketches from scratch to enable automatic grading and more general customization.

Representing the knitted shape. Our interactive time function editing capabilities provide one means of customizing the knitting flow. Unfortunately, it makes low-curvature assumptions that prevent some forms of garments from being easily modeled – e.g., socks. While it supports a form of seam editing a posteriori, seams typically come from large flows that merge or split and are thus inherently discontinuous. A complete solution would support locally large curvature – e.g., local aggregations of short-rows or seam discontinuities. In terms of workflow, the tuning of regions tend to be more important than the time function itself, and thus one may wonder if delineating and composing regions instead of a time function may be a simpler, more

robust strategy. Given that the underlying region graph is semantically similar to our parametric skeleton, it may in fact be possible to instead compose shape primitives over the sketch atlas. Regarding the resulting stitch graph, we currently allow indirect seam editing. Yet, one may often want to control other low-level topology properties such as to specify short-rows or make specific regions symmetric.

Integration with 3D simulation. Our system only takes care of the implicit geometry whereas full professional garment editing tools provide both 2D manufacturing and 3D simulation views side-by-side [25, 39, 113]. We envision that our system could be integrated within similar systems, extending their CAM capabilities to whole-garment weft knitting machines. In terms of customization, we could provide direct editing of the underlying stitch graph such as is done with stitch meshes [123, 187], but this may lead to workflow issues if those editing operations get lost as the user makes changes to the shape. Furthermore, those do not necessarily match existing digital workflows based on image layers for customization. Similarly to complex cut & sew garments, knitted garments often include multiple layers of yarn – e.g., colorwork, structural yarn patterns, inlay, or even multi-layer pockets and spacer fabric – and modeling these is critical for adoption.

Parametric sketches. Traditional garment patterns often model more than just a single garment. They typically provide graded curves to produce various sizes of the same garment. Grading is an important part of garment modeling [119, 148] and we envision that parametric sketches could go beyond by providing per-user tunable designs. This is however more complicated than just providing parametric sketch editing. The core issue is in the underlying parametric representation to enable proper size customization capabilities[1], while also taking into account the need for user customization of the corresponding sketch garment.

[1]See for example the work and community started by Joost De Cock at http://freesewing.org

Appendix A

Proofs and Definitions

Proof of Theorem 1

We first describe the necessary definitions and lemmas to prove Theorem 1. We need a general way to measure the discrepancy between two distributions, which we borrow from the definition of discrepancy suggested by [111].

Definition 1 (Discrepancy [111]). *Let $\mathcal{H}$ be a class of functions mapping from $\mathcal{X}$ to $\mathcal{Y}$. The discrepancy between two distribution $\mathcal{D}_1$ and $\mathcal{D}_2$ over $\mathcal{X}$ is defined as*

$$\mathrm{disc}_{\mathcal{H}}(\mathcal{D}_1, \mathcal{D}_2) = \max_{h,h' \in \mathcal{H}} \left| \mathcal{L}_{\mathcal{D}_1}(h, h') - \mathcal{L}_{\mathcal{D}_2}(h, h') \right|. \tag{A.1}$$

The discrepancy is symmetric and satisfies the triangle inequality, regardless of any loss function. This can be used to compare distributions for general tasks even including regression.

The following lemma is the extension of Lemma 4 in [19] to be generalized by the above discrepancy.

Lemma 1. *Let h be a hypothesis in class $\mathcal{H}$, and assume that $\mathcal{L}$ is symmetric and obeys the triangle inequality. Then*

$$|\mathcal{L}_\alpha(h,y) - \mathcal{L}_T(h,y)| \leq \alpha \left(\mathrm{disc}_{\mathcal{H}}(\mathcal{D}_S, \mathcal{D}_T) + \lambda\right), \tag{A.2}$$

where λ=$\mathcal{L}_S(h^*,y)$+$\mathcal{L}_T(h^*,y)$, and the ideal joint hypothesis h^* is defined as h^*= $\arg\min_{h\in\mathcal{H}} \mathcal{L}_S(h,y)$+$\mathcal{L}_T(h,y)$.

Proof. The proof is based on the triangle inequality of $\mathcal{L}$, and the last inequality follows the definition of the discrepancy.

$$\begin{aligned}
&|\mathcal{L}_\alpha(h,y) - \mathcal{L}_T(h,y)| \\
=&\alpha|\mathcal{L}_S(h,y) - \mathcal{L}_T(h,y)| \\
=&\alpha\,|\mathcal{L}_S(h,y) - \mathcal{L}_S(h^*,h) + \mathcal{L}_S(h^*,h) \\
&- \mathcal{L}_T(h^*,h) + \mathcal{L}_T(h^*,h) - \mathcal{L}_T(h,y)\,| \\
\leq&\alpha\big|\,|\mathcal{L}_S(h,y) - \mathcal{L}_S(h^*,h)| + \\
&|\mathcal{L}_S(h^*,h) - \mathcal{L}_T(h^*,h)| + |\mathcal{L}_T(h^*,h) - \mathcal{L}_T(h,y)|\,\big| \\
\leq&\alpha\big|\mathcal{L}_S(h^*,y)+|\mathcal{L}_S(h^*,h)-\mathcal{L}_T(h^*,h)|+\mathcal{L}_T(h^*,y)\big| \\
\leq&\alpha\left(\mathrm{disc}_{\mathcal{H}}(\mathcal{D}_S, \mathcal{D}_T) + \lambda\right).
\end{aligned} \tag{A.3}$$

We conclude the proof. □

Many types of losses satisfy the triangle inequality: e.g., the $0-1$ loss [19, 41] and l_1-norm obey the triangle inequality, and l_p-norm ($p > 1$) obeys the pseudo triangle inequality [53].

Lemma 1 bounds the difference between the target loss and α-mixed loss. In order to derive the relationship between a true expected loss and its empirical loss, we rely on the following lemma.

Lemma 2 ([19]). *For a fixed hypothesis h, if a random labeled sample of size m is generated by drawing βm points from $\mathcal{D}_S$ and $(1-\beta)m$ points from $\mathcal{D}_T$, and labeling them according to y_S and y_T respectively, then for any $\delta \in (0,1)$, with probability at least $1-\delta$ (over the choice of the samples),*

$$|\hat{\mathcal{L}}_\alpha(h,y) - \mathcal{L}_\alpha(h,y)| \leq \epsilon(m,\alpha,\beta,\delta), \tag{A.4}$$

where $\epsilon(m,\alpha,\beta,\delta) = \sqrt{\frac{1}{2m}\left(\frac{\alpha^2}{\beta} + \frac{(1-\alpha)^2}{1-\beta}\right)\log(\frac{2}{\delta})}$.

The detail function form of ϵ will be omitted for simplicity. We can fix m, α, β, and δ when the learning task is specified, then we can treat $\epsilon(\cdot)$ as a constant.

Theorem 1. *Let $\mathcal{H}$ be a hypothesis class, and $\mathcal{S}$ be a labeled sample of size m generated by drawing βm samples from $\mathcal{D}_S$ and $(1-\beta)m$ samples from $\mathcal{D}_T$ and labeling them according to the true label y. Suppose $\mathcal{L}$ is symmetric and obeys the triangle inequality. Let $\hat{h} \in \mathcal{H}$ be the empirical minimizer of $\hat{h} = \arg\min_h \hat{\mathcal{L}}_\alpha(h,y)$ on $\mathcal{S}$ for a fixed $\alpha \in [0,1]$, and $h_T^* = \arg\min_h \mathcal{L}_T(h,y)$ the target error minimizer. Then, for any $\delta \in (0,1)$, with probability at least $1-\delta$ (over the choice of the samples), we have*

$$\tfrac{1}{2}|\mathcal{L}_T(\hat{h},y) - \mathcal{L}_T(h_T^*,y)| \leq \alpha\left(\mathrm{disc}_{\mathcal{H}}(\mathcal{D}_S,\mathcal{D}_T) + \lambda\right) + \epsilon, \tag{A.5}$$

where $\epsilon(m,\alpha,\beta,\delta) = \sqrt{\frac{1}{2m}\left(\frac{\alpha^2}{\beta} + \frac{(1-\alpha)^2}{1-\beta}\right)\log(\frac{2}{\delta})}$, *and* $\lambda = \min_{h\in\mathcal{H}} \mathcal{L}_S(h,y) + \mathcal{L}_T(h,y)$.

Proof. We use Lemmas 1 and 2 for the bound derivation with their associated assumptions.

$$
\begin{aligned}
&\mathcal{L}_T(\hat{h}, y) \\
&\leq \mathcal{L}_\alpha(\hat{h}, y) + \alpha \left(\mathrm{disc}_{\mathcal{H}}(\mathcal{D}_S, \mathcal{D}_T) + \lambda\right), && \text{(A.6)} \\
& && \text{(By Lemma 1)} \\
&\leq \hat{\mathcal{L}}_\alpha(\hat{h}, y) + \alpha \left(\mathrm{disc}_{\mathcal{H}}(\mathcal{D}_S, \mathcal{D}_T) + \lambda\right) + \epsilon, && \text{(A.7)} \\
& && \text{(By Lemma 2)} \\
&\leq \hat{\mathcal{L}}_\alpha(h_T^*, y) + \alpha \left(\mathrm{disc}_{\mathcal{H}}(\mathcal{D}_S, \mathcal{D}_T) + \lambda\right) + \epsilon, && \text{(A.8)} \\
& && (\hat{h} = \underset{h \in \mathcal{H}}{\arg\min}\, \hat{\mathcal{L}}_\alpha(h)) \\
&\leq \mathcal{L}_\alpha(h_T^*, y) + \alpha \left(\mathrm{disc}_{\mathcal{H}}(\mathcal{D}_S, \mathcal{D}_T) + \lambda\right) + 2\epsilon, && \text{(A.9)} \\
& && \text{(By Lemma 2)} \\
&\leq \mathcal{L}_T(h_T^*, y) + 2\alpha \left(\mathrm{disc}_{\mathcal{H}}(\mathcal{D}_S, \mathcal{D}_T) + \lambda\right) + 2\epsilon, && \text{(A.10)} \\
& && \text{(By Lemma 1)}
\end{aligned}
$$

which concludes the proof. □

Theorem 1 does not have unnecessary dependencies for our purpose, which are used in [19] such as unsupervised data and the restriction of the model type to finite VC-dimensions.

Appendix B

Implementation Details

This appendix provides implementations details of the hierarchical sampling algorithms presented in Section 6.5.

B.1 Solving the 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 [84] library. We use the Improved Augmented Lagrangian Method [22] as global solver, and L-BFGS [105] 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.

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:

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

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, while removing all equality constraints and adding a small number of inequality constraints.

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 $(\mathbf{e}_a, \mathbf{e}_b) \in \mathcal{R}$.

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 $(\mathcal{S}_i, \mathcal{S}_j) \in \mathcal{A}$:

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

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

$$N^* = |t(\mathbf{e}_b) - t(\mathbf{e}_a)|/D_{\text{wale}}. \tag{B.1}$$

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.

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:

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

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.

B.2 Affordable 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 [42] 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. [162]. To further restrict the search space during the edge-window propagation, we only consider the neighborhoods adjacent to our initial approximate geodesic path.

B.3 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.

B.3.1 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.

B.3.2 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.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[2] Colin C Adams. *The Knot Book*. American Mathematical Soc., 1994.

[3] Sabit Adanur. *Handbook of Weaving*. CRC press, 2020.

[4] Adobe Inc. Adobe Photoshop, March 2019. URL https://www.adobe.com/products/photoshop.html. [Online; Accessed: 08-16-2021].

[5] Michael Agnes and David Bernard Guralnik. *Webster's New World College Dictionary*. Macmillan New York, 1999.

[6] C.L. Ahles. *Fine Machine Sewing: Easy Ways to Get the Look of Hand Finishing and Embellishing*. Taunton Press, 2001. ISBN 978-1-56158-487-1. URL https://books.google.com/books?id=ft_qCrwc_BYC.

[7] Lea Albaugh, Scott Hudson, and Lining Yao. Digital Fabrication of Soft Actuated Objects by Machine Knitting. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, CHI '19, pages 1–13, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 978-1-4503-5970-2.

[8] W. Albrecht, H. Fuchs, and W. Kittelmann. *Nonwoven Fabrics: Raw Materials, Manufacture, Applications, Characteristics, Testing Processes*. Wiley, 2006. ISBN 978-3-527-60531-6. URL https://books.google.com/books?id=pvQwXBi3HwMC.

[9] Nikolay Anguelov. *The Dirty Side of the Garment Industry: Fast Fashion and Its Negative Impact on Environment and Society*. CRC Press, 2015.

[10] Michael Ashikmin, Simon Premože, and Peter Shirley. A microfacet-based BRDF generator. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 65–74, 2000.

[11] Autodesk, INC. Fusion 360, August 2021. URL https://www.autodesk.com/products/fusion-360/. [Online; Accessed: 08-16-2021].

[12] Firas Awaja and Dumitru Pavel. Recycling of PET. *European polymer journal*, 41(7):1453–1477, 2005.

[13] Cagri Ayranci and Jason Carey. 2D braided composites: A review for stiffness critical applications. *Composite Structures*, 85(1):43–58, 2008.

[14] Tavmjong Bah. *Inkscape: Guide to a Vector Drawing Program*. prentice hall press, 2007.

[15] Alfred Barlow. *The History and Principles of Weaving by Hand and by Power*. Low, Marston, Searle, and Rivington, 1878.

[16] Robert K Barnhart. *The Barnhart Dictionary of Etymology*. New York: HW Wilson Company, 1988.

[17] Aric Bartle, Alla Sheffer, Vladimir G. Kim, Danny M. Kaufman, Nicholas Vining, and Floraine Berthouzoz. Physics-Driven Pattern Adjustment for Direct 3D Garment Editing. *ACM Trans. Graph.*, 35(4), July 2016. ISSN 0730-0301.

[18] Sean Bell, Paul Upchurch, Noah Snavely, and Kavita Bala. Material recognition in the wild with the materials in context database. *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[19] Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. A theory of learning from different domains. *Machine learning*, 79(1-2):151–175, 2010.

[20] Floraine Berthouzoz, Akash Garg, Danny M. Kaufman, Eitan Grinspun, and Maneesh Agrawala. Parsing Sewing Patterns into 3D Garments. *ACM Trans. Graph.*, 32(4), July 2013. ISSN 0730-0301.

[21] Rachel Bick, Erika Halsey, and Christine C Ekenga. The global environmental injustice of fast fashion. *Environmental Health*, 17(1):1–4, 2018.

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

[23] Ronald V Book and Friedrich Otto. String-rewriting systems. In *String-Rewriting Systems*, pages 35–64. Springer, 1993.

[24] Brother. Brother Sewing and Embroidery Machines, 2021. URL https://www.brother-usa.com/home/sewing-embroidery. [Online; Accessed: 2021-08-19].

[25] Browzwear. Browzwear VStitcher, 2009. URL http://www.browzwear.com/vstitcher.htm. [Online; Accessed: 08-27-2009].

[26] H.D. Buck. *Flat Machine Knitting and Fabrics*. Bragdon, Lord & Nagle Company, 1921. URL https://books.google.com/books?id=1xsjkgAACAAJ.

[27] Ann Budd. *Knitter's Handy Book of Top-Down Sweaters: Basic Designs in Multiple Sizes and Gauges*. Interweave, 2012. ISBN 978-1-59668-483-6.

[28] Butterick Publishing Inc. *The Art of Knitting (Dover Knitting, Crochet, Tatting, Lace)*. Dover Publications, 2016.

[29] Michele Calì, Salvatore Massimo Oliveri, Ubaldo Cella, Massimo Martorelli, Antonio Gloria, and Domenico Speranza. Mechanical characterization and modeling of downwind sailcloth in fluid-structure interaction analysis. *Ocean Engineering*, 165:488–504, 2018.

[30] Denis DR Cartie, Giuseppe Dell'Anno, Emilie Poulin, and Ivana K Partridge. 3D reinforcement of stiffener-to-skin T-joints by Z-pinning and tufting. *Engineering fracture mechanics*, 73(16):2532–2540, 2006.

[31] E. Castro-Aguirre, F. Iñiguez-Franco, H. Samsudin, X. Fang, and R. Auras. Poly(lactic acid)—Mass production, processing, industrial applications, and end of life. *Advanced Drug Delivery Reviews*, 107:333–366, 2016. ISSN 0169-409X. doi: 10.1016/j.addr.2016.03.010. URL https://www.sciencedirect.com/science/article/pii/S0169409X16300965. PLA biodegradable polymers.

[32] Edwin Catmull and James Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-aided design*, 10(6):350–355, 1978.

[33] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 40(4):834–848, 2018.

[34] Maurice Chiodo. An introduction to braid theory. *Msc, University of Melbourne*, 2005.

[35] Wonseok Choi and Nancy B Powell. Three dimensional seamless garment knitting on V-bed flat knitting machines. *Journal of Textile and Apparel, Technology and Management*, 4(3):1–33, 2005.

[36] PL Chu and T Whitbread. Measurement of stresses in optical fiber and preform. *Applied Optics*, 21(23):4241–4245, 1982.

[37] Deborah DL Chung and Deborah Chung. *Carbon Fiber Composites*. Elsevier, 2012.

[38] K.S. Clair. *The Golden Thread: How Fabric Changed History*. Liveright, 2019. ISBN 978-1-63149-636-3. URL https://books.google.com/books?id=VweLDwAAQBAJ.

[39] Clo3D. Clo3D, 2020. URL https://www.clo3d.com/. [Online; Accessed: 2021-08-19].

[40] Isabel De Nyse Conover. *A Complete Course in Dressmaking, (Vol. 8, Draping and Pattern Making)*. New York, E. J. Clode, 1922. URL http://archive.org/details/completecoursein08cono.

[41] Koby Crammer, Michael Kearns, and Jennifer Wortman. Learning from multiple sources. *Journal of Machine Learning Research*, 9(Aug):1757–1774, 2008.

[42] Keenan Crane, Clarisse Weischedel, and Max Wardetzky. The Heat Method for Distance Computation. *Commun. ACM*, 60(11):90–99, October 2017. ISSN 0001-0782.

[43] Douglas Crockford. *JavaScript: The Good Parts: The Good Parts*. " O'Reilly Media, Inc.", 2008.

[44] Patrick Davison. Because of the Pixels: On the History, Form, and Influence of MS Paint. *Journal of Visual Culture*, 13(3):275–297, 2014. doi: 10.1177/1470412914544539. URL https://doi.org/10.1177/1470412914544539.

[45] Fernando de Goes, Mathieu Desbrun, Mark Meyer, and Tony DeRose. Subdivision exterior calculus for geometry processing. *ACM Trans. Graph.*, 35 (4), July 2016. ISSN 0730-0301. doi: 10.1145/2897824.2925880. URL https://doi.org/10.1145/2897824.2925880.

[46] Philippe Decaudin, Dan Julius, Jamie Wither, Laurence Boissieux, Alla Sheffer, and Marie-Paule Cani. Virtual Garments: A Fully Geometric Approach for Clothing Design. *Computer Graphics Forum*, 25(3):625–634, 2006.

[47] Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdelrahman Mohamed, and Pushmeet Kohli. RobustFill: Neural Program Learning under Noisy I/O. In *International Conference on Machine Learning*, 2017.

[48] Thomas G Dietterich, Richard H Lathrop, and Tomás Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial intelligence*, 89(1-2):31–71, 1997.

[49] Nanette Donohue. *750 Knitting Stitches: The Ultimate Knit Stitch Bible*. REED BUSINESS INFORMATION 360 PARK AVENUE SOUTH, NEW YORK, NY 10010 USA, 2015.

[50] Thomas Dublin. *Women at Work. The Transformation of Work and Community in Lowell, Massachusetts, 1826–1860*. Columbia University Press, 1979.

[51] J. Essinger. *Jacquard's Web: How a Hand-Loom Led to the Birth of the Information Age*. OUP E-Books. OUP Oxford, 2007. ISBN 978-0-19-280578-2. URL https://books.google.com/books?id=zXoRDAAAQBAJ.

[52] S. Fouchier. *Felt*. Textiles Handbooks. A&C Black, 2009. ISBN 978-0-7136-8494-0. URL https://books.google.com/books?id=bcUbC60NQbsC.

[53] Tomer Galanti and Lior Wolf. A Theory of Output-Side Unsupervised Domain Adaptation. *arXiv:1703.01606*, 2017.

[54] Yaroslav Ganin, Evgeniya Ustinova, Hana Ajakan, Pascal Germain, Hugo Larochelle, François Laviolette, Mario Marchand, and Victor Lempitsky. Domain-adversarial training of neural networks. *Journal of Machine Learning Research*, 17(1):2096–2030, 2016.

[55] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[56] N. Gokarneshan, B. Varadarajan, and C.B. Senthil Kumar. 4 - Types of cams in textile and their design. In N. Gokarneshan, B. Varadarajan, and C.B. Senthil Kumar, editors, *Mechanics and Calculations of Textile Machinery*, pages 66–80. Woodhead Publishing India, 2013. ISBN 978-0-85709-104-8. doi: 10.1533/9780857095527.1.66. URL https://www.sciencedirect.com/science/article/pii/B9780857091048500043.

[57] R.H. Gong. *Specialist Yarn and Fabric Structures: Developments and Applications*. Woodhead Publishing Series in Textiles. Elsevier Science, 2011. ISBN 978-0-85709-393-6. URL https://books.google.com/books?id=KQ5IAgAAQBAJ.

[58] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, 2014.

[59] S. Gordon and Y.L. Hsieh. *Cotton: Science and Technology*. Woodhead Publishing Series in Textiles. Elsevier Science, 2006. ISBN 978-1-84569-248-3. URL https://books.google.com/books?id=VsBQAwAAQBAJ.

[60] Peng Guan, Loretta Reiss, David A. Hirshberg, Alexander Weiss, and Michael J. Black. DRAPE: DRessing Any PErson. *ACM Trans. Graph.*, 31(4), July 2012. ISSN 0730-0301.

[61] Alexander Gumennik, Alexander M Stolyarov, Brent R Schell, Chong Hou, Guillaume Lestoquoy, Fabien Sorin, William McDaniel, Aimee Rose, John D Joannopoulos, and Yoel Fink. All-in-fiber chemical sensing. *Advanced Materials*, 24(45):6005–6009, 2012.

[62] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1321–1330. JMLR. org, 2017.

[63] Runbo Guo, Jenny Lin, Vidya Narayanan, and James McCann. Representing crochet with stitch meshes. In *Symposium on Computational Fabrication*, SCF '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-8170-3. doi: 10.1145/3424630.3425409. URL https://doi.org/10.1145/3424630.3425409.

[64] Yuanyuan Guo, Shan Jiang, Benjamin JB Grena, Ian F Kimbrough, Emily G Thompson, Yoel Fink, Harald Sontheimer, Tatsuo Yoshinobu, and Xiaoting Jia. Polymer composite with carbon nanofibers aligned during thermal drawing as a microelectrode for chronic neural interfaces. *Acs Nano*, 11(7):6574–6585, 2017.

[65] J Hagewood. Technologies for the manufacture of synthetic polymer fibers. In *Advances in Filament Yarn Spinning of Textiles and Polymers*, pages 48–71. Elsevier, 2014.

[66] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[67] Francesann L. Heisey, Peter Brown, and Robert F. Johnson. Three-dimensional pattern drafting: A theoretical framework. *Clothing and Textiles Research Journal*, 6(3):1–9, 1988. doi: 10.1177/0887302X8800600301. URL https://doi.org/10.1177/0887302X8800600301.

[68] Anamaría Henao, Marco Carrera, Antonio Miravete, and Luis Castejón. Mechanical performance of through-thickness tufted sandwich structures. *Composite structures*, 92(9):2052–2059, 2010.

[69] David W Henderson and Daina Taimina. Crocheting the hyperbolic plane. *The Mathematical Intelligencer*, 23(2):17–28, 2001.

[70] Judy Hoffman, Eric Tzeng, Taesung Park, Jun-Yan Zhu, Phillip Isola, Kate Saenko, Alexei A Efros, and Trevor Darrell. Cycada: Cycle-consistent adversarial domain adaptation. In *International Conference on Machine Learning*, 2018.

[71] Megan Hofmann, Lea Albaugh, Ticha Sethapakadi, Jessica Hodgins, Scott Hudson, Jame McCann, and Jennifer Mankoff. KnitPicking Textures: Programming and Modifying Complex Knitted Textures for Machine and Hand Knitting. *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, pages 5–16, 2019.

[72] Chen Huang, Yining Li, Chen Change Loy, and Xiaoou Tang. Learning Deep Representation for Imbalanced Classification. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[73] Ping Huang, Junfeng Yao, and Hengheng Zhao. Automatic realistic 3D garment generation based on two images. In *2016 International Conference on Virtual Reality and Visualization (ICVRV)*, pages 250–257. IEEE, 2016.

[74] Scott E. Hudson. Printing teddy bears: A technique for 3D printing of soft interactive objects. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 459–468, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2557338. URL https://doi.org/10.1145/2556288.2557338.

[75] D. Hunter. *Papermaking: The History and Technique of an Ancient Craft*. Dover Books Explaining Science. Dover Publications, 1978. ISBN 978-0-486-23619-3. URL https://books.google.com/books?id=1sEp3rtK994C.

[76] Janet Hunter. *Women and the Labour Market in Japan's Industrialising Economy: The Textile Industry before the Pacific War*. Routledge, 2004.

[77] Takeo Igarashi and John F. Hughes. Clothing Manipulation. *ACM Trans. Graph.*, 22(3):697, July 2003. ISSN 0730-0301.

[78] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. Image-to-image translation with conditional adversarial networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[79] F.H. Jackson. *Intarsia and Marquetry*. Handbooks for the Designer and Craftsman. Sands, 1903. URL https://books.google.com/books?id=B0JLAAAAMAAJ.

[80] Wenzel Jakob, Marco Tarini, Daniele Panozzo, and Olga Sorkine-Hornung. Instant Field-Aligned Meshes. *ACM Trans. Graph.*, 34(6), October 2015. ISSN 0730-0301.

[81] C. James. *The Complete Serger Handbook*. A Sterling/Sewing Information Resources Book. Sterling Publishing Company, Incorporated, 1998. ISBN 978-0-8069-9807-7. URL https://books.google.com/books?id=MjgOSEEjxnkC.

[82] Justin Johnson, Alexandre Alahi, and Li Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In *European Conference on Computer Vision*, 2016.

[83] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Judy Hoffman, Li Fei-Fei, C Lawrence Zitnick, and Ross Girshick. Inferring and executing programs for visual reasoning. In *IEEE International Conference on Computer Vision*, 2017.

[84] Steven G Johnson. The NLopt nonlinear-optimization package, 2014. URL http://github.com/stevengj/nlopt. [Online; Accessed: 08-16-2021].

[85] S.J. Kadolph. *Textiles*. Fashion Series. Pearson, 2010. ISBN 978-0-13-500759-4. URL https://books.google.com/books?id=vsO9QQAACAAJ.

[86] Jonathan Kaldor. *Simulating Yarn-Based Cloth*. PhD Thesis, Cornell University, 2011.

[87] Jonathan M. Kaldor, Doug L. James, and Steve Marschner. Simulating knitted cloth at the yarn level. In *ACM Transactions on Graphics (TOG)*, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 978-1-4503-0112-1. doi: 10.1145/1399504.1360664. URL https://doi.org/10.1145/1399504.1360664.

[88] Mehmet Kanik, Sirma Orguc, Georgios Varnavides, Jinwoo Kim, Thomas Benavides, Dani Gonzalez, Timothy Akintilo, C Cem Tasan, Anantha P Chandrakasan, Yoel Fink, et al. Strain-programmable fiber-based artificial muscle. *Science*, 365(6449):145–150, 2019.

[89] Neel Kant. Recent Advances in Neural Program Synthesis. *arXiv:1802.02353*, 2018.

[90] Alexandre Kaspar, Liane Makatura, and Wojciech Matusik. Knitting Skeletons: A Computer-Aided Design Tool for Shaping and Patterning of Knitted Garments. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*, pages 53–65, New Orleans, Louisiana, USA, October 2019. ISBN 978-1-4503-6816-2.

[91] Alexandre Kaspar, Tae-Hyun Oh, Liane Makatura, Petr Kellnhofer, and Wojciech Matusik. Neural Inverse Knitting: From Images to Manufacturing Instructions. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 3272–3281, Long Beach, California, USA, June 2019. PMLR. URL http://proceedings.mlr.press/v97/kaspar19a.html.

[92] Alexandre Kaspar, Kui Wu, Yiyue Luo, Liane Makatura, and Wojciech Matusik. Knit Sketching: From Cut & Sew Patterns to Machine-Knit Garments. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 40(4), 2021.

[93] F Selcen Kilinc. *Handbook of Fire Resistant Textiles*. Woodhead Publishing Series in Textiles. Elsevier Science, 2013. ISBN 978-0-85709-893-1. URL https://books.google.com/books?id=dGVEAgAAQBAJ.

[94] Marius Kintel and Clifford Wolf. OpenSCAD, the programmers solid 3D CAD modeller, 2017. URL https://openscad.org/. [Online; Accessed: 2021-08-16].

[95] Berthold Laufer. The early history of felt. *American Anthropologist*, 32(1):1–18, 1930.

[96] James Laver. *Costume and Fashion: A Concise History (World of Art)*. Thames & Hudson, 2020.

[97] Jonathan Leaf, Rundong Wu, Eston Schweickart, Doug L. James, and Steve Marschner. Interactive Design of Yarn-Level Cloth Patterns. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2018)*, 37(6), November 2018. doi: 10.1145/3272127.3275105.

[98] Olivier Lecarme and Karine Delvare. *The Book of GIMP: A Complete Guide to Nearly Everything*. No Starch Press, 2013.

[99] Beverly Lemire. Draping the body and dressing the home: The material culture of textiles and clothes in the Atlantic world, c. 1500–1800. In *History and Material Culture*, pages 89–105. Routledge, 2017.

[100] Jenny Li and James McCann. An Artin Braid Group Representation of Knitting Machine State with Applications to Validation and Optimization of Fabrication Plans. In *2021 International Conference on Robotics and Automation (ICRA)*, 2021.

[101] Jun Li, Kai Xu, Siddhartha Chaudhuri, Ersin Yumer, Hao Zhang, and Leonidas Guibas. Grass: Generative recursive autoencoders for shape structures. *ACM Transactions on Graphics (TOG)*, 36(4):1–14, 2017.

[102] Minchen Li, Alla Sheffer, Eitan Grinspun, and Nicholas Vining. Foldsketch: Enriching Garments with Physically Reproducible Folds. *ACM Trans. Graph.*, 37(4), July 2018. ISSN 0730-0301.

[103] Jenny Lin, Vidya Narayanan, and James McCann. Efficient Transfer Planning for Flat Knitting. In *Proceedings of the 2Nd ACM Symposium on Computational Fabrication*, SCF '18, pages 1:1–1:7, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5854-5.

[104] Tsung-Yi Lin, Priyal Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

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

[106] LingShan Liu, Tao Zhang, Peng Wang, Xavier Legrand, and Damien Soulat. Influence of the tufting yarns on formability of tufted 3-Dimensional composite reinforcement. *Composites Part A: Applied Science and Manufacturing*, 78: 403–411, 2015.

[107] Gabriel Loke, Tural Khudiyev, Brian Wang, Stephanie Fu, Syamantak Payra, Yorai Shaoul, Johnny Fung, Ioannis Chatziveroglou, Pin-Wen Chou, Itamar Chinn, et al. Digital electronics in fibres enable fabric-based machine-learning inference. *Nature communications*, 12(1):1–9, 2021.

[108] Matt Lombard. *SolidWorks 2013 Bible*. John Wiley & Sons, 2013.

[109] Yiyue Luo, Yunzhu Li, Pratyusha Sharma, Wan Shou, Kui Wu, Michael Foshey, Beichen Li, Tomás Palacios, Antonio Torralba, and Wojciech Matusik. Learning human–environment interactions using conformal tactile textiles. *Nature Electronics*, 4(3):193–201, 2021.

[110] Yiyue Luo, Kui Wu, Tomás Palacios, and Wojciech Matusik. KnitUI: Fabricating Interactive and Sensing Textiles with Machine Knitting. In *Proceedings of*

the 2021 CHI Conference on Human Factors in Computing Systems, CHI '21, pages 1–12, New York, NY, USA, 2021. Association for Computing Machinery.

[111] Yishay Mansour, Mehryar Mohri, and Afshin Rostamizadeh. Domain adaptation: Learning bounds and algorithms. In *Conference on Learning Theory*, 2009.

[112] Shashank G Markande and Elisabetta A Matsumoto. Knotty knits are tangles on tori. *arXiv preprint arXiv:2002.01497*, 2020.

[113] MarvelousDesigner. MarvelousDesigner, 2020. URL https://www.marvelousdesigner.com. [Online; Accessed: 2021-08-16].

[114] William S Massey. *A Basic Course in Algebraic Topology*, volume 127. Springer, 2019.

[115] James McCann. The "Knitout" (.k) File Format, 2017. URL https://textiles-lab.github.io/knitout/knitout.html. [Online; Accessed: 2021-08-16].

[116] James McCann, Lea Albaugh, Vidya Narayanan, April Grow, Wojciech Matusik, Jennifer Mankoff, and Jessica Hodgins. A Compiler for 3D Machine Knitting. *ACM Transactions on Graphics (TOG)*, 35(4):49:1–49:11, July 2016. ISSN 0730-0301.

[117] Ministry of Supply. Ministry of Supply, 2021. URL https://www.ministryofsupply.com/. [Online; Accessed: 2021-08-16].

[118] Juan Montes, Bernhard Thomaszewski, Sudhir Mudur, and Tiberiu Popa. Computational Design of Skintight Clothing. *ACM Trans. Graph.*, 39(4), July 2020. ISSN 0730-0301.

[119] Carolyn L Moore, Kathy K Mullet, and Margaret Prevatt Young. *Concepts of Pattern Grading: Techniques for Manual and Computer Grading*. Fairchild Books, 2001.

[120] K. Murasugi and B. Kurpita. *A Study of Braids*. Mathematics and Its Applications. Springer Netherlands, 2012. ISBN 978-94-015-9319-9. URL https://books.google.com/books?id=VLTnCAAAQBAJ.

[121] Georges Nader, Yu Han Quek, Pei Zhi Chia, Oliver Weeger, and Sai-Kit Yeung. KnitKit: A flexible system for machine knitting of customizable textiles. *ACM Transactions on Graphics (TOG)*, 40(4), 2021.

[122] Vidya Narayanan, Lea Albaugh, Jessica Hodgins, Stelian Coros, and James Mccann. Automatic Machine Knitting of 3D Meshes. *ACM Transactions on Graphics*, 37(3), August 2018. ISSN 0730-0301.

[123] Vidya Narayanan, Kui Wu, Cem Yuksel, and James McCann. Visual knitting machine programming. *ACM Transactions on Graphics (TOG)*, 38(4):1–13, 2019.

[124] Addy Ngan, Frédo Durand, and Wojciech Matusik. Experimental analysis of BRDF models. *Rendering Techniques*, 2005(16th):2, 2005.

[125] Kirsi Niinimäki, Greg Peters, Helena Dahlbo, Patsy Perry, Timo Rissanen, and Alison Gwilt. The environmental price of fast fashion. *Nature Reviews Earth & Environment*, 1(4):189–200, 2020.

[126] Syamantak Payra, Irmandy Wicaksono, Juliana Cherston, Cedric Honnet, Valentina Sumini, and Joseph A Paradiso. Feeling through spacesuits: Application of space-resilient e-textiles to enable haptic feedback on pressurized extravehicular suits. In *2021 IEEE Aerospace Conference (50100)*, pages 1–12, 2021.

[127] Huaishu Peng, Jennifer Mankoff, Scott E. Hudson, and James McCann. A layered fabric 3D printer for soft interactive objects. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 1789–1798. Association for Computing Machinery, New York, NY, USA, 2015. ISBN 978-1-4503-3145-6. URL https://doi.org/10.1145/2702123.2702327.

[128] Huaishu Peng, Scott Hudson, Jennifer Mankoff, and James McCann. Soft printing with fabric. *XRDS: Crossroads, The ACM Magazine for Students*, 22(3): 50–53, 2016.

[129] Joel Peterson, Jonas Larsson, Jan Carlsson, and Peter Andersson. Knit on demand - development and simulation of a production and shop model for customised knitted garments. *International Journal of Fashion Design, Technology and Education*, 1(2):89–99, 2008. doi: 10.1080/17543260802353399. URL https://doi.org/10.1080/17543260802353399.

[130] Salvinija Petrulyte and Renata Baltakyte. Static water absorption in fabrics of different pile height. *Fibres & Textiles in Eastern Europe*, 17(3):60–65, 2009.

[131] John Picton and John Mack. *African Textiles*. Trustees of the British Museum, 1989.

[132] Ivan Poupyrev, Nan-Wei Gong, Shiho Fukuhara, Mustafa Emre Karagozler, Carsten Schwesig, and Karen E. Robinson. Project Jacquard: Interactive Digital Textiles at Scale. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 4216–4227, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 978-1-4503-3362-7.

[133] Andrea Berman Price. *Knitspeak: An A to Z Guide to the Language of Knitting Patterns*. Open Road Media, 2011.

[134] J Dale Prince. 3D printing: An industrial revolution. *Journal of electronic resources in medical libraries*, 11(1):39–45, 2014.

[135] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants*. Springer Science & Business Media, 2012.

[136] Michael Rein, Valentine Dominique Favrod, Chong Hou, Tural Khudiyev, Alexander Stolyarov, Jason Cox, Chia-Chun Chung, Chhea Chhav, Marty Ellis, John Joannopoulos, et al. Diode fibres for fabric-based optical communications. *Nature*, 560(7717):214, 2018.

[137] Juergen Riegel, Werner Mayer, and Yorik van Havre. FreeCAD. FreeCAD, 2016. URL https://www.freecadweb.org/. [Online; Accessed: 08-16-2021].

[138] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2015.

[139] Harold A Rothbart. *Cam Design Handbook*. McGraw-Hill Education, 2004.

[140] Carsten Rother, Vladimir Kolmogorov, and Andrew Blake. Grabcut: Interactive foreground extraction using iterated graph cuts. *ACM Transactions on Graphics*, 23(3):309–314, 2004.

[141] Grzegorz Rozenberg and Arto Salomaa. *The Mathematical Theory of L Systems*. Academic press, 1980.

[142] Gerard Rubio. OpenKnit: Open Source Digital Knitting, 2014. URL http://openknit.org. [Online; Accessed: 2018-09-01].

[143] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115(3):211–252, 2015.

[144] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans. In *Advances in Neural Information Processing Systems*, 2016.

[145] Gustav Sandin and Greg M. Peters. Environmental impact of textile reuse and recycling – A review. *Journal of Cleaner Production*, 184:353–365, 2018. ISSN 0959-6526. doi: 10.1016/j.jclepro.2018.02.266. URL https://www.sciencedirect.com/science/article/pii/S0959652618305985.

[146] Triambak Saxena, Gerard Rubio, and Tom Catling. Kniterate: The Digital Knitting Machine, 2017. URL https://www.kickstarter.com/projects/kniterate/kniterate-the-digital-knitting-machine. [Online; Accessed: 2018-09-01].

[147] Abu Sadat Muhammad Sayem, Richard Kennon, and Nick Clarke. 3D CAD systems for the clothing industry. *International Journal of Fashion Design, Technology and Education*, 3(2):45–53, 2010.

[148] Nancy A Schofield. *Pattern Grading*. Cambridge, Woodhead Publishing Limited, 2007.

[149] Peter Schröder. Subdivision as a fundamental building block of digital geometry processing algorithms. *Journal of Computational and Applied Mathematics*, 149(1):207–219, 2002. ISSN 0377-0427. doi: 10.1016/S0377-0427(02)00530-7. URL https://www.sciencedirect.com/science/article/pii/S0377042702005307. Scientific and Engineering Computations for the 21st Century - Me thodologies and Applications Proceedings of the 15th Toyota Conference.

[150] Shima Seiki. SDS-ONE Apex3, 2011. URL http://www.shimaseiki.com/product/design/sdsone_apex/flat/. [Online; Accessed: 2018-09-01].

[151] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge university press, 2014.

[152] Nicholas Sharp, Keenan Crane, et al. Geometry-central, 2019. URL https://www.geometry-central.net. [Online; Accessed: 08-16-2021].

[153] Yu Shen, Junbang Liang, and Ming C Lin. GAN-based Garment Generation Using Sewing Pattern Images. In *Proceedings of the European Conference on Computer Vision (ECCV)*, volume 1, page 3, 2020.

[154] Hitomi Shida and Gayle Roehm. *Japanese Knitting Stitch Bible: 260 Exquisite Patterns by Hitomi Shida*. Tuttle Publishing, 2017.

[155] Ashish Shrivastava, Tomas Pfister, Oncel Tuzel, Joshua Susskind, Wenda Wang, and Russell Webb. Learning from simulated and unsupervised images through adversarial training. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[156] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[157] Richard Socher, Cliff Chiung-Yu Lin, Andrew Y Ng, and Christopher D Manning. Parsing natural scenes and natural language with recursive neural networks. In *ICML*, 2011.

[158] D.J. Spencer. *Knitting Technology: A Comprehensive Handbook and Practical Guide*. Woodhead Publishing Series in Textiles. Technomic publishing, 2001. ISBN 978-1-58716-121-6. URL https://books.google.com/books?id=zsoRvDWPd2gC.

[159] Stoll. M1Plus pattern software, 2011. URL http://www.stoll.com/stoll_software_solutions_en_4/pattern_software_m1plus/3_1. [Online; Accessed: 2018-09-01].

[160] Eliza Strickland. Shapeways bringing 3-D printing to the masses. *Ieee Spectrum*, 50(11):22–22, 2013.

[161] Bjarne Stroustrup. *The C++ Programming Language*. Pearson Education, 2013.

[162] Vitaly Surazhsky, Tatiana Surazhsky, Danil Kirsanov, Steven J. Gortler, and Hugues Hoppe. Fast Exact and Approximate Geodesics on Meshes. *ACM Trans. Graph.*, 24(3):553–560, July 2005. ISSN 0730-0301.

[163] Guangming Tao, Heike Ebendorff-Heidepriem, Alexander M Stolyarov, Sylvain Danto, John V Badding, Yoel Fink, John Ballato, and Ayman F Abouraddy. Infrared fibers. *Advances in Optics and Photonics*, 7(2):379–458, 2015.

[164] A. Thompson. *Narrow Fabric Weaving*. Read Books Limited, 2013. ISBN 978-1-4733-8996-0. URL https://books.google.com/books?id=gi9-CgAAQBAJ.

[165] Bruce Trace. On the Reidemeister moves of a classical knot. *Proceedings of the American Mathematical Society*, pages 722–724, 1983.

[166] E. Turquin, J. Wither, L. Boissieux, M. Cani, and J. F. Hughes. A Sketch-Based Interface for Clothing Virtual Characters. *IEEE Computer Graphics and Applications*, 27(1):72–81, 2007.

[167] Eric Tzeng, Judy Hoffman, Kate Saenko, and Trevor Darrell. Adversarial Discriminative Domain Adaptation. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[168] Dmitry Ulyanov, Andrea Vedaldi, and Victor Lempitsky. Instance Normalization: The Missing Ingredient for Fast Stylization. *arXiv:1607.08022*, 2016.

[169] Nobuyuki Umetani, Danny M. Kaufman, Takeo Igarashi, and Eitan Grinspun. Sensitive Couture for Interactive Garment Modeling and Editing. *ACM Trans. Graph.*, 30(4), July 2011. ISSN 0730-0301.

[170] Jenny Underwood. *The Design of 3D Shape Knitted Preforms*. PhD Thesis, Fashion and Textiles, RMIT University, 2009.

[171] Lieva Van Langenhove. *Smart Textiles for Medicine and Healthcare: Materials, Systems and Applications*. Elsevier, 2007.

[172] Vasturiano. Force-directed graph rendered on HTML5 canvas, 2018. URL https://github.com/vasturiano/force-graph. [Online; Accessed: 2018-08-22].

[173] Diederik Veenendaal, Mark West, and Philippe Block. History and overview of fabric formwork: Using fabrics for concrete casting. *Structural Concrete*, 12(3): 164–177, 2011.

[174] Petras Vestartas, Mary Katherine Heinrich, Mateusz Zwierzycki, David Andres Leon, Ashkan Cheheltan, Riccardo La Magna, and Phil Ayres. Design tools and workflows for braided structures. In Klaas De Rycke, Christoph Gengnagel, Olivier Baverel, Jane Burry, Caitlin Mueller, Minh Man Nguyen, Philippe Rahm, and Mette Ramsgaard Thomsen, editors, *Humanizing Digital Reality: Design Modelling Symposium Paris 2017*, pages 671–681. Springer Singapore, Singapore, 2018. ISBN 978-981-10-6611-5. URL https://doi.org/10.1007/978-981-10-6611-5_55.

[175] W. D. F. Vincent. Part 1: Juvenile and Youth's Garments. In *The Cutters' Practical Guide*. John Williamson Co. Limited, 1898.

[176] Pascal Volino, Frederic Cordier, and Nadia Magnenat-Thalmann. From early virtual garment simulation to interactive fashion design. *Computer-Aided Design*, 37(6):593–608, 2005. ISSN 0010-4485. doi: 10.1016/j.cad.2004.09.003. URL https://www.sciencedirect.com/science/article/pii/S0010448504002209. CAD Methods in Garment Design.

[177] Frederick T Wallenberger and Paul A Bingham. Fiberglass and glass technology. *Energy-Friendly Compositions And Applications*, 2010.

[178] Charlie C. L. Wang, Yu Wang, and Matthew M. F. Yuen. Design Automation for Customized Apparel Products. *Comput. Aided Des.*, 37(7):675–691, June 2005. ISSN 0010-4485.

[179] Huamin Wang. Rule-Free Sewing Pattern Adjustment with Precision and Efficiency. *ACM Trans. Graph.*, 37(4), July 2018. ISSN 0730-0301.

[180] Sen Wang, Ang Lu, and Lina Zhang. Recent advances in regenerated cellulose materials. *Progress in Polymer Science*, 53:169–206, 2016.

[181] Tuanfeng Y. Wang, Duygu Ceylan, Jovan Popović, and Niloy J. Mitra. Learning a Shared Shape Space for Multimodal Garment Design. *ACM Trans. Graph.*, 37(6), December 2018. ISSN 0730-0301.

[182] James CY Watt, Anne E Wardwell, and Morris Rossabi. *When Silk Was Gold: Central Asian and Chinese Textiles*. Metropolitan Museum of art, 1997.

[183] Irmandy Wicaksono, Carson I Tucker, Tao Sun, Cesar A Guerrero, Clare Liu, Wesley M Woo, Eric J Pence, and Canan Dagdeviren. A tailored, electronic textile conformable suit for large-scale spatiotemporal physiological sensing in vivo. *Nature Flexible Electronics*, 4(1):1–13, 2020.

[184] U Wollina, M Heide, W Müller-Litz, D Obenauf, and J Ash. Functional textiles in prevention of chronic wounds, wound healing and tissue engineering. *Curr Probl Dermatol*, 31:82–97, 2003.

[185] K. Wu and C. Yuksel. Real-time cloth rendering with fiber-level detail. *IEEE Transactions on Visualization and Computer Graphics*, PP(99):1–1, 2017. ISSN 1077-2626. doi: 10.1109/TVCG.2017.2731949.

[186] Kui Wu, Xifeng Gao, Zachary Ferguson, Daniele Panozzo, and Cem Yuksel. Stitch Meshing. *ACM Transactions on Graphics (SIGGRAPH)*, 37(4):130:1–130:14, July 2018. ISSN 0730-0301.

[187] Kui Wu, Hannah Swan, and Cem Yuksel. Knittable Stitch Meshes. *ACM Transactions on Graphics*, 38(1):10:1–10:13, January 2019. ISSN 0730-0301.

[188] Kui Wu, Marco Tarini, Cem Yuksel, James Mccann, and Xifeng Gao. Wearable 3D Machine Knitting: Automatic Generation of Shaped Knit Sheets to Cover Real-World Objects. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–1, 2021.

[189] Fisher Yu and Vladlen Koltun. Multi-Scale Context Aggregation by Dilated Convolutions. In *International Conference on Learning Representations*, 2016.

[190] Cem Yuksel, Jonathan M. Kaldor, Doug L. James, and Steve Marschner. Stitch Meshes for Modeling Knitted Clothing with Yarn-level Detail. *ACM Transactions on Graphics (SIGGRAPH)*, 31(3):37:1–37:12, 2012.

[191] Shuang Zhao, Fujun Luan, and Kavita Bala. Fitting procedural yarn models for realistic cloth rendering. *ACM Transactions on Graphics (TOG)*, 35(4):51, 2016.

[192] Shuai Zheng, Sadeep Jayasumana, Bernardino Romera-Paredes, Vibhav Vineet, Zhizhong Su, Dalong Du, Chang Huang, and Philip HS Torr. Conditional random fields as recurrent neural networks. In *IEEE International Conference on Computer Vision*, 2015.

[193] Bolei Zhou, Hang Zhao, Xavier Puig, Tete Xiao, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Semantic understanding of scenes through the ADE20K dataset. *International Journal of Computer Vision*, 2018.

[194] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks. In *IEEE International Conference on Computer Vision*, 2017.