Knitting with Directed Graphs

by

Jared B. Counts

B.S., Massachusetts Institute of Technology (2017)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

Author . . . .
Department of Electrical Engineering and Computer Science
May 25, 2018

Certified by . . . .
Hiroshi Ishii
Jerome B. Wiesner Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Knitting with Directed Graphs

by

Jared B. Counts

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Knitting has historically been communicated by its means of construction. For hand knitting, this is typically a list of instructions or a pictorial grid with knitting symbols. For machine knitting, a similar pictorial grid is used to express needle-level instructions. However, these formats suffer by the nature of their tight coupling with the method used to construct the garments they represent. Alternatively, we use Knit Meshes, which represent knitting structures by their geometry separate from a directed graph description of their topology. This thesis presents an algorithm that can generate a natural, deformed two-dimensional layout of Knit Meshes as well as a conversion pipeline that converts written hand knitting instructions to and from Knit Meshes and an algorithm that converts certain Knit Meshes into knitting machine code.

Thesis Supervisor: Hiroshi Ishii
Title: Jerome B. Wiesner Professor of Media Arts and Sciences

Acknowledgments

I would like to show gratitude to girlfriend Catalina Nguyen for all of her love and support over these last 3 years. Her positivity and affection has kept me sane through difficult times.

Thanks to Professor Hiroshi Ishii for his countless opportunities given to me during my time here at MIT. It has been an amazing privilege to work with someone with his energy and insight.

I had a tremendous time working with my colleagues in Tangible Media Group. The group administrator Deema Qashat has been a crucial part in the success of my thesis. My mentor Jifei Ou has been incredibly helpful in his counsel and creative ideas, and without his help I would have had little direction.

This research was made possible thanks to Shima Seiki and Matsuya for providing access to their knitting machines.

Contents

1 Introduction **13**

2 Background **15**

2.1 Overview 15

3 Knit Mesh **17**

3.1 Topology 18

3.1.1 Related 20

3.2 Geometry 20

3.3 Procedures 21

4 Shape Prediction **23**

4.1 Related 24

4.2 Distance Metric 24

4.3 Embedding 25

4.3.1 Algorithms 25

4.4 Results 28

4.5 Discussion 30

5 Hand Knitting **31**

5.1 Virtual Hand Knitter 33

5.2 Instructions to Knit Mesh 34

5.2.1 Hand knitting procedures 34

5.2.2 Interpreter 36

5.2.3 Results 39

5.3 Knit Mesh to Instructions 39

5.3.1 Node Processing 41

5.3.2 Pin to stack 41

5.3.3 Node dropping 44

5.3.4 Decreases and PSSOs 44

5.3.5 Other additions 45

5.3.6 Results 45

5.4 Discussion 49

6 Machine Knitting 51

6.1 Generating Knitting Machine Code 53

6.1.1 Related 54

6.1.2 Limitations 55

6.1.3 Virtual Knitting Machine 55

6.1.4 Machine Knitting Graph 56

6.1.5 Knitting Code 60

6.1.6 Results 62

6.2 Discussion 62

7 Conclusion 65

List of Figures

2-1 On the left, a knitted structure with a single stitch shaded. On the right, a woven structure, not to be confused with knitting. 15

3-1 An example of a knit graph overlaying a stockinette pattern. The black circles are nodes, the horizontal orange arrows are yarn edges, and the vertical blue arrows are loop edges. 19

4-1 Hand knitted 2x2 Waves Edging pattern from "Up, Down, All-Around Stitch Dictionary" [4]. 23

4-2 Embeddings of different knitted patterns. From top to bottom: stockinette, Tilted Blocks, Waves Edging, and Horseshoe Lace. Each of these patterns were taken from the book, "Up, down, all-around stitch dictionary" [4]. 28

5-1 Illustration of stitch creation in hand knitting. On the left, a knitting pin is inserted through a loop, after which another loop is formed around the end of this knitting pin. On the right, the loop formed on the knitting pin is pulled through the other loop, forming a complete stitch. 31

6-1 Screenshot of machine knitting code from Matsuya's SPJWin [2]. . . 54

List of Tables

5.1 Mapping of common abbreviations to procedures. 36

5.2 Interpreter results showing the Python [24] generated hand knitting instructions compared with the automatically generated knit graphs, embedded. Knitting patterns are from "Up, down, all-around stitch dictionary" [4]. 40

5.3 Handling incoming loops to a given node that is being processed (circled in green). 42

5.4 Generated instructions for various knit graphs. 47

5.5 Instructions were automatically generated from each of these knit meshes, which were then followed to hand knit these samples. 49

6.1 Terms used to describe hand knitting and their analogous counterparts for machine knitting. 52

6.2 Examples of machine knitting graphs. Note that, although transfers happen on two separate beds, the displayed graphs doesn't make a differentiation between which bed each node is transferring to or from. It may be assumed that every transfer alternates the beds that the node is occupying. 59

6.3 Knitting machine graphs and their corresponding knitting machine pseudocode. Cells indicating SWAP correspond to a transfer from one bed to the opposite at the same wale position. Cells indicating XFER refer to a transfer operation to the opposite bed with some offset. Each grid cell is labeled with the operation name and a number to indicate their ordering. 62
6.4 Results from computer generating machine code from a given knit graph. The machine knitted samples were knitted on the Matsuya M-100 knitting machine. The patterns for the sample were borrowed from the book, "Up, down, all-around stitch dictionary" [4]. 63

Chapter 1

Introduction

The concept of knitting has been around for a millenium. Despite the long existence of knitting, computational tools have been underutilized within the textile industry. Aside from computation, the way we describe knitting structures has greatly limited the kinds of structures we knit.

Current computational tools are designed for the purpose of programming knitting machines, where the very structure of knitting is treated as an afterthought. Consequently, the design of garments is tightly coupled with various knitting machine procedures, making the process of designing knitted structures cumbersome and awkward. Furthermore, the tools used for knitting machines tend to be proprietary, where each knitting machine company uses their own formats and computational tools.

More broadly, knitting has always been viewed through the lens of a two-dimensional grid. Yet, when we look at knitting structures topologically, there doesn't seem to be anything necessarily two-dimensional about it.

My thesis explores a description of knitted structures that separates topology from geometry, called Knit Meshes. Topologically, I describe knitted structures using directed graphs. Through topology, the knitted structures we describe no longer have to be 2D grid embeddable, nor do we need to concern ourselves with how it is physically knitted in practice. The geometric part further describes components of knitted patterns such as loop orientation and position.

I examine two groups of algorithms for Knit Meshes. First, it is shown that the

Knit Mesh topology is the only thing necessary to produce a 2D embedding that can predict the shape of its physically knitted product. Second, two conversion pipelines: one for converting Knit Meshes back and forth from hand knitting instructions and another for converting Knit Meshes into machine knitting instructions. This allows us to manufacture arbitrary Knit Meshes as well as translate hand knitting instructions into machine knitting code.

Chapter 2

Background

2.1 Overview

Knitting broadly refers to the construction of soft structures by intermeshing loops formed from one or more yarns. This is not to be confused with weaving, which refers to a lattice of overlapping, mostly straight pieces of yarn. Figure 2-1 illustrates the differences between knitting and weaving.

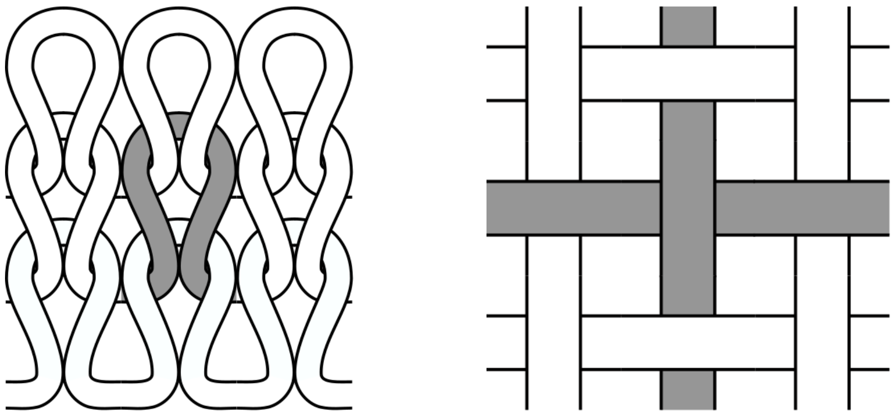


Figure 2-1: On the left, a knitted structure with a single stitch shaded. On the right, a woven structure, not to be confused with knitting.

A *stitch* is a basic unit of knitting. Each stitch is a loop that is intermeshed with one or more other loops.

Stitches in knitted structures are typically arranged along *courses* and *wales*. Each course corresponds to a row of stitches, and each wale corresponds to a column of intermeshed loops.

Knitted structures may be constructed in the *warp* or *weft* direction. The warp runs parallel to each wale, where-as the weft direction is parallel to each course. The methods that I will discuss in this thesis, both hand knitting and machine knitting, primarily relate to knitting in the weft direction.

There are many different ways to produce knitted garments. Two of the ways that I discuss include hand knitting and flat bed weft machine knitting. These will be explored in more depth in Chapters 5 and 6. Furthermore, related work will be further discussed in each chapter as they become relevant.

Chapter 3

Knit Mesh

Hand knitters typically communicate knitting in one of two ways. The first is as a list of abbreviated instructions, organized by row. The second is pictorially in a grid, where each cell contains some symbol corresponding to a certain knitting instruction, known as a *stitch chart*.

Programmers for industrial knitting machines usually work with a pictorial programming language similar to the stitch charts that hand knitters use, such as Shima Seiki's KnitPaint [13]. They're a good way for programming a knitting machine, but it takes experts to design new structures or shapes with these languages. Machine knitting will be explored in more depth in Chapter 6.

Consider, by analogy, 3D printers. A native format for 3D printers might be some numerical control programming language, dictating the paths that the extruder or laser needs to follow in order to manufacture the 3D object one layer at a time. However, when a designer wants to create or modify a virtual object to be 3D printed, they don't typically work with this path-based representation. One usually designs the 3D object using an intermediate format, which represents the shape of the virtual object in a way that makes it easier to computationally deal with. Only after the design is finished, the user run slicing and path-planning algorithms to generate the necessary format for 3D printing.

In theory, one can virtually represent 3D objects by these paths, but making changes, running algorithms, or converting to other formats is less than trivial when dealing with paths. The path information is a convoluted way of representing 3D objects for most purposes, so a more generalized form is used instead, i.e. 3D polygonal meshes.

3D polygonal meshes are separated into two parts. First is the mesh geometry, usually stored as a list of points in space. The second part is the mesh topology, whichs gives the connectedness of these points. The topology is given by some set of point index pairs or tuples, indicating edges and triangles respectively. There are many benefits to keeping geometry and topology separate. One may perform transformations on the mesh simply by applying functions to the geometry, without worrying about the topology. Likewise for topology, one can analyze or make changes to certain properties of the mesh's connectedness without affecting geometry.

Knitting machine code is analogous to the path-planning code used to 3D print. Unlike in 3D printing, however, there doesn't seem to be a well understood format for representing knitting structures in the same way that 3D polygonal meshes directly represent 3D objects.

Consider a long strand of yarn. To knit, one forms two loops in the yarn, and then pulls one through the other. The process of forming and intermeshing loops is repeated until some desired shape or structure is reached. From this, we can think of two ways of describing this structure. First, in what order along the yarn do we form loops? Second, what loops intermesh with what other loops? Central to my thesis is the concept of knit graphs, which provide a topological description of knitting based on the relationship between loops and the yarn they lie on. We will also briefly explore what makes up the geometry of knitted structures.

The topology of a knitted structure combined with its geometry constitutes what may be known as a *Knit Mesh.*

3.1 Topology

Let $T = \{V, Y, L\}$ be a *knit graph*, which topologically describes some knitted structure. $V = \{v\}$ is a set of nodes, each representing some point along a yarn. The yarn edges, $Y = \{(u, v) \mid u, v \in V\}$, is a set of directed edges between pairs of nodes. Yarn edge $y \in Y$ describes the adjacency and ordering of two nodes along a yarn. The loop edges, $L = \{(u, v) \mid u, v \in V\}$, is another set of directed edges. Loop edge $l \in L$ describes a loop of yarn by the point along the yarn it is formed, i.e., l's starting point u, and another point along the yarn from which this loop is intermeshed around, i.e., l's ending point v. A diagram of a knit graph overlaying a stockinette pattern can be seen in Figure 3-1.

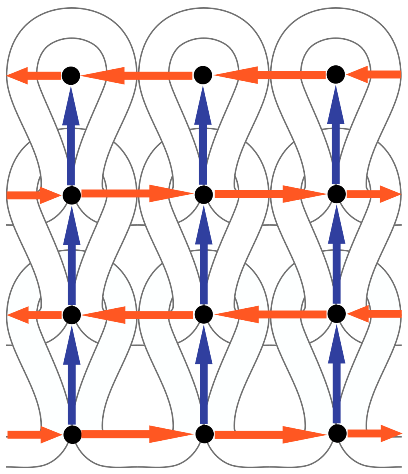


Figure 3-1: An example of a knit graph overlaying a stockinette pattern. The black circles are nodes, the horizontal orange arrows are yarn edges, and the vertical blue arrows are loop edges.

One can immediately consider 3 rules for what can be considered a valid knit graph.

1. **No cycles** may be formed in Y, as the yarn is assumed to be cut at some point.

2. Every connected set of edges in Y must form a **directed path**. While there may be multiple, separate yarn, none of them may branch and form tree structures. And, since Y determines an ordering of nodes, all edges must be oriented in one direction.

3. **No self edges** in Y or L. A loop formed around itself would either be two loops, or would simply unravel. In yarn, we expect nodes to be ordered relative to each other, not to themselves.

It's also worth noting a certain limitation of knit graphs. In knit graphs, a loop may overlap multiple loops only if those loops start from the same node. Certain knitting patterns contain stitches that involving looping yarn around multiple groups of loops. For example, the Pelerine eyelet [23].

3.1.1 Related

Stitch Meshes by Cem Yuksel et al. [26] is one interesting topological interpretation of knitting structures. Their idea is to use a 3D polygonal mesh, where the edge adjacency of polygons indicate how stitches are connected. This is an excellent way of generating knitting patterns for Kaldor et al.'s yarn level simulator [11], however it is restrictive as a knitting structure are not necessarily 3D surface embeddable.

Knit graphs are not novel. In 2011, Jonathan Kaldor [10] describes knit geometry as a set of loops that are related to each other by predecessor and successor. In 2017, the papers by Jiang et al. [8] and Popescu et al. [19] describe network-like ways of representing knit structure. The paper by Narayanan et al. [18] describes a directed graph construction very similar to the one used in this thesis.

3.2 Geometry

So far, with knit graphs, we've only described the topology of knitting structures. Earlier we drew an analogy with 3D polygonal meshes, which separates topology from geometry. Certain geometric aspects of knitting structures are left undescribed by knit graphs alone.

Geometric properties may include

1. **Node positions** describing the 2 or 3 dimensional layout of nodes in a knitted structure. This is ambiguous, as knitted structures are pliable and may take many different relaxed forms. For example, one may describe node positions along a grid using their course and wale numbers, or by some 2 or 3D spatial position determined by some physically based relaxation.

2. **Loop face** describing whether a stitch loop is technical front or technical back. Technical front means that the loop is inserted behind some other loop, whereas technical back means the loop is inserted in front of another loop. Each of these creates a different appearance in knitting.

3. **Loop plane** determining how loops stack when they overlap. This would be especially useful in cable patterns, where some number of vertical loops overlap another group of vertical loops to create a twisting cable appearance.

4. **Loop twistedness**, the number of times a loop is twisted left or right.

The geometric properties here are somewhat ambiguously defined. For instance, the loop face, plane, and twistedness are dependent on what side of the textile we define as the front versus the back. Furthermore, some constraints may be needed between some of these definitions to prevent contradictions. If one were to knit a mobius strip, but define every loop as technical face, that would be contradictory as a mobius strip has only one face. It remains to be done to design a more formal description of knit geometry.

In my thesis, I focus less on the geometry and more on the use of knit graphs, but in some cases I also consider geometric properties like the loop face and node position. Otherwise, it's assumed that there is no twisting and that loops are ordered arbitrarily.

3.3 Procedures

In later chapters of this thesis, I will make use of certain procedures for manipulating Knit Meshes. These include

1. New-Node(T, y) where T is the directed graph description of the knit topology, and y is some choice of yarn. This adds a node to the knit graph and then returns said node.

2. New-Loop(T, u, v) where T is again the knit graph, and u and v are distinct nodes. This creates a loop l between nodes u and v, and then returns the said loop.

3. Set-Face(G, l, F) where G is the knit geometry, l is some loop, and F is the desired face. F may be KNIT or PURL. This sets the loop face to match the instruction a hand knitter must follow to construct the given loop.

Chapter 4

Shape Prediction

Given just a knitted topology without any explicit geometric information, the problem of determining a suitable layout becomes nontrivial. Typically knitting information is grid aligned, however it isn't clear, from a knit graph alone, as to what nodes belong to which courses and rows. Furthermore, it's desirable to capture some of the deformation that patterns have. For example, the 2x2 Waves Edging pattern [4] naturally rests in a wave-like shape, as shown in Figure 4-1.

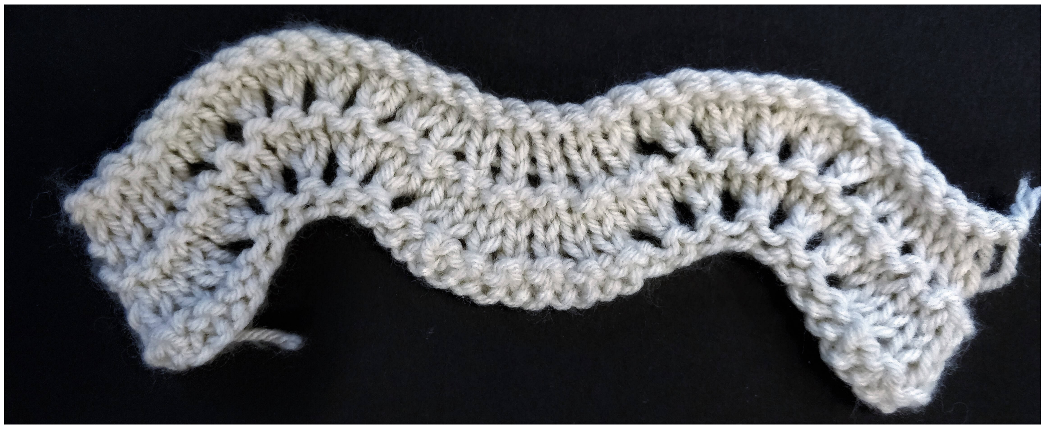


Figure 4-1: Hand knitted 2x2 Waves Edging pattern from "Up, Down, All-Around Stitch Dictionary" [4].

The goal of this project is to find an embedding algorithm that can approximate the physical layout of knit graphs.

4.1 Related

In his thesis [10], Jonathan Kaldor demonstrates methods for simulating cloth at a yarn level. To initialize the simulation, he simply inserts pre-modeled stitches along a grid. However, Kaldor does briefly discuss an attempt to create a better initial 3D layout using locally linear embedding and spectral embedding. He found that both methods required human intervention to produce ideal layouts.

There exists many other embedding algorithms for graphs. I will describe some of them in more detail in Section 4.3, as well as their suitability for knitting.

4.2 Distance Metric

One of the difficulties with embedding a graph is that there is no clear ideal distance metric to use. Since knitted textiles are soft and pliable, it's somewhat arbitrary as to how we decide to geometrically plot it. If we decide to plot it based on some physical interpretation, we may need to consider some of the complicated physical forces are at play, as yarn may push, pull, curl, and twist with itself just by the nature of how loops are intermeshed.

In the late 1950's, Munden [17] studied the shape of a knitted loop. He found that, under energy minimization assumptions, the dimensional properties of textile is solely determined by length of the yarn in individual stitches.

In his derivations, he determined that

$$\frac{\text{courses}}{\text{inch}} = \frac{a}{l}$$

and

$$\frac{\text{wales}}{\text{inch}} = \frac{b}{l}$$

where a and b are constants and l is the length of the stitch. It must be true then that

$$\text{length of 1 course} = O(l)$$

and

$$\text{length of 1 wale} = O(l),$$

i.e., the distance between two wales and the distance between two courses of stitches are linear with respect to the length of yarn in a stitch. The exact constant that the stitch requires must be dependent on the yarn material, so we arbitrarily choose that

$$\text{yarn edge length} = \text{loop edge length} = 1\,. \tag{4.1}$$

The problem now is that the distance between adjacent vertices is not enough to do a full embedding. The knit graph is typically sparse with respect to edge count, so we should expect that most embeddings would collapse in on itself.

When examining a fabric on a flat surface, one may try to stretch it out as much as possible, perhaps by pinning the edges. This means that nodes on opposite ends of the graph may be pushed apart as much as the yarn between them allows. Thus, we choose the graph theoretic distances as our distance metric, i.e. we use the all-pair-shortest-path distances, where each edge is of length 1 as derived in Equation 4.1. This is somewhat arbitrary, but we'll see in Section 4.4 that this gives good results for certain embedding algorithms.

4.3 Embedding

4.3.1 Algorithms

I experimented with 5 different algorithms. I will briefly describe each here.

1. **Heuristic**.

This algorithm grid-aligns nodes by course and wale. It works by processing nodes in order of their yarn edges. As each node gets processed, there is a running course number and wale number that gets assigned. For each node, I increment or decrement the wale, depending on the parity of the course number. The course number gets incremented whenever there is a loop edge from the current node into the current course. In a stockinette, this signals that there is a new course. This is generally how knitting patterns are laid out. Physical deformation is not modeled using this algorithm.

2. **Laplacian Eigenmaps** (Spectral) [15].

   This applies spectral decomposition to the graph Laplacian of the given affinity matrix. Given that it requires an affinity matrix instead of a distance matrix, we compute the affinity A by taking

$$A = 1 - \frac{D}{\max_d(D)}$$

   where D is the distance matrix, and the denominator is the maximum value in the matrix D.

3. **Locally-linear Embedding** (LLE) [20].

   Given some point, LLE finds a neighborhood of nearby points and then computes a weighted sum of these points needed to reconstruct the given point. The set of weighted sums is then used to generated lower dimensional positions, i.e., the embedding. The sum of squared distance differences is the objective function for each patch of neighboring points.

4. **Metric Multidimensional Scaling** (mMDS) [14].

   This uses a notion of stress, which is the square root sum of squared distances in an embedding. This is minimized by using principle component analysis on the distance matrix.

5. **Kamada-Kawai using Barzilai Borwein** (KKBB) [12] [7].

Kamada and Kawai's algorithm treats pairwise distances as a kind of magnetic-spring hybrid. Its objective function is the sum of spring potential energy between every node pair, but the spring constants are chosen to be inversely proportional to the ideal distances squared. In the original paper [12], Kamada and Kawai minimize this by finding the node with the largest magnitude of energy gradient, and then iteratively moving the node using Newton-Raphson until its energy gradient is minimized. This process is repeated until a minimum is reached.

The problem is that the full energy gradient and Hessian needs to be recomputed every time we move a node. This takes a prohibitively long time for any reasonably sized graph. Thus, I used a Barzilai Borwein minimization scheme, as presented by Hasal, et al. [7]. The benefit with their approach is two-fold.

First, Barzilai Borwein only requires the gradient. In contrast, the Newton-Raphson method requires both the gradient, which is a vector, and the Hessian of the energy, a dense matrix. Using only the gradient is more efficient in both space and time. To replace the Hessian, the Barzilai Borwein minimization scheme estimates its inverse size using least squares, which is then used as a step size with respect to the energy gradient.

Second, Hasal, et al. moves the entire vector of locations along their gradient for each iteration. We no longer need to recompute the gradient for moving every point, but rather only for each move of the entire set of points.

Python programming language [24] was used for implementation. I used SciPy's [9] manifold embedding functions for Laplacian Eigenmaps, Locally-linear Embedding, and Metric Multidimensional Scaling. I implemented the grid-aligned algorithm and the Kamada-Kawai with Barzilai Borwein using NumPy [25]. For analyzing results. I will assess them qualitatively. The graph visualizations were made using Pygame [22].

4.4 Results

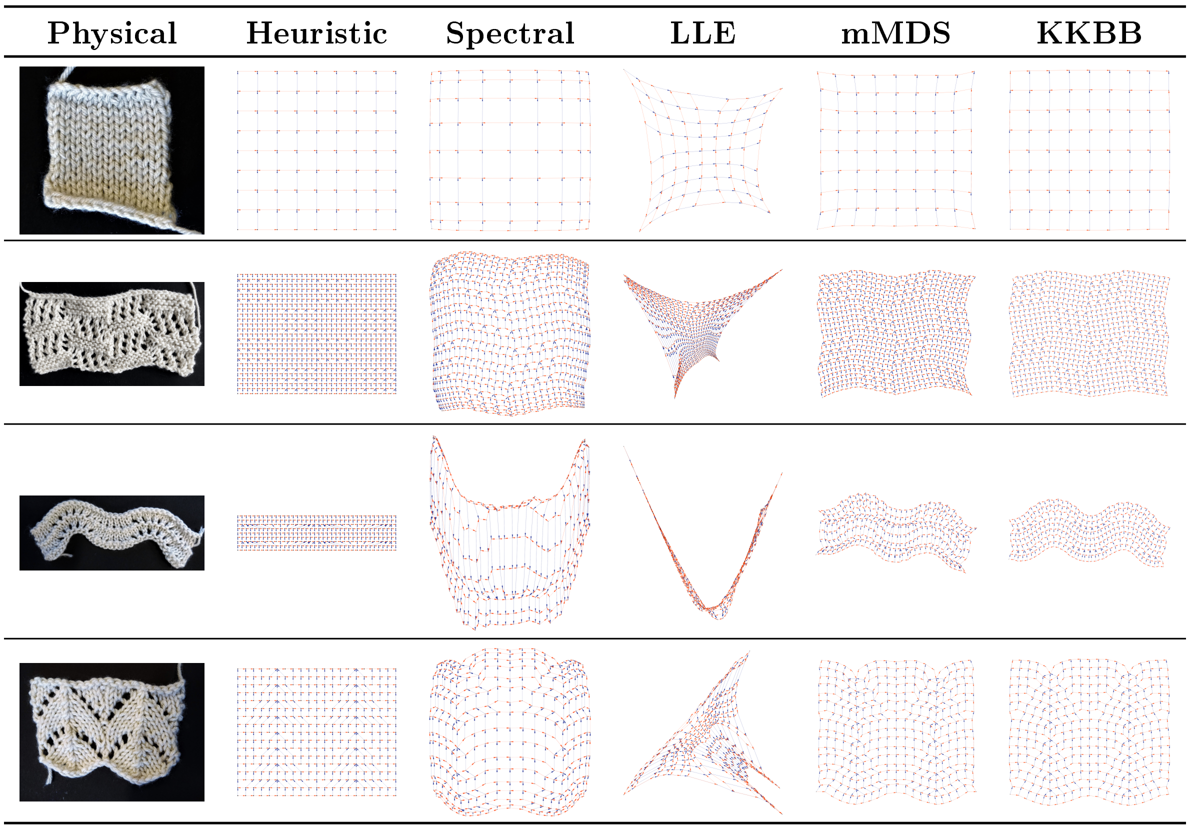


Figure 4-2: Embeddings of different knitted patterns. From top to bottom: stockinette, Tilted Blocks, Waves Edging, and Horseshoe Lace. Each of these patterns were taken from the book, "Up, down, all-around stitch dictionary" [4].

See Figure 4.4 for visual results. I will list out qualitative observations for each embedding algorithm.

1. **Heuristic**: As noted in Section 4.3.1, this embedding algorithm does nothing special for distortions caused by the knitting pattern. While this may be helpful for following along when hand-knitting, it's a poor predictor of the knitted textile's shape.

2. **Spectral**: Spectral embedding seems to suffer by ballooning effects. Nodes towards the edge of the graph tend to be closer together than nodes in the center. This may be due to the use of the graph Laplacian, which takes into

account the degree of graph nodes as well as their adjacency. As such, the degree of nodes at the edge of the graphs tend to be lower than those at the middle. It was also observed by Kaldor that it's often needed to manually choose the principal axes [10].

3. **LLE**: LLE performed the worse out of all the embedding methods used. It left strange artefacts, such as parabolic stretching at the edges of the graph. I should also note that LLE seems highly sensitive to the choice in neighbor count. In Figure 4.4, I used the choice of 4 as that tends to be the most common degree for knitting graphs.

4. **mMDS**: Metric MDS performed reasonably well. There were some small artifacts, such as minor crumpling in the waves edging pattern and slight ballooning in the stockinette pattern.

5. **KKBB**: This gave the best results across the board. The expected tilting effects are evident in Tilted Blocks, the Waves Edging is wavy just like in the physical example, and the stockinette is perfectly rectangular.

In the physical constructions, one can observe that there is some curling effects on the edges for many of the samples. This is due to the unequal force loops place on each other when they all share the same orientation. This is a 3D effect, as curling forms against the 2D plane normal. If we were to embed these in three dimensions, some extra information may be needed to indicate, for example, the orientation of stitches.

In summary, the KKBB algorithm performed the best. This is due to how each algorithm chooses to optimize their layout. For example, mMDS attempts to minimize the difference between distances in the high dimensional space and the distances in the low dimensional space, paying no special preference towards one pair over the other. On the other hand, KKBB penalizes differences between points that should be nearby much more than points that should be far apart.

4.5 Discussion

The minimization scheme, Barzilai Borwein, is not monotonic, so sometimes it would iterate erroneously without hitting a minimum. One way to fix this is to only use Barzilai Borwein to compute the step size in conjunction with a line search algorithm. In terms of speed, it's noted by Hasal et al. that there are many opportunities for parallelizing KKBB [7].

In all of the embedding algorithms, the biggest bottleneck in computation time is the graph theoretic distances calculation. I used Floyd-Warshall's Algorithm to compute the pairwise distances, which unfortunately takes $O(|V|^3)$ time. One way around this is to embed the graph at different scales, such as done in GRIP [5]. This works by limiting the number of pair-wise distances to compute by performing Kamada-Kawai's algorithm on some well-spread subset of vertices. The process is then repeated by recursing into neighborhoods around these vertices.

Existing libaries should be investigated for generating knit graph layouts. For example, Graphviz [6] is known for its fast force directed layout generation for large graphs.

An accurate embedding algorithm is the first step towards a powerful design tool for knitting structures. Given a more efficient algorithm, users may be able to design new structures while immediately visualizing the resulting shape of their pattern. It would also be useful to overlay a rendering of the knitted stitches, such as that done by Jiang et al. [8].

Chapter 5

Hand Knitting

In flat knitting, one uses two long and thin pins to form and intermesh loops. These are called *knitting pins* or *needles*. To avoid ambiguity with needles used in machine knitting, I will call them knitting pins. The process of hand knitting with pins is illustrated in Figure 5-1. Generally loops are formed row-by-row, where the prior row is on a knitter's left pin, and the following row of loops is formed onto the right pin. When the row on the left pin is exhausted, and all the loops are formed on the right pin, the knitter swaps these pins to form the next row.

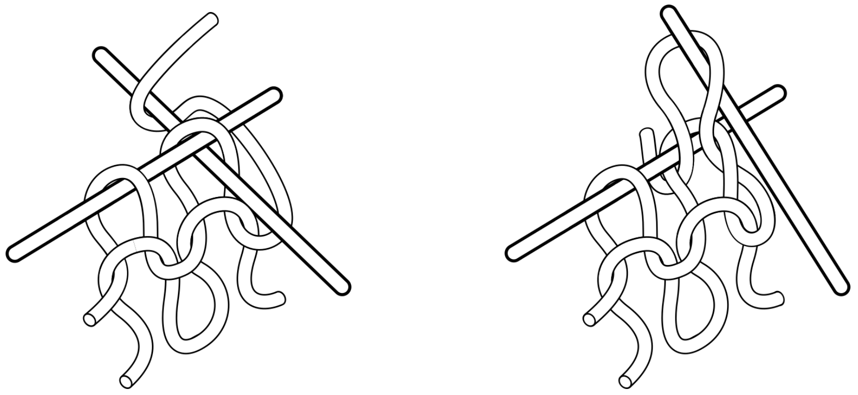


Figure 5-1: Illustration of stitch creation in hand knitting. On the left, a knitting pin is inserted through a loop, after which another loop is formed around the end of this knitting pin. On the right, the loop formed on the knitting pin is pulled through the other loop, forming a complete stitch.

These knitting pins are pointed only at one end and blocked at the other end. This ensures that the knitting process only happens at one end and that the loops farther down don't fall off. However, there are special pins that are pointed at both ends. These double ended pins are typically used for overlapping stitches to create cabled patterns. However, I will assume that all knitting pins are only pointed at one end.

Given some loops held on the left knitting pin, a knitter inserts their right pin through one or more of these loops and can then do one of the following.

- **Knit** or **purl**. When a single loop is formed around the right pin, and pulled through a single loop held on the left pin. After, the loop on the top of the left pin is pulled off. A knit means the loop on the right pin is pulled through the back of the left pin loop, and a purl means it is pulled through the front.

- **Yarn over**. A knitter wraps the yarn around their right pin. This results in an extra loop on the right pin.

- **Increase**. The same as knit, but the loop on the left pin is not dropped. The knitter can then re-insert the right pin into the left pin loop and form another loop to do an increase.

- **Knit** or **purl** n **together**. One loop is wrapped around the right pin, and it is pulled through n loops on the left pin, which are then dropped. Knit and purl indicates the direction the loop is pulled through. This is otherwise known as a decrease.

- **Slip**. A loop on the left pin is slipped to the top of the right pin.

- **Pass stitch over**. The second-to-top loop on the right pin is passed over the top-most loop on the right pin. This is otherwise known as **Pass slipped stitch over**, as it's typically done after a slip. Usually the passed loop is dropped after this.

The key insight in this chapter is the idea that knitting pins act as last-in-first-out stacks. This provides a basis for an abstract hand knitting model. This will be necessary for designing algorithms for converting existing knitting instructions into

Knit Meshes, as well as the inverse. I will first describe how to convert knitting instructions into Knit Meshes, and then I will discuss a way to convert Knit Meshes into hand knitting instructions.

5.1 Virtual Hand Knitter

In this section I will be outlining an abstract hand knitting model. This abstraction will be similar to that done for knitting machines by McCann et al. [16]. This will be necessary for keeping track of the relationships between loops and nodes as we build our conversion pipeline for hand knitting and Knit Meshes.

Let $p_j = (l_{j,1}, l_{j,2}, \cdots, l_{j,n})$ be virtual knitting pin j with n loops in its stack, where l_n is at the top of the stack. Then the hand knitting state is the set of pins, $P = \{p_1, p_2, \cdots\}$, which may be arbitrarily long. To complete the model, we also need some set of yarns, $Y = \{y_1, y_2, \cdots\}$ from which we knit from. I will define a set of knitting procedures for evolving this state.

- Knit($p_{\text{from}}, p_{\text{to}}, y$)
  where $p_{\text{from}},\ p_{\text{to}} \in P$ and $y \in Y$:
  1. Draw new loop l from yarn y.
  2. Add the newly drawn loop l onto the top of pin p_{to}.
  3. Remove the top loop from knitting pin p_{from}.
- Yarn-Over(p, y)
  where $p \in P$ and $y \in Y$:
  1. Draw new loop l from yarn y.
  2. Add the newly drawn loop l onto the top of pin p.
- Increase($p_{\text{from}}, p_{\text{to}}, y$)
  where $p_{\text{from}},\ p_{\text{to}} \in P$ and $y \in Y$:
  1. Draw new loop l from yarn y.

    2. Add the newly drawn loop l onto the top of pin p_{to}.

- Slip$(p_{\text{from}}, p_{\text{to}})$
  where $p_{\text{from}},\ p_{\text{to}} \in P$:

    1. Pop the loop l from the top of pin p_{from}.
    2. Add the newly popped loop l onto the top of pin p_{to}.

- Decrease$(p_{\text{from}}, p_{\text{to}}, y, n, n_{\text{drop}})$
  where $p_{\text{from}},\ p_{\text{to}} \in P,\ y \in Y$, and $n,\ n_{\text{drop}}$ are positive integers:

    1. Draw new loop l from yarn y.
    2. Add the newly drawn loop l onto the top of pin p_{to}.
    3. Remove n_{drop} from knitting pin p_{from}.

- Pass-Stitch-Over(p)
  where $p \in P$:

    1. Remove the second-to-top loop from knitting pin p.

It can be noted that we didn't relate loops to each other and some parameters are left unused in many of the procedures listed above. This is intentional, as I will be augmenting the virtual hand knitter in Sections 5.2 and 5.3.

5.2 Instructions to Knit Mesh

5.2.1 Hand knitting procedures

In Section 5.1, I outlined an abstract data type for virtual hand knitting. In that case, the virtual hand knitter contained infinitely many pins P and infinitely many yarn Y. However, for converting hand knitting instructions to Knit Meshes, I will make the assumption that there is one yarn and two pins. Patterns including cabling and multiple colors will not be supported in this case, however other interesting structures will still be supported.

For us to convert hand knitting instructions into Knit Meshes, we need to relate common knitting instructions to the abstract knitting procedures in Section 5.1. To do so, we create a mapping of each instruction to the relevant virtual hand knitting procedure, as well as the necessary procedures for evolving the Knit Mesh itself.

Let the knit graph T and geometry G be the Knit Mesh that we are evolving. To relate this with the virtual knitting state, we define the mapping $V : q \to v$, where q is some loop in the virtual knitting state and v is some node in T. In addition, we use the procedure $\textsc{Top}(p, i)$, which returns the top i'th loop in pin p. For convenience, the top-most loop is returned when i is left blank. Lastly, let y be some yarn.

The mapping from knitting abbreviation to procedure is listed in Table 5.1. Note that the mapping for "p" and "p2tog", i.e. purl and purl two together, are the same as "k" and "k2tog" but with PURL instead of KNIT for loop faces.

Instruction	**Procedure**
"k" : Knit	// **Update Knit Mesh**
	1 $u \leftarrow V(\textsc{Top}(p_{\text{left}}))$
	2 $v \leftarrow \textsc{New-Node}(T, y)$
	3 $l \leftarrow \textsc{New-Loop}(T, u, v)$
	4 $\textsc{Set-Face}(G, l, \text{KNIT})$
	// **Update virtual knitter state**
	5 $\textsc{Knit}(p_{\text{left}}, p_{\text{right}}, y)$
	// **Update mappings**
	6 $V(\textsc{Top}(p_{\text{right}})) \leftarrow v$
"yo" : Yarn over	// **Update Knit Mesh**
	1 $v \leftarrow V(\textsc{Top}(p_{\text{right}}))$
	// **Update virtual knitter state**
	2 $\textsc{Yarn-Over}(p_{\text{right}}, y)$
	// **Update mappings**
	3 $V(\textsc{Top}(p_{\text{right}})) \leftarrow \textsc{New-Node}(T, y)$

	// **Update Knit Mesh**
	1 $u_0 \leftarrow V(\textsc{Top}(p_{\text{left}}, 1))$
	2 $u_1 \leftarrow V(\textsc{Top}(p_{\text{left}}))$
	3 $v \leftarrow V(\textsc{Top}(p_{\text{right}}))$
"k2tog" :	4 $l_0 \leftarrow \textsc{New-Loop}(T, u_0, v)$
Knit 2 together	5 $l_1 \leftarrow \textsc{New-Loop}(T, u_1, v)$
aka Decrease	6 $\textsc{Set-Face}(G, l_0, \text{KNIT})$
	7 $\textsc{Set-Face}(G, l_1, \text{KNIT})$
	// **Update virtual knitter state**
	8 $\textsc{Decrease}(p_{\text{left}}, p_{\text{right}}, y, 2, 2)$
	// **Update mappings**
	9 $V(\textsc{Top}(p_{\text{right}})) \leftarrow \textsc{New-Node}(T, y)$
	// **Update Knit Mesh**
	1 $u \leftarrow V(\textsc{Top}(p_{\text{right}}))$
"psso" :	2 $v \leftarrow V(\textsc{Top}(p_{\text{right}}, 1))$
Pass slipped stitch over	3 $l \leftarrow \textsc{New-Loop}(T, u, v)$
	4 $\textsc{Set-Face}(G, l, \text{KNIT})$
	// **Update virtual knitter state**
	5 $\textsc{Pass-Stitch-Over}(p_{\text{right}})$
"sl" or "s" : Slip	// **Update virtual knitter state**
	1 $\textsc{Slip}(p_{\text{left}}, p_{\text{right}})$

Table 5.1: Mapping of common abbreviations to procedures.

5.2.2 Interpreter

At this point we have a virtual hand knitting state and a mapping from hand knitting instructions to procedures that evolve a Knit Mesh. Now we need to design an algorithm that can interpret an entire list of instructions.

Generally, knitting instructions are organized by row and listed in sequence. For example, the pattern Herringbone Eyelets, from "Up, Down, All-Around Stitch Dictionary" [4],

FLAT

(multiple of 9 sts + 3, 8-row repeat)

Row 1(RS): K2tog, yo, *k7, k2tog, yo; repeat from * to last st, k1.

Row 2 and all WS Rows: Purl.

Row 3: K2, *yo, ssk, k4, k2tog, yo, k1; repeat from * to last st, k1.

Row 5: K2, *k1, yo, ssk, k2, k2tog, yo, k2; repeat from * to last st, k1.

Row 7: K2, *k2, yo, ssk, k2tog, yo, k3; repeat from * to last st, k1.

Row 8: Purl.

Repeat Rows 1-8 for Herringbone Eyelets Flat.

To simplify our input slightly, I assume that the instructions are supplied as an ordered list of rows, where each row is a string of instructions. The row string is comma separated, and each instruction may or may not have a number at the end to indicate repetition. Lastly, I do not include support for the "repeat from *" parts. One can instead programmatically generate instruction strings based on some given parameters.

When a person knits, they use two knitting pins, one held in each hand. For those who are right handed, the left pin holds loops from the previous course, and the right pin is used to draw new loops through the loops on the left pin. When the loops on the left pin have been exhausted, the knitter swaps the pins they are holding and continues knitting.

In interpretting instructions, we need to track the current left and right knitting pin. I make the assumption that each row represents a completion of the loops from each pin. The algorithm for interpreting instructions is thus,

```
INTERPRET-INSTRUCTIONS($R$)
    Input: A list $R$ of instruction strings, with instructions separated by comma.
    Output: The corresponding knit mesh.
1   $p_{\text{left}} \leftarrow$ empty virtual knitting pin
2   $p_{\text{right}} \leftarrow$ empty virtual knitting pin
3   $M \leftarrow$ new Knit Mesh with topology $T$ and geometry $G$.
4   for each $row \in R$:
5       PROCESS-ROW($M, row, p_{\text{left}}, p_{\text{right}}$)
6       Swap $p_{\text{left}}$ and $p_{\text{right}}$
7   return $M$
```

On lines 1 and 2, two knitting pins are added to the hand knitting state. The PROCESS-ROW procedure, called on Line 5, takes the Knit Mesh to evolve, a single string of instructions, and the current "left" and "right" knitting pins that is being virtually held. This procedure is outlined below.

```
PROCESS-ROW($M, row, p_{\text{left}}, p_{\text{right}}$)
    Input: Row string $row$, and currently held left and right pins $p_{\text{left}}$ and $p_{\text{right}}$
    Result: $M$ will be evolved using the instructions from $row$
1   $instructions \leftarrow$ SPLIT($row$)
2   for each $instruction \in instructions$
3       $k \leftarrow$ instruction count
4       $f \leftarrow$ procedure mapped to $instruction$
5       while $k > 0$
6           Perform $f()$ with necessary inputs
7           $k \leftarrow k - 1$
```

Line 1 splits the instruction strings into individual instructions by commas. The instruction count is determined using Regular Expression on Line 3. On Line 4, a procedure mapped to the current instruction is determined using Table 5.1. The corresponding procedure is then used to mutate the left and right pins and makes the corresponding additions to the Knit Mesh.

5.2.3 Results

The algorithms outlined above were implemented in Python [24]. Some example results are shown in Table 5.2. Note, that "cast" is aliased to "yo," which simply indicates a new node.

5.3 Knit Mesh to Instructions

We can follow a similar process as the one outlined above to do the inverse. We originally assumed that there are only two pins and one yarn to convert hand knitting instructions to Knit Meshes. For this case, however, we'll assume that we can hold infinitely many pins and we have infinitely many yarn. This will allow us to add and remove virtual stacks as we need them.

Converting to instructions involves processing the knit graph in order of nodes along each yarn. At each node, we perform procedures depending on a number of different factors. The algorithm is thus,

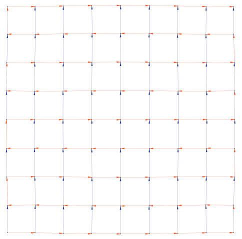
```
HAND-KNIT(T,G)
1 for each yarn in $T$:
2     for each node $v$ in $T$:
3         PROCESS-NODE($T, G, v$)
```

Note that this isn't necessarily a unique ordering of processing nodes. I made the arbitrary choice to process nodes one yarn at a time, but in many cases it would be easier for the knitter to knit along all of their yarns in parallel. One needs to take into account the loop dependency of nodes in order to determine a valid ordering. The best ordering for nodes remains to be explored, but I suspect scheduling algorithms would be highly relevant.

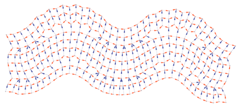
Similar to how we evolved the virtual knitting state in Section 5.2.1, we will again evolve the knitting state as we process nodes. Meanwhile, we will generate a list of instruction strings. This part is relatively trivial, as the instructions more-or-less matches the procedure names in Section 5.1.

Python code **Embedded knit graph**

```python
def stockinette(wale_count, course_mult):
    rows = ["cast%i" % wale_count] + \
           ["k%i" % wale_count,
           "p%i" % wale_count] * course_mult
    return interpret_instructions(rows)
```



```python
def waves_edging(wale_mult, course_mult):
    wale_count = wale_mult * 18 + 2
    rows = ["cast %i" % wale_count] + \
           ["k %i" % wale_count,
            "k1" + (", ssk 3" + ", k1, yo" * 6 + ",
               ssk 3") * wale_mult + ",k1",
            "p %i" % wale_count,
            "k %i" % wale_count] * course_mult
    return interpret_instructions(rows)
```



```python
def tilted_blocks(wale_mult, course_mult):
    wale_count = 16 * wale_mult + 1
    rows = ["cast %i" % wale_count] + \
        ([("ssk, yo, " * 4 + "k8, ") * wale_mult + "k
           ",
         "k9, p7, " * wale_mult + "k"] * 3 + \
        ["k1" + (", k8" + ", yo, k2tog" * 4) *
           wale_mult,
         "k1" + ", p7, k9" * wale_mult] * 3) *
            course_mult

    return interpret_instructions(rows)
```

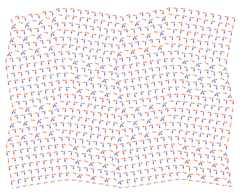


Table 5.2: Interpreter results showing the Python [24] generated hand knitting instructions compared with the automatically generated knit graphs, embedded. Knitting patterns are from "Up, down, all-around stitch dictionary" [4].

At each node, the goal is to have generated instructions that completely matches the Knit Mesh up to that point. Thus, for a given node v, we need to determine the correct instructions needed to account for any loop edges connected to v that are connected to nodes that have already been processed. We also need to determine which pin to stack v on in a way that guarantees that v will be on the top the next time we need to make physical loops to it, while still limiting the number of pins that we're holding. Finally, we need to determine which nodes are finished and may be dropped.

5.3.1 Node Processing

Any loops going forward in time, to or from nodes that will be processed, will be handled later on when those nodes get processed. Thus when we are processing node v, we need to account for only the loops adjacent to previously processed nodes. Note that, the terms "from node" and "to node" respectively refer to the source and destination nodes of a given directed edge.

Let $P : v \to p$ map each node to the pin that is current holding it, $Y : v \to y$ map each node to the yarn it is on. At this point, we will assume that v and all of the nodes it is looped to are at the top of their respective pins. Table 5.3 shows the cases for each knitting procedure depending on the number of incoming and outgoing loops and their adjacency.

It has yet to be shown as to which pin we will assign each node, as well as the cases when a node may be dropped.

5.3.2 Pin to stack

Let Pin-To-Stack(v, P) be the procedure that determines which pin we can stack. The procedure must guarantee that the chosen knitting pin has no nodes already on it that will be needed for loops before the current node v.

The cases that we need to consider are enumerated below. Let v be the current node getting processed, u be the node on top of some candidate knitting pin for v

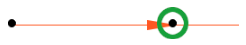
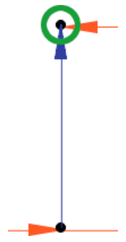
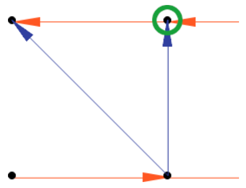
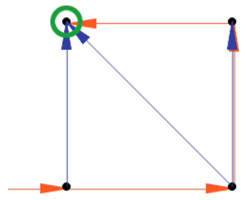
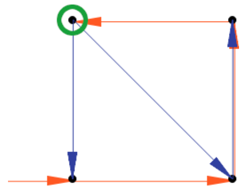
Loops	**Knit graph**	**Knitting procedure**
No incoming loops		$\textsc{Yarn-Over}(P(v), Y(v))$
1 incoming loop, from node has no future loops.		$\textsc{Knit}(P(v_{\text{from}}), P(v_{\text{to}}), Y(v_{\text{to}}))$
1 incoming loop, from node has future loops.		$\textsc{Increase}(P(v_{\text{from}}), P(v_{\text{to}}), Y(v_{\text{to}}))$
> 1 incoming loops		$\textsc{Slip}(P(v_{\text{from}\,1}), P(v_{\text{from}\,2}))$ $\textsc{Decrease}(P(v_{\text{from}\,1}), P(v_{\text{to}}), Y(v_{\text{to}}), n, n_{\text{drop}})$
1 or more outgoing loops		For each outgoing loop, $\textsc{Slip}(P(v_{\text{from}}), P(v_{\text{to}}))$ $\textsc{Pass-Stitch-Over}(P(v_{\text{from}}))$

Table 5.3: Handling incoming loops to a given node that is being processed (circled in green).

to be stacked onto. When we use the term "before" or "after" in regards to a given loop, we are referring to the time that the loop will be processed, which is dependent on the node farthest along on its yarn.

1. Node to stack, v, will be dropped before candidate pin's top node u's next loop. Then we can safely stack v onto this candidate pin.

2. Node to stack, v, will be dropped after candidate pin's top node u's next loop. Then stacking v onto this candidate pin would require us to eventually slip it out of the way. In this case, we should not stack v onto this candidate pin.

3. Node to stack, v, will be dropped at the same time as the candidate pin's top node $u's$ next loop. Then the four following conditions must be met for v to stack onto this candidate pin.

   (a) Both nodes must have an outgoing loop to the same node. That is, these nodes will be decreased together.

   (b) Both nodes must get dropped when they decrease together. Otherwise, we would need to do erroneous slips to assure that these nodes are properly ordered.

   (c) Any other nodes that will decrease with u and v must get dropped during this decrease. Again, this is to avoid needing to reorder nodes on a single pin.

   (d) The node that u and v decreases into must not have any outgoing loops back into any of these nodes. If this were the case, we would again need to reorder nodes so the outgoing loop to node does not get dropped.

Case number 3 may seem contrived, but it is necessary to avoid requiring erroneous slips and extra pins for decreases, which happens to be a relatively common instruction.

5.3.3 Node dropping

Finally, we need to tell when it is time to drop a node. Let DROP-NODE(v, t) return TRUE if it is safe to drop a node at time t, otherwise return FALSE.

Before we continue, we need to more formally define the meaning of time. At the beginning of Section 5.3, we chose to process nodes yarn-by-yarn, from the start of the yarn to the end. Let $y_0, y_1, \cdots, y_n$ be the yarns in the order we process them. Then for each yarn y_i, there are nodes $v_{i,0}, v_{i,1}, \cdots, v_{i,m_i}$ in order of processing. For node $v_{i,j}$, the processing time is

$$t(v_{i,j}) = \sum_{k=0}^{i-1} m_k + j$$

where m_k is the number of nodes in yarn y_k.

Note that there are two relevant times for each node. t is defined to be the processing time of a given node, that is, the time at which we process the given node. However, this node is looped to other nodes that have their own processing times. Thus, a given loop l between nodes l_{from} and l_{to} may only be added at the time $t(l) = \max(t(l_{\text{from}}), t(l_{\text{to}}))$. It follows that a node is completely finished once its last loop has been accounted for, that is, $t_{\text{finish}}(v) = \max_{v \in l}(t(l))$, which is therefore the time we can safely drop node v.

5.3.4 Decreases and PSSOs

In Section 5.3.2, we're able to guarantee that a node is at the top of its respective pin when it is time to create loops with it. However, there are some cases when we need for this to not be true. For instance, if nodes u and v will be decreased together, we need them to be on a single pin before performing the decrease. Certain instances of this has already been accounted for in the cases outlined in Section 5.3.2, but there may be situations where not all the nodes we're decreasing will be dropped.

In the decrease case, we simply need to gather all the nodes onto a single pin using a series of SLIP operations. However, we must gather the nodes in order of their t_{finish} time to ensure that the nodes at the top of the pin are the nodes to be dropped the soonest. Thus, if we decrease them together and only some of the nodes get dropped, we'll only need to supply the count of nodes to be dropped, n_{drop} to DECREASE. After decreasing, we slip the undropped nodes back to their original pins.

The other case we need to consider is for PSSO operations, i.e. when there are outgoing loops to the past from the node we're processing. For these loops, we typically just PSSO each node sequentially. Whenever we pass a loop over another, their pin stack ordering gets swapped. This can be problematic if we're not dropping the node we just PSSO'd over. In this case, we'll need to re-swap their ordering using a temporary swap pin. Before doing this, one can check if the ordering is broken using the cases we outlined for PIN-TO-STACK.

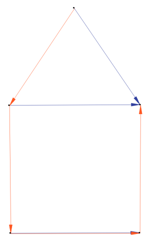
5.3.5 Other additions

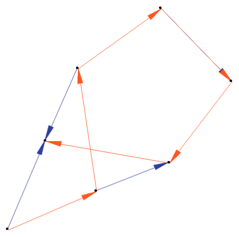
To avoid mistakes, the knitter has to track in their head how many loops are on each of their pins. Sometimes loops get erroneously dropped or added, so it is necessary for the knitter to catch this mistake early so they can go back and correct it. To aide in this, my instruction generator periodically outputs the number of loops that should be on each pin.

5.3.6 Results

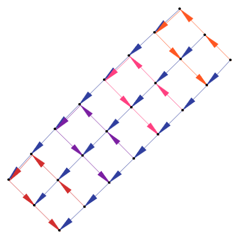
Note that the algorithm treats each starting node as a yarn over, and for each ending node it drops once it is finished. However, when hand knitting, one needs to cast and bind off the starting and ending stitches to prevent the fabric from unraveling.

Embedded knit graph	**Generated instructions**

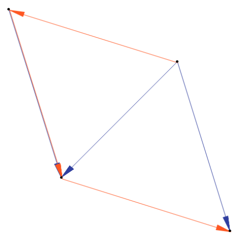
	`Yarn over Yarn A on to Pin A (x3)`
	`Knit Yarn A from Pin A to Pin B`
	`Drop a stitch from Pin B`
	`Knit 2 loops together from Pin A to Pin B with Yarn A`
	`Drop a stitch from Pin B`



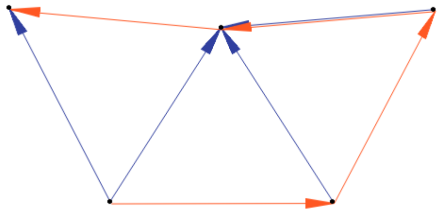
```
Yarn over Yarn A on to Pin A (x2)
Yarn over Yarn A on to Pin B
Yarn over Yarn A on to Pin A
Knit Yarn A from Pin A to Pin B
Drop a stitch from Pin B
Current knitting state: Pin A: 2 loops, Pin B: 1 loop

Knit Yarn A from Pin A to Pin B
Drop a stitch from Pin B
Slip from Pin A to Pin B
Knit 2 loops together from Pin B to Pin C with Yarn A
Drop a stitch from Pin C
```



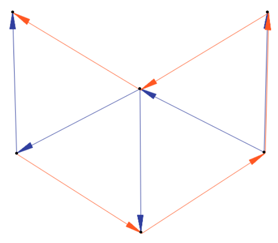
```
Yarn over Yarn A on to Pin A (x3)
Knit Yarn A from Pin A to Pin B (x3)
Knit Yarn B from Pin B to Pin A (x3)
Knit Yarn B from Pin A to Pin B (x3)
Knit Yarn C from Pin B to Pin A (x3)
Current knitting state: Pin A: 3 loops

Knit Yarn C from Pin A to Pin B (x3)
Knit Yarn D from Pin B to Pin A (x3)
Knit Yarn D from Pin A to Pin B
Drop a stitch from Pin B
Knit Yarn D from Pin A to Pin B
Current knitting state: Pin A: 1 loop, Pin B: 1 loop

Drop a stitch from Pin B
Knit Yarn D from Pin A to Pin B
Drop a stitch from Pin B
```



```
Yarn over Yarn A on to Pin A
Yarn over Yarn A on to Pin B
Slip from Pin B to Pin A
Knit 2 loops together from Pin A to Pin C with Yarn A, but
     only drop the first 1 loop
Drop a stitch from Pin C
Current knitting state: Pin A: 1 loop

Knit Yarn A from Pin A to Pin B
Drop a stitch from Pin B
```



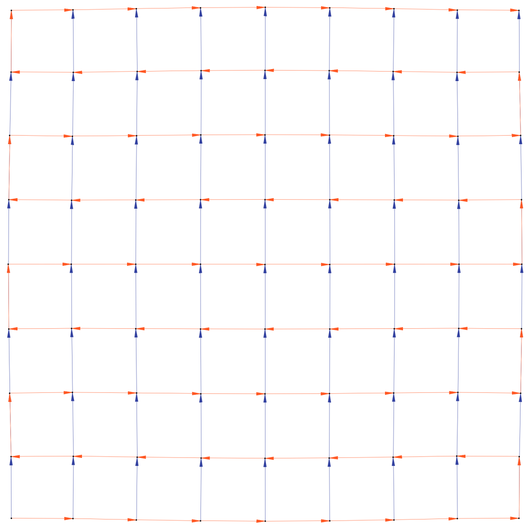
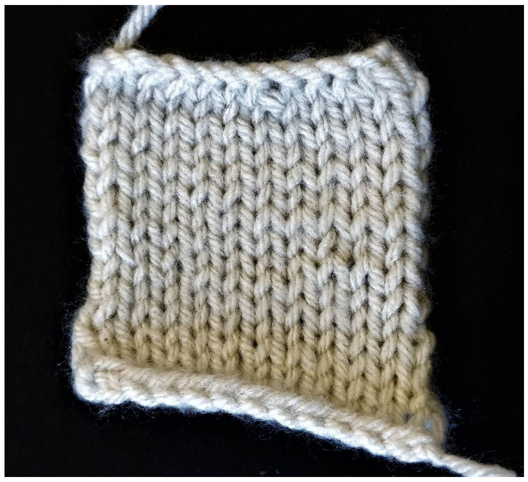
```
Yarn over Yarn A on to Pin A
Yarn over Yarn A on to Pin B
Yarn over Yarn A on to Pin C
Slip from Pin C to Pin A
Slip from Pin B to Pin A
Current knitting state: Pin A: 3 loops

Knit 3 loops together from Pin A to Pin D with Yarn A, but
     only drop the first 2 loops
Drop a stitch from Pin D
Knit Yarn A from Pin A to Pin B
Drop a stitch from Pin B
```

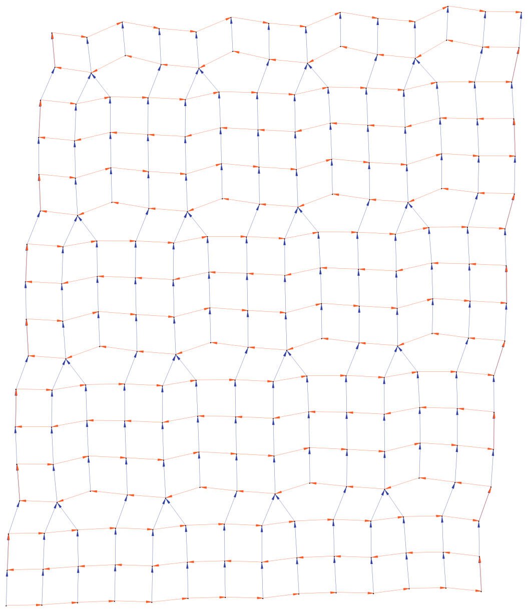
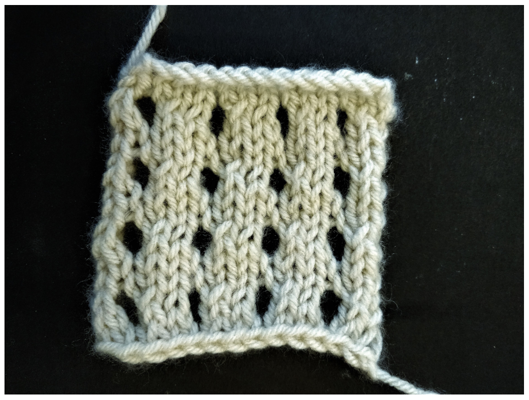
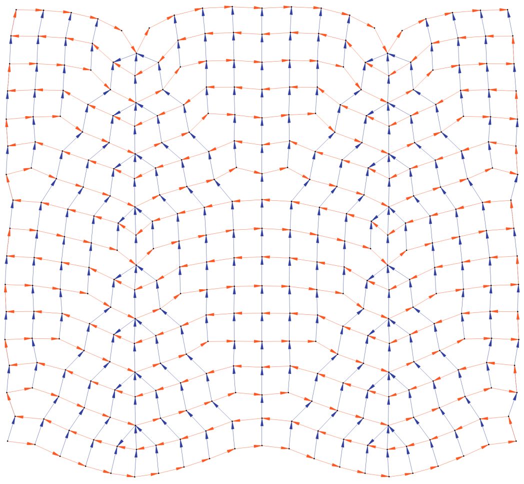


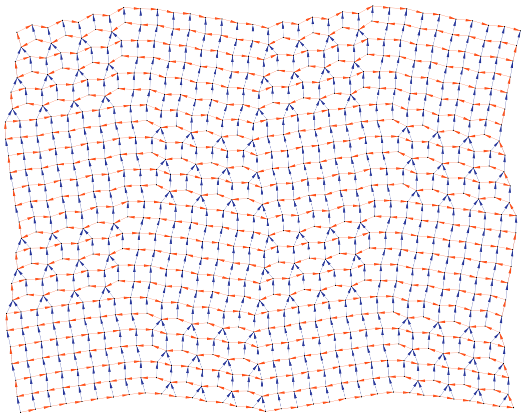
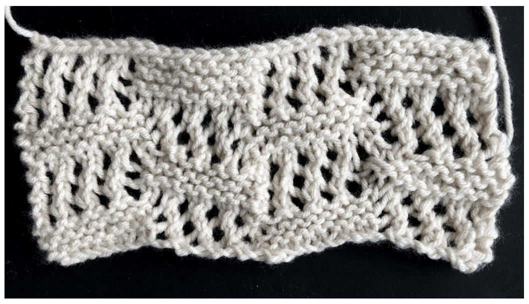
```
Yarn over Yarn A on to Pin A
Yarn over Yarn A on to Pin B
Yarn over Yarn A on to Pin C
Increase Yarn A from Pin C to Pin A
Drop a stitch from Pin A
Current knitting state: Pin C: 1 loop, Pin B: 1 loop, Pin A
     : 1 loop

Knit Yarn A from Pin C to Pin D
Slip from Pin D to Pin B
PSSO: Pass and drop second loop on Pin B over first loop.
Slip from Pin B to Pin A
PSSO: Pass but don't drop second loop on Pin A over first
     loop.
Current knitting state: Pin A: 2 loops

Slip from Pin A to Pin C
Slip from Pin A to Pin B
Slip from Pin C to Pin A
Drop a stitch from Pin B
Knit Yarn A from Pin A to Pin B
Current knitting state: Pin B: 1 loop

Drop a stitch from Pin B
```

Table 5.4: Generated instructions for various knit graphs.

Table 5.4 shows some generated instructions given different knit graphs. Hand knitted results are shown in Table 5.5.

Embedded knit graph	**Hand knitted**
	

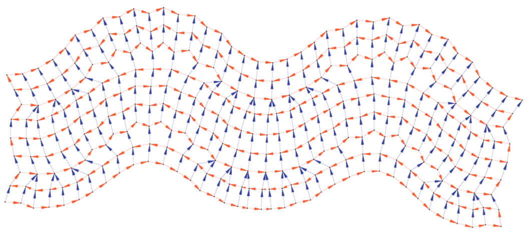
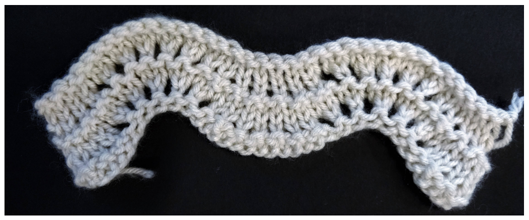



Table 5.5: Instructions were automatically generated from each of these knit meshes, which were then followed to hand knit these samples.

5.4 Discussion

It's yet to be formally shown that the instruction generator can generate every valid knit graph. However, from experience, the algorithm seems to finish on all 45 test cases.

There is some more work to be done. The number of knitting pins may not necessarily be fully minimized, as some resorting and gathering is required. In addition, double-ended knitting pins would be useful to support, as they make cabling patterns much easier. Finally, the algorithm doesn't output traditionally formatted knitting instructions, but insteads writes them out in a verbose manner.

The algorithm that converts instructions into Knit Meshes is much more limited. It is assumed that only one yarn and two pins are used, so cables and multiple colored patterns are not supported. Furthermore, knitting patterns are often times expressed as charts. It would be useful to also be able to convert knitting charts into Knit Meshes.

Chapter 6

Machine Knitting

When hand knitting, a knitter usually holds 2 or 3 knitting pins at a time. For flat, weft knitting machines, the process is different. Instead of 2 or 3 knitting pins, the machine has hundreds of hook-shaped *knitting needles*. These needles are typically organized into two opposing rows, called *beds*, at a slight angle from each other. This is known as a *V-bed machine*, due to the two beds forming an upside down "V" formation. There also exists *X-bed machines*, where 4 beds of needles face each other.

Each knitting needle sits on a track facing the opposing bed, from which it can slide up and down. As the needle slides up, the needle's latch opens and any loops it is currently holding slides down from its hook into a gap. Simultaneously, the hook at the end of the needle draws a new loop from the yarn overhead. While sliding down, the hook latch closes, pulling its currently held loop through the previously held loop, forming a complete stitch.

Knitting needles can perform a number of different tasks, depending on what yarn they're holding and how they are actuated with respect to each other. For instance, if there is no yarn in a knitting needle, it performs a *tuck*, equivalent to a yarn over in hand knitting. If there is loops, then the needle performs a *knit*. Two opposing needles, actuated one after the other, causes what's called a *transfer* from the former to the other. If one wants to perform a *decrease*, i.e. knit multiple loops together, then they need to transfer those loops onto a single needle and then knit with that needle. When knitting, a needle may also transfer its formerly held loop to the opposite bed,

Hand knitting term	**Machine knitting term**
Knit	Knit
Yarn over	Tuck
Slip	Transfer
Knit n together	Transfer and knit
Increase	Split and knit

Table 6.1: Terms used to describe hand knitting and their analogous counterparts for machine knitting.

doing what is called a *split*. This would allow the machine to knit multiple loops from a single loop. These are sometimes called *increases*, but note that tucks are also sometimes called increasing, since they both effectively increase the number of wales in a row. A table of the analogies between hand knitting and machine knitting can be seen in Table 6.1.

For a more detailed take on machine knitting, the reader is referred to David Spencer's "Knitting technology" [23].

The knitting needles are actuated by what's called a *cam system*. The cam system is a machine that moves along a track perpendicular to the needle beds. Inside the cam system are mechanical tracks, from which extrusions on the needles funnel into and slide along as the cam system passes through. The tracks inside the cam system mechanically changes shape depending on the desired action of the needle sliding through. At the same time, the cam system moves with it one of the *yarn carriers* for the knitting needles to grab loops from.

At this point we may notice that knitting needles don't act like last-in-first-out stacks like knitting pins. When a loop is added to a knitting needle with other loops, it is impossible to later remove that loop separately from the other loops. Thus, needles are more like ordered lists where loops can be added or cleared, but not removed individually. The other difference to note is the way that loops move from needle to needle. In hand knitting, we make the assumption that a person can hold infinitely many pins and can slip loops between any pair of them. However, in machine knitting, a needle can only transfer to the needle directly across from it on the opposite bed.

To allow more flexibility, standard weft knitting machines can slide one of their beds side to side, thus allowing each needle to transfer to other needles on the opposite bed. However, if loops span across the beds of the machine, transferring a loop too far left or right will cause the opposing bed to stretch the yarn, thus breaking either the yarn or the needles on the bed.

The last difference to note is that there is no analogy to "pass stitch over" for knitting machines. That is, two loops currently held by separate needles on a knitting machine may not knit through each other. Knitting can only happen from the yarn carried overhead. Consequently, weft knitting machines cannot knit loops that go backwards along their yarn.

6.1 Generating Knitting Machine Code

Knit programmers generally work with a grid-aligned pictorial programming language. An example is shown in Figure 6-1. Each column in this grid corresponds to a needle index for both beds. Each row corresponds to a single traversal of the cam system from one side to the next, where time moves vertically from bottom to top. Operations in each cell instructs the cam system whether or not to slide each needle, at what time, and how far. For instance, one can put "Front knit" or "Back knit" to instruct the cam system to knit with the front needle or the back needle respectively. One can also "Front tuck" or "Back tuck," as well as "Transfer front to back" and vice versa. There are variants to the transfer command which allows the user to transfer diagonally to the left or to the right some number of places, where the machine offsets one of the knitting beds and then performs a transfer.

Knitting machine manufacturers provide software programming software for their knitting machines. Examples include Matsuya's SPJWin [2], Shima Seiki's Apex3 KnitPaint [1], and Stoll's M1plus pattern software [3].

The issue with knitting machine code is its tight coupling with the way knitting machines work. When designing new garments, a knitting programmer needs to manually plan transfers of stitches, keep track of each yarn carrier, and embed the

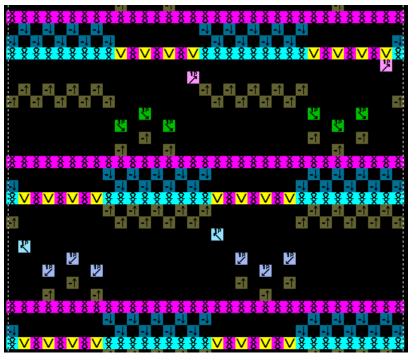


Figure 6-1: Screenshot of machine knitting code from Matsuya's SPJWin [2].

entire garment into the flat, 2D grid format. This process is highly error prone, as the programmer often has to go back and make changes to correct for things like dropped loops, yarn breakage, and so on.

Knit Meshes, however, directly describes knitted structures in a way that is decoupled from both machine knitting and hand knitting. This gives the user freedom to make manipulations without having to account for external processes. It would be highly desirable to design an algorithm that can automatically convert Knit Meshes into knitting machine code. The goal in this chapter is to design such an algorithm.

6.1.1 Related

In 2016, McCann et al. [16] wrote a paper that describes a low level language for virtual machine knitting. I followed a similar process for hand knitting in Section 5.1. In addition, I will outline a similar virtual knitting machine in Section 6.1.3. The virtual knitting machine is necessary for tracking which needles are holding onto which loops, as well as planning transfers between courses.

Popescu et al. [19] described a process for converting quad meshes into knitting machine code. This was limited as the machine layout and transfer scheduling was still a manual process. Knitting 3D geometry was further generalized in 2018 by Narayanan et al. [18], where any oriented 2D manifold whose Reeb graph has an upward planar embedding could be knitted. Or, more plainly, any cylindrical-like surface could be knitted using their method. While this paper solves the difficult problem of machine knitting 3D surfaces, it doesn't address how to automatically knit structures other than the standard stockinette pattern.

6.1.2 Limitations

My algorithm makes a number of assumptions before generating machine knitting code.

1. The graph contains no backwards facing loops. As stated earlier in this chapter, knitting machines simply can't knit backwards facing loops.
2. There are no increases. Increases could in theory be supported using splits, but my project doesn't support that functionality.
3. Loops on the knitting machine may be transferred to any needle on the opposite bed. A smarter transfer planning algorithm would prevent yarn from stretching too far.
4. There is only one yarn.
5. The knitting machine is a V-bed machine.

This project is preliminary, but does provide a starting framework for automatically generating machine knitting code.

6.1.3 Virtual Knitting Machine

Similar to what we did in Section 5.1 and the work done by McCann et al. [16], we will again design a virtual knitting machine which will allow us to track the state of the knitting needles.

Let $\eta = l_1, l_2, \cdots, l_n$ be some abstract knitting needle holding n loops. Then $H_{\text{back}} = \eta_0^{(b)}, \eta_1^{(b)}, \cdots$ and $H_{\text{front}} = \eta_0^{(f)}, \eta_1^{(f)}, \cdots$ are the knitting needles on the back and front beds respectively. Let $H = H_{\text{back}} \bigcup H_{\text{front}}$ be the set of all knitting needles. We again assume that y is some yarn path $\in Y$. The virtual knitting machine supports the following operations.

- KNIT(η, y)
  where $\eta \in H$ and $y \in Y$:

  1. Draw new loop l from yarn y.
  2. Add the newly drawn loop l onto the needle η.

- TUCK(η, y)
  where $\eta \in H$ and $y \in Y$:

  1. Draw new loop l from yarn y.
  2. Add the newly drawn loop l onto the needle η.

- TRANSFER$(\eta_{\text{from}}, \eta_{\text{to}})$
  where $\eta_{\text{from}} \in H_{\text{back}}$ and $\eta_{\text{to}} \in H_{\text{front}}$, or vice versa :

  1. Remove the loop list l_η from the needle η_{from}.
  2. Append the loop list l_η to η_{to}, reversed.

6.1.4 Machine Knitting Graph

One key difference from hand knitting is the relationship among needles. In hand knitting, one requires two pins to form a complete knitted stitch. However, on machine knitting, a single needle draws a loop through its previously held loops to perform a knit.

For each loop edge l with nodes u and v, it must be true that v will be knitted by the needle currently holding u. In terms of knit graphs, loop edges must trace out a vertical path for a given column of loops.

However, it's not always the case that loop edges *can* trace out a vertical path, as they may span across different wales. This is why we need to modify our knit graph to create one more intermediate format. This intermediate format includes *transfer edges*, which may be used to align nodes to share the same needle. Transfer edges occupy their own course, separate from the knit and tuck operations.

Let $T_{\text{machine}} = (\bar{V}, Y, L, \Gamma)$ be the knitting machine knit graph, where $\bar{V}$ represents abstract positions on a knitting machine, Γ is the set of transfer edges. It should be noted that $\bar{V}$ no longer strictly represents points along a yarn, but instead abstract positions that yarn edges may or may not traverse across. Next, we will define some mappings. Let $\pi : \bar{v} \to (w, c, b)$ be a mapping of $\bar{v} \in \bar{V}$ to course c for virtual knitting machine needle position at wale w and bed b. Let $\phi : \eta \to (w, b)$ map knitting needle η to virtual knitting machine position, and $\theta : \bar{v} \to \eta$ is the mapping of nodes to abstract knitting needles. θ can be generated from π and ϕ.

ϕ may be assigned ahead of time, where an η occupies wales $0, 1, \cdots, n$ for the beds FRONT and BACK.

We also need to assign π, that is, a default course c, wale w, and bed b for each node. This is done using a number of different running counters. For each node, it is assigned a wale from a wale counter, which is then incremented, as well as a course number. Whenever there are edge loops into the current course, the course counter is incremented and the wale counter is reversed. The Knit Mesh geometry may require a node to be technical face or technical back, which determines which bed that the node is assigned to. The face of each loop depends on both the even-or-odd parity of the course number as well as whether or not that loop is assigned KNIT or PURL. This is because, when hand knitting, a garment with all KNIT will form a garter pattern, meaning the loop orientation alternates direction every row. But in machine knitting, using just the front or back bed keeps all the loops oriented the same way.

Between each course, the algorithm needs to determine the transfer edges necessary to align nodes. This is done simply by checking each outgoing loop edge from the currently held course into the next course. If any loop edge has nodes that are not aligned, then we need to transfer those nodes.

However, there are some cases we must handle. For a given node on needle η_{from} and its desired needle position η_{to},

1. The desired needle position $\phi(\eta_{\text{to}})$ has the same bed as $\phi(\eta_{\text{from}})$. In this case, we choose an empty intermediate needle, $\eta_{\text{intermediate}}$, on the opposite bed whose wale is between $\phi(\eta_{\text{to}})$ and $\phi(\eta_{\text{from}})$. Thus, the transfer would be $\eta_{\text{from}} \to$

$\eta_{\text{intermediate}} \rightarrow \eta_{\text{to}}$.

2. $\phi(\eta_{\text{to}}) = \pi(\bar{u})$ where $\bar{u}$ is some other node. That is, some other node is occupying the desired needle, η_{to}. In this case, we need to transfer $\bar{u}$ to an empty needle η_{empty}.

3. If we have a cycle of transfers, like $\eta_A \rightarrow \eta_B \rightarrow \eta_C \rightarrow \eta_D \rightarrow_A$, then we need to break the cycle by moving one of the nodes to a temporary needle $\eta_{\text{temporary}}$, and then perform its transfer on a separate cam traversal.

The last step is sanitation. We need to do four more things to ensure that the transfers will work cleanly on the knitting machine.

1. When many transfers are happening too close to each other, errors become more likely depending on the knitting machine. It's often beneficial to modulate and separate the transfers across several cam traversals on the machine.

2. On every course where there is knitting or tucking, the yarn carrier must be carried from one side to the next. On transfer rows, the cam may traverse to the other side without the yarn carrier. Thus, we need to make sure that the cam system is on the same side as the yarn carrier after the transfer rows. We can do this by enforcing the rule that there can only be an even number of transfer rows, simply by inserting a blank row if it is odd.

3. We need to assume that the back knitting bed may slide a fixed amount for a given traversal. This means that we can't have multiple transfers of different offsets within a given cam traversal. If this is the case, then these transfers must again be split into different rows.

4. Lastly, our transfers must be done in their topological order. For instance, if we have the set of transfers $\eta_A \rightarrow \eta_B$ and $\eta_B \rightarrow \eta_C$, then we need to be sure to do the latter transfer occurs before the former. Otherwise, loops will be combined into a single needle, causing an erroneous decrease.

Examples of machine knitting graphs are shown in Table 6.2.

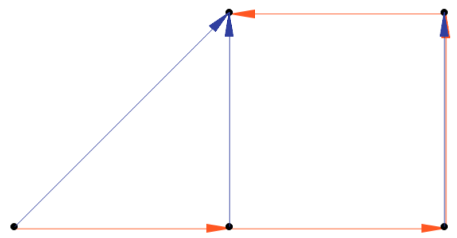
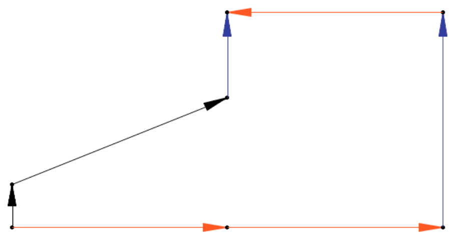
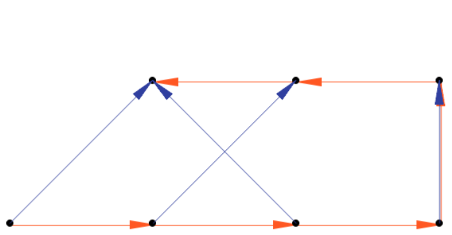
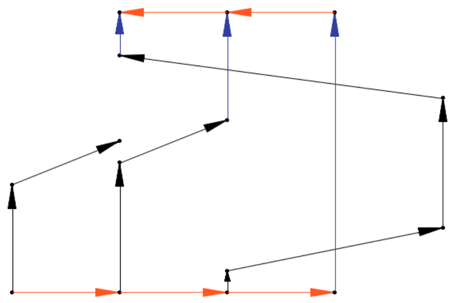
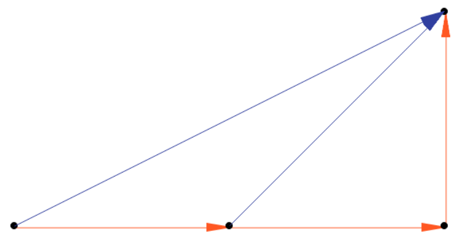
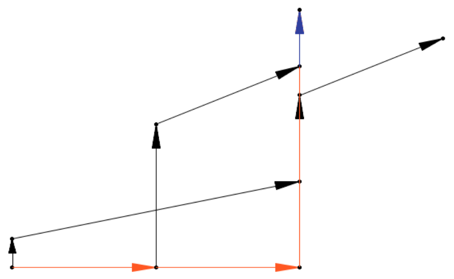
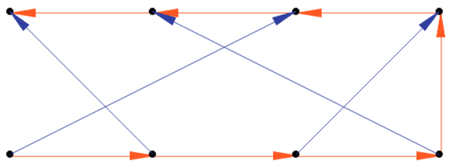
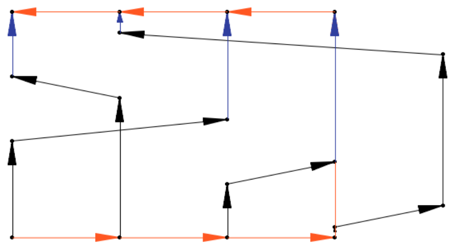
Grid aligned knit graph	Machine knitting graph
	
	
	
	

Table 6.2: Examples of machine knitting graphs. Note that, although transfers happen on two separate beds, the displayed graphs doesn't make a differentiation between which bed each node is transferring to or from. It may be assumed that every transfer alternates the beds that the node is occupying.

6.1.5 Knitting Code

In Section 6.1.3, we defined a language for a virtual knitting machine. The next thing we need to do is to convert the machine knitting graph into a series of instructions using this pseudocode.

The machine knitting graph has two properties we may exploit when converting it to knitting code.

1. Loop edges are wale and bed aligned.

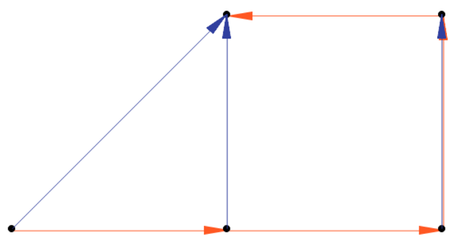
2. There is at most one loop edge from a given wale and bed on a course.

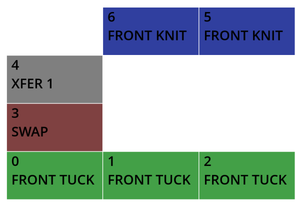
From these two properties, we may assume that every node position on the machine knitting graph corresponds to exactly one operation. Then, our algorithm works by processing each node in the graph in order, course by course, in alternating directions, starting with the yarn direction in the first course.

Given some yarn y and for each node $\bar{v}$,

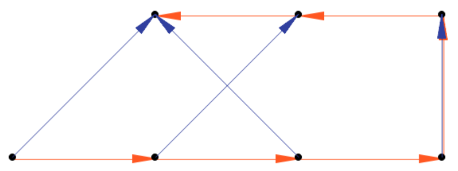
- If there is a yarn edge coming into $\bar{v}$,
  - and if an incoming loop from $\bar{u}$ to $\bar{v}$, then
    Knit$(\theta(\bar{v}), y)$ and
    add operation KNIT from needle position $\pi(\bar{u})$ to needle position $\pi(\bar{v})$.
  - but if there are no incoming loops, then
    Tuck$(\theta(\bar{v}), y)$ and
    add operation TUCK to needle position $\pi(\bar{v})$.
- Otherwise, there is a transfer edge coming into $\bar{v}$. Then
  Transfer$(\theta(\bar{u}), \theta(\bar{v}))$ and
  add operation TRANSFER from needle position $\pi(\bar{u})$ to needle position $\pi(\bar{v})$.

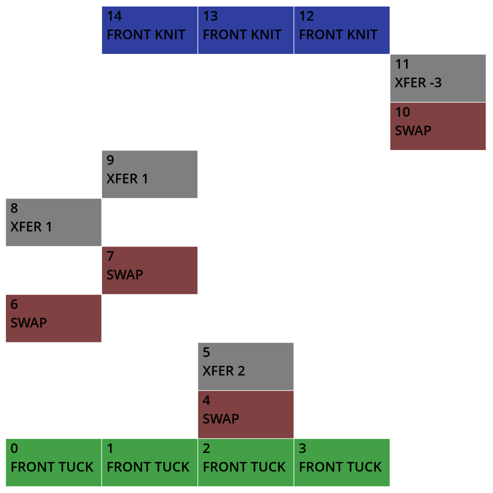
Grid aligned knit graph	**Knitting machine pseudocode**



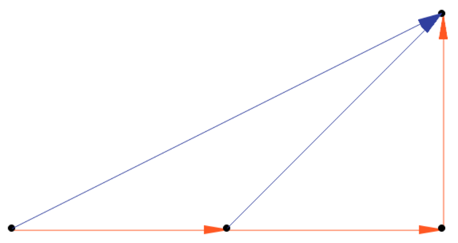


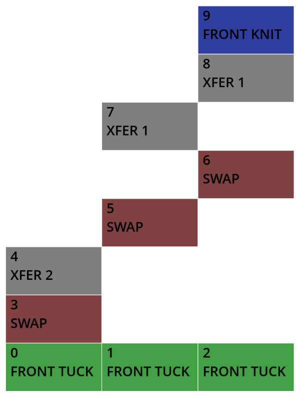
6 FRONT KNIT
5 FRONT KNIT
4 XFER 1
3 SWAP
0 FRONT TUCK
1 FRONT TUCK
2 FRONT TUCK





14 FRONT KNIT
13 FRONT KNIT
12 FRONT KNIT
11 XFER -3
10 SWAP
9 XFER 1
8 XFER 1
7 SWAP
6 SWAP
5 XFER 2
4 SWAP
0 FRONT TUCK
1 FRONT TUCK
2 FRONT TUCK
3 FRONT TUCK





9 FRONT KNIT
8 XFER 1
7 XFER 1
6 SWAP
5 SWAP
4 XFER 2
3 SWAP
0 FRONT TUCK
1 FRONT TUCK
2 FRONT TUCK

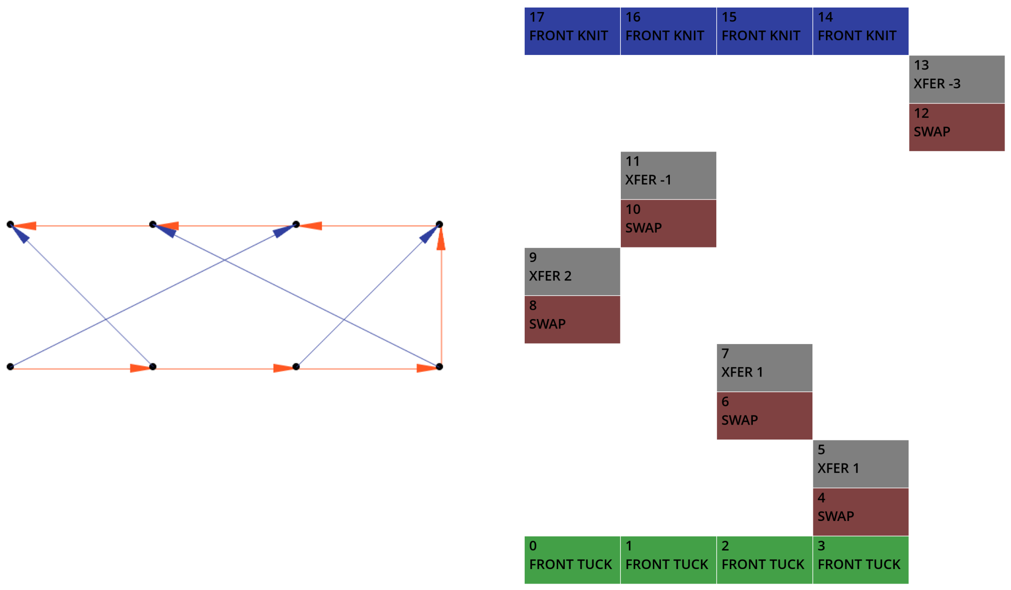


Table 6.3: Knitting machine graphs and their corresponding knitting machine pseudocode. Cells indicating SWAP correspond to a transfer from one bed to the opposite at the same wale position. Cells indicating XFER refer to a transfer operation to the opposite bed with some offset. Each grid cell is labeled with the operation name and a number to indicate their ordering.

Examples of knit graphs and their corresponding knitting machine pseudocode is shown in Table 6.3

6.1.6 Results

Some resulting physical knits, compared with their hand knitted counterparts, are shown in Table 6.4.

6.2 Discussion

The algorithm outlined in Section 6.1 is able to convert a limited subset of Knit Meshes into valid code for knitting machines. As stated, backwards facing loops are

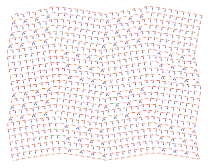
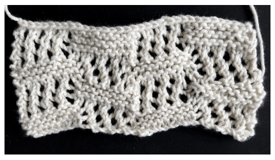
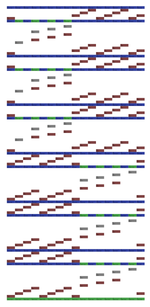
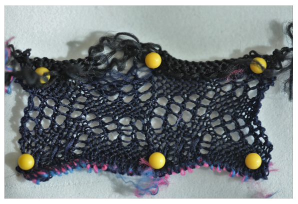
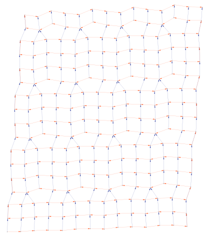
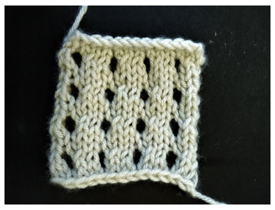
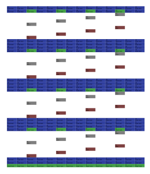
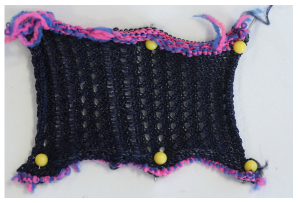
Knit graph	Hand knitted	Machine code	Machine knitted
			
			

Table 6.4: Results from computer generating machine code from a given knit graph. The machine knitted samples were knitted on the Matsuya M-100 knitting machine. The patterns for the sample were borrowed from the book, "Up, down, all-around stitch dictionary" [4].

not supported by flat weft knitting machines, and our algorithm currently does not support increases. For most patterns, one can work around these limitations simply by reversing backwards loops and using tucks in place of increases.

It's also worth noting that the transfer planning algorithm is more primitive than the one demonstrated by Narayanan, et al. [18]. We do not account for how much each yarn stretches, which is a significant constraint in knitting machines. There is more work to be done to get around these limitations for Knit Meshes.

A completely generalized algorithm for converting Knit Meshes to knitting code may be the holy grail for computational knitting. This would mean one could perform computations on knitted structures without needing to account for transfer planning, yarn carrier management, or other machine-specific functions. However, there may exist further limitations as to what kinds of knit graphs are theoretically knittable by V or X-bed machines. It remains to be shown as to what these limitations are, formally. If these limitations are formalized, one could define a constrained subspace of machine knittable Knit Meshes.

A conversion pipeline that does the inverse remains to be done. That is, an algorithm that can convert knitting machine code into Knit Meshes. We can follow the same process as in Section 5.2 using the virtual knitting machine constructed in Section 6.1.3. With this, one could then freely convert knitting machine code into human readable knitting instructions.

Chapter 7

Conclusion

In Chapter 3, we described a knitting abstraction that decouples knitting structure from machine or method while separating its topology from geometry. We showed how one can create a natural looking 2D embedding given just a knitting topology in Chapter 4, and then we designed two conversion pipelines, one for converting Knit Meshes to and from hand knitting instructions in Chapter 5 and one for converting to knitting machine code in Chapter 6.

While we strictly explored flat, weft-direction knitting, there is no reason why both circular and warp knitting cannot also be represented by Knit Meshes. Hand crocheting may also be represented with Knit Meshes. An interesting research direction would be into Knit Meshes as a universal description for all knitted structures.

In search of research opportunities in knitting, one should first look into computer graphics. Extensive research has been done in 3D polygonal models in Computer Graphics, which may have equivalent counterparts in Knit Meshes. For instance, it would be interesting to be able to re-tesselate a given Knit Mesh to a different structure while preserving some notion of its geometry, such as the silhouette.

Another path would be towards a computational knitting expression language. A knitting programming language would allow one to express complicated knitted structures, enabling new and unique computationally generated fashion designs.

Furthermore, there is a growing need for design tools in knitting. For instance, 3D printers have recently become accessible to the mainstream consumer market, despite their historically high cost. Knitting machines may see a similar push. Kniterate [21] is one recent example of this. With new consumer knitting hardware comes a need for accessible knitting design software, which will only be possible if we continue to improve the computational tools for knitting.

This is only a small subset of what can be pursued in computational knitting. Knit Meshes are central to these pursuits, as a generalized way of representing knitted structures.

Bibliography

[1] Sds-one apex3 | design systems | products | shima seiki, 2018. URL `http://www.shimaseiki.com/product/design/sdsone_apex/flat/`. Shima Seiki, Multi-Ply Fabric Cutting Machines | CAD/CAM Systems | Products | SHIMA SEIKI.

[2] Spjwin, 2018. URL `http://www.matsuya.com.cn/`. Matsuya.

[3] M1plus pattern software, 2018. URL `http://stoll.com/stoll_software_solutions_en_4/pattern_software_m1plus/3_2`. Stoll.

[4] Wendy Bernard. *Up, down, all-around stitch dictionary.* Stewart, Tabori & Chang, 2014.

[5] Pawel Gajer and Stephen G Kobourov. Grip: Graph drawing with intelligent placement. In *International Symposium on Graph Drawing*, pages 222–228. Springer, 2000.

[6] Emden R Gansner. Drawing graphs with graphviz. *Technical report, AT&T Bell Laboratories, Murray, Tech. Rep, Tech. Rep.*, 2009.

[7] Martin Hasal, Lukas Pospisil, and Jana Nowakova. Barzilai-borwein method in graph drawing algorithm based on kamada-kawai algorithm. In *AIP Conference Proceedings*, volume 1738, page 360005. AIP Publishing, 2016.

[8] Gaoming Jiang, Zhiwen Lu, Honglian Cong, Aijun Zhang, Zhijia Dong, and Dandan Jia. Flat knitting loop deformation simulation based on interlacing point model. *Autex Research Journal*, 17(4):361–369, 2017.

[9] Eric Jones, Travis Oliphant, and Pearu Peterson. {SciPy}: open source scientific tools for {Python}. 2014.

[10] Jonathan Kaldor. Simulating yarn-based cloth. 2011.

[11] Jonathan M Kaldor, Doug L James, and Steve Marschner. Simulating knitted cloth at the yarn level. In *ACM Transactions on Graphics (TOG)*, volume 27, page 65. ACM, 2008.

[12] Tomihisa Kamada, Satoru Kawai, et al. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989.

[13] Kenji Kotaki, Hidekazu Kitada, and Kiyoshi Minami. Knit design system and a method for designing knit fabrics, September 17 1996. US Patent 5,557,527.

[14] Joseph B Kruskal and Myron Wish. *Multidimensional scaling*, volume 11. Sage, 1978.

[15] Bin Luo, Richard C Wilson, and Edwin R Hancock. Spectral embedding of graphs. *Pattern recognition*, 36(10):2213–2230, 2003.

[16] 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, 2016.

[17] DL Munden. 26-the geometry and dimensional properties of plain-knit fabrics. *Journal of the Textile Institute Transactions*, 50(7):T448–T471, 1959.

[18] Vidya Narayanan, Lea Albaugh, Jessica Hodgins, Stelian Coros, and James McCann. Automated machine knitting of 3d meshes. *ACM Transactions on Graphics (TOG)*, 2018 forthcoming.

[19] Mariana Popescu, Matthias Rippmann, Tom Van Mele, and Philippe Block. Automated generation of knit patterns for non-developable surfaces. In *Humanizing Digital Reality*, pages 271–284. Springer, 2018.

[20] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500):2323–2326, 2000.

[21] Gerard Rubio, Saxena Triambak, and Tom Catling. Kniterate, 2018. URL `https://www.kniterate.com/`.

[22] Pete Shinners, Lenard Lindstrom, RenÃľ Dudfield, and Nicholas Dudfield. Pygame, 2018. URL `https://www.pygame.org`.

[23] David J. Spencer. *Knitting technology: a comprehensive handbook and practical guide.* Woodhead Publishing Limited, 3rd edition, 2001.

[24] Guido Van Rossum and Fred L Drake Jr. *Python reference manual.* Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[25] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[26] 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 (TOG)*, 31(4):37, 2012.