KnitGIST: A Programming Synthesis Toolkit for Generating Functional Machine-Knitting Textures

Megan Hofmann
Human Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA, USA

Jennifer Mankoff
Paul. G. Allen School of Computer Science
University of Washington
Seattle, WA, USA

Scott E. Hudson
Human Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA, USA

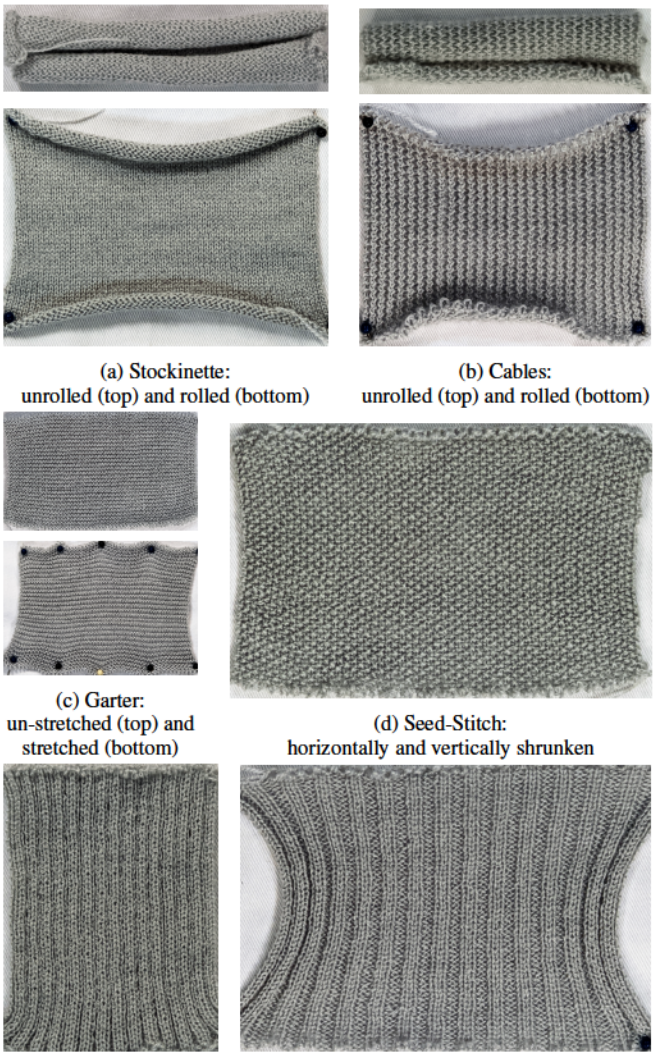
ABSTRACT
Automatic knitting machines are robust, digital fabrication devices that enable rapid and reliable production of attractive, functional objects by combining stitches to produce unique physical properties. However, no existing design tools support optimization for desirable physical and aesthetic knitted properties. We present KnitGIST (Generative Instantiation Synthesis Toolkit for knitting), a program synthesis pipeline and library for generating hand- and machine-knitting patterns by intuitively mapping objectives to tactics for texture design. KnitGIST generates a machine-knittable program in a domain-specific programming language.

Author Keywords
knitting; program synthesis; generative design

INTRODUCTION
Machine knitting is a powerful fabrication medium for crafting complex interactive objects, but knitting design systems are a relatively new research topic. The complex physical and visual properties of knit *texture* come from the combinations of numerous *stitches*. There are numerous properties of knit textures that must be maintained to create a knit object that will not unravel. Maintaining these properties while adding requirements to generate specific textures constrains the problem further. Such design work currently requires extensive domain expertise. If we can encapsulate that expertise in a generative design tool, a wider variety of people could produce knit textures more easily. Rather than tediously specifying textures stitch by stitch, a design tool should enable designers to specify high-level objectives. However, generative design of knit textures poses two key challenges: (1) maintenance of the hard constraints that ensure a knit object will not unravel [13, 28]; and (2) generation of user-defined physical and aesthetic objectives.

UIST '20, October 20–23, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7514-6/20/10…$15.00
https://doi.org/10.1145/3379337.3415590



(a) Stockinette: unrolled (top) and rolled (bottom)
(b) Cables: unrolled (top) and rolled (bottom)
(c) Garter: un-stretched (top) and stretched (bottom)
(d) Seed-Stitch: horizontally and vertically shrunken
(e) Knit 2 Purl 2 Rib: un-stretched (left) and stretched (right)

Figure 1: Exemplar textures produced during the described scenario that are curly (a), cabled (b), vertically-elastic (c), shrunken (d), and horizontally-elastic (e).

Generative Instance Synthesis Toolkit for knitting (KnitGIST) is a programming framework for generative design through program synthesis. The framework synthesizes knit textures by iteratively modifying a domain specific language for programming knitted textures, KnitSpeak [13], based on pairings of programmer- defined objectives and tactics. Objectives measure how well the generated texture meets the programmer's requirements. Tactics change a KnitSpeak program so that the resulting texture is more likely to meet those objectives. In each iteration a texture and tactic are selected, and the tactic is applied to produce a new texture that is likely closer to the goal. The new texture is added to the population of best textures if it compiles and demonstrates an improvement over other textures. This method rapidly explores the space of possible knit textures represented by KnitSpeak programs. The textures that get generated can be repeated to create scalable sheets of knit fabric with the desired properties.

Using the KnitGIST framework, we can easily construct multiple knit texture optimizers by changing the objectives and tactics used to modify/synthesize knit textures represented using KnitSpeak. Objectives evaluate aesthetic and physical properties of a texture, and tactics modify KnitSpeak to produce new textures. By combining these, we can produce textures with a variety of physical and aesthetic properties. To support this, the framework provides an library for optimizer creation. Using KnitGIST and this library we demonstrate how to build optimizers that support: (1) instantiation of functional textures, (2) swapping of textures for aesthetic purposes, and (3) generation of a functional and attractive lacy lampshade.

This framework lays the groundwork for creating generative design tools that reliably produce functional knit textures. KnitGIST separates programmer concerns in two ways. First it separates hard-constraints (what must be true to make a texture knittable) from soft-constraints (the properties of a specific texture the programmer wants) into a verifiable programming language and compiler, and into objectives. Second, by factoring out objectives and tactics, programmers can explicitly specify heuristics to indicate simple steps through the search space that the programmer believes will lead to an improved outcome. This separation of concerns lays the groundwork for pluggable user-interfaces where objectives and tactics can be mixed and matched to reliably produce a wide variety of compiled, functional, knittable results. Plugging in different combinations of objectives and tactics enables versatility to blend simple criteria to achieve complex effects.

Scenario: Property Driven Stockinette

A programmer can create a curly, symmetrical texture using KnitGIST without detailed knitting knowledge. Seeded with a mapping between objectives and tactics, but no domain specific knowledge, the programmer can construct a generative design program to create this curly texture. First, they map an objective that the texture curls up on itself to a tactic that modifies random stitches to curl forward. Then, they map two objectives that the texture is vertically and horizontally symmetrical to two tactics, respectively, that mirror stitches across the axes of symmetry. An experienced knitter would identify *stockinette* (a texture of all forward-curling knit stitches) as a simple and common texture that meets these criteria. This KnitGIST optimizer consistently produces stockinette texture. When the programmer changes the objectives to target a variety of physical and aesthetic properties, the new optimizers reliably produce other canonical textures (Fig 1).

RELATED WORK

Algorithmic Knit-Design

The majority of computer-aided knitting tools fit into two limited categories. Machine knitting is largely done with (1) stitch-level machine knit charting tools and (2) higher level 'wizard' tools that limit designs [33, 37, 35] to a narrow set of templates (i.e.hat, scarf, socks) which can be modified in limited regions (i.e.cuffs) with a small set of pre-defined textures. Hand-knitters rely on sourcing designs from books and repositories [20, 31, 7], or designing textures with adopted tools like spreadsheets [26]. These improvised methods do not support any verification of the final knit object.

To address this, recent work has focused on algorithmic, verifiable, computational solutions for generating knitting patterns. McCann et al.created a machine-knitting compiler which included a simple, machine-level language for controlling knitting machines [25]. The simplified machine language, Knitout [14], is an instruction set for assigning loops of yarn to be held on beds of hook-shaped needles[1]. Using this instruction set, Transfer planning is the process of assigning needle locations to loops in a graph structure such that the represented knit object can be knitted on a knitting machine [23].

Building on this architecture, knit surfaces can be created from 3D models [30, 28, 29]. Narayanan et al.added support for applying textures which deform the 3D surface and add new physical properties [29]. Karmon et al.developed a tool to simulate such deformations [16], but designing for those deformations remains a challenge. Alternatively, textures and knit objects can be composed in higher level domain specific languages and compiled to knitting machine instructions. Two approaches to machine knitting have focused on developing domain specific languages for texture and pattern design. Kaspar et al.developed a language that isolates regions of stitches in a garment and applies patterning operations in layers [17]. KnitSpeak uses existing hand-knitting nomenclature which is straightforward to write and interpret as hand-knitting instructions or compile down to machine-knitting instructions [13].

Overall, algorithmic machine knitting has formed a cohesive architecture and work flow: (1) model knit object by 3d modeling or texture programming; (2) convert (e.g., compile) those models to KnitGraphs to be verified, manipulated, and evaluated, and (3) convert the KnitGraph to knitting machine instructions by transfer planning. While there has been more substantial work on generating KnitGraphs from 3D modeled

[1] Additional knitting machine details, see [25, 1]

shapes [28, 29], little work has focused on generating functional texture programs based on user specifications and assurance of knittability.

Generative Design in Fabrication

Instead of relying on designers' skills, generative design treats design as an optimized-search problem. The designer defines a search space and evaluation criteria and an optimizer can search for a design that maximizes desired properties. In most commercial tools the designer has a limited ability to manipulate the evaluation function [4, 3, 12].

Research into generative design and fabrication is extensive. The primary focus of this work has been to develop optimizations over physical properties (e.g., structural integrity [39], strength [9, 41], material usage [9], shape [41, 24], aesthetics [24], and actuation [22, 6, 34]). Generally, these design tools enable designers to intuitively bound the search to: a 3D model's shape [24], sketches [9, 18], or video inputs [22].

Some work has been done to translate these approaches into domain-specific tools. For example: Li et al.use generative design to generate actuation mechanisms for controlling non-digital objects [22]. Anderson et al.use generative design to construct circuits based on easy to program trigger-action structures [2]. In the domain of soft fabrication, Bern et al.use this class of algorithms to generate "animated plushies" [6] which can be fabricated to move like a provided animation. Narayanan et al.use these methods to translate 3D models into machine knitting instructions [28, 29].

Despite such innovations, there is no uniform architecture for generative design [8]. Krish notes this lack of a unifying method and breaks down broad classes of generative design (e.g., genetic design [5], shape-grammars [19], hill-climbing methods) into three components: (1) the design schema or representation, (2) a means of creating variations, and (3) a means of selecting desirable outcomes [21]. Each of these components can be challenging to implement and requires a significant amount of domain expertise. However, a formal combination of these components could lead to easier implementation. A simplified optimization framework can be effective for generating user-interface layouts [11] if it trades off algorithmic specificity for ease of implementation.

Program Synthesis in Fabrication

Program synthesis algorithms generate a *program* that meets specified requirements. It is generally broken into three methods: oracle guided synthesis [15], stochastic super-optimization [32], and enumerative search [38]. Schufza et al's[32] stochastic super-optimization approach starts with a complete program and randomly mutates the design. Each mutation is selected based on a cost-evaluation of the current program. The search will generally lead to mutations that make the current program more efficient, enabling rapid random exploration of a large design space. In this method, there is a critical verification step that checks that the new candidate program produces the same output as the original program. Implicit in this step, is verification that the program is also a valid program. In the space of fabrication, shape grammars arguably fit into the field of program synthesis. In this case, the grammar defines a set of shapes and ways of modifying these shapes [19]. Generative design systems that use these representation are essentially searching the space of candidate programs which generate the desired shape.

There has been a recent focus on using these methods for fabrication. Nandi et al.use program synthesis to parameterize 3D meshes, effectively bridging the gap from difficult to edit but easy to share model formats to an easy to edit format [27]. Wu et al.use similar compilation methods to generate low-level carpentry instructions from manufacturing requirements [40]. Dumas et al.use synthesis methods to generate 3D printed textures from user provided examples [10]. However, these techniques have not be applied to machine-knitting.

GIST: GENERATIVE INSTANCE SYNTHESIS TOOLKIT

KnitGIST is a framework that enables programmers to create knit texture optimizers that consist of five components: (1) a KnitSpeak compiler, (2) a *population* of promising textures, (3) a KnitSpeak synthesizer; (4) *objectives* which measure how well a texture meets a set of programmer-specified requirements, and (5) *tactics* that produce a modified instance of the KnitSpeak texture. When constructing an optimizer, the programmer associates the objectives and tactics by the expectation that a tactic will modify a KnitSpeak program to improve a given objective. To do this, the programmer only needs to understand what an objective evaluates and what a tactic does, not how either were implemented.The programmer assigns weights to objectives to express their importance, and to tactics to express the likelihood that it will improve an associated objective. These weights are used to compute *expectation improvement scores* that guide the selection of tactics to, likely, improve high-value objective scores.

With KnitGIST we can easily construct optimizers for knit textures. By breaking up and organizing these components, programmers are able to explicitly encode the relationships between soft-constraints (represented by objectives), and steps through the search space (tactics). The hard-constraints of knittability are managed bu the KnitSpeak-compiler. Using program synthesis separates the concerns of user goals (what is being optimized for) from hard constraints of knitting. This framework is similar to Schufza et al'sapproach to super-optimization [32].

Given these components, KnitGIST executes as follows (Fig 2). (1) The synthesizer produces the initial population of textures. (2) Each texture is compiled; if a member does not compile it is thrown out since it will not knit. (3) Each texture is scored with an aggregate objective function which is the weighted sum of the individual objective scores. (4) A member of the population is randomly selected to be modified; high-scoring textures are more likely to be selected. (5) A tactic is selected to modify the selected texture. Tactics with a higher expectation of changing the texture to meet a highly weighted objective are more likely to be selected. (5) The tactic produces a new, modified texture which is (6) either added to the population or filtered out if it does not compile or is not better than at least one member of the population. Only the N top scoring textures are kept to cap the population size.

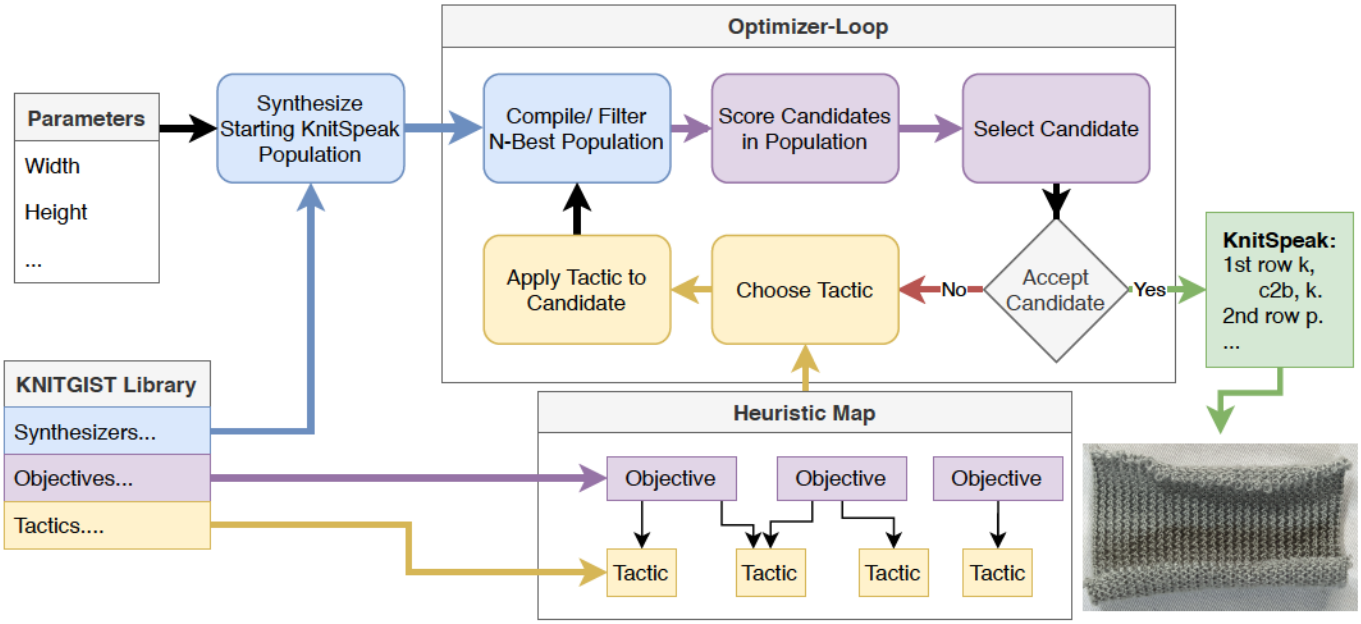


Figure 2: A programmer provides parameters and selects a synthesizer (blue). They construct a weighted heuristic map between objectives (purple) and tactics (yellow). This gets plugged into an optimizer which generates optimized KnitSpeak (green).

Steps 3 through 6 are repeated until a program scores high enough over the objective-function, or a maximum number of iterations have been exceeded. Using this process we can structure a wide range of discrete-search optimizers that represent hill-climbing algorithms and genetic algorithms.

Heuristic-Map: Deciding How to change a Program

The space of knit textures may not be convex, so strict hill-climbing is unlikely to find a globally optimal solution. Instead, we consider a collection of promising points across the search space. Selection on these starting points is based on an expectation of improvement over the current population. When a tactic is applied to a texture, it modifies the texture. Since our goal is to move from poor performing textures to high performing textures, we generally want to select tactics that we expect will improve the score of the objective-function. However, if we strictly follow the best tactic, we are likely to find local minima and not explore the whole search space. Instead, we randomly select tactics, with tactics we expect to improve the score being more likely to be selected. This *expectation improvement score* (EIS) (Eq. 1c) is calculated based on a mapping of objectives to tactics—called the heuristic map. To create a heuristic map, the programmer assigns: priority weights to each objective, tactics to objectives, and weights to tactics based on how likely the programmer thinks the tactic will improve the respective objective's score.

To calculate the EIS for a tactic, we will we first need to calculate the value of that tactic for each possible objective. The value, $V(t,o)$, of a tactic is estimated as the weight on the mapping between the tactic, t, and the weighted objective, o: $\alpha_{o\to t}$, normalized by the sum of all weighted mappings between o and all other tactics (Eq. 1a). The importance of an objective, $I(o,O)$, depends on current score of the objective, $s(o)$. $I(o,O)$ is calculated as the weight of the objective multiplied by the distance from the maximum score to the current score, normalized by the weighted sum of all score distances (Eq. 1b). Given the value of a tactic and the importance of each objective, we calculate $EIS(o,O)$ as product of the tactic value and the objective importance for all objectives (Eq.1c).

$$V(t,o) = \frac{\alpha_{o\to t}}{\sum_{t'\in T_o} \alpha_{o\to t'}} \tag{1a}$$

$$I(o,O) = \frac{\beta_o(1-s(o))}{\sum_{o'\in O} \beta_{o'}(1-s(o'))} \tag{1b}$$

$$EIS(t,O) = \sum_{o\in O} V(t,o)I(o,O) \tag{1c}$$

MACHINE KNITTING BACKGROUND

To create effective objectives and tactics, we first introduce common knitting concepts. A knitted fabric starts with a row of loops and then additional rows of loops are created by pulling loops through the top-most row to create a sheet of fabric. A single loop is not stable; pull on its ends and the loop falls apart. Knitted fabric gains its stability from the relationship between loops: when pulling a child loop through a parent loop; the parent loop becomes stable. A loop pulled through another loop is called a *stitch*; adjacent stitches are called *rows*; and columns of stacked stitches are called a *wale*.

As shown in Fig 3, a loop can be pulled through another loop from the back-to-the-front (*knit*) or from the front-to-the-back (*purl*). A loop can be pulled through more than one loop ,to decrease the number loops on the row, or added without being pulled through another loop (*yarn-over*), to increase the

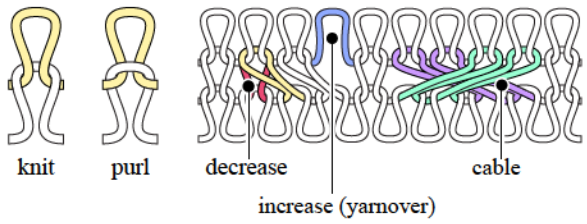


Figure 3: Diagrams of the most common stitches.

Table 1: Summary of common knitting nomenclature.

Term	Definition
Stitch	A loop with zero or more loops pulled through it. May colloquially also include cables.
Row	A row of stitched loops making up a horizontal section of fabric
Wale	A series of loops in a column that are pulled through each other in a vertical chain.

number loops on the row. As loops are pulled through other loops they can cross over each (*cable*). These combinations of different stitches determine the texture and shape of a knitted fabric. A summary of these terms is presented in Table 1.

A Computational Model of Knitting

Given the structure of knitting, we represent a knitted object as a graph where nodes represent loops, and edges represent loops pulled through other loops. (e.g., [28, 29, 25, 13]). There are a couple of variations on graph structures that have been used in the literature; in this paper our graphs match the KnitGraph structure described in [13] (Fig 4). As such we use the same notation whenever possible, reviewed in Table 2.

A KnitGraph consists a set of *loops* with directed *stitch edges* representing where a loop is pulled through a parent loop. A loop may have multiple or no *parent* loops but can only have one child loop. Each stitch edge has an *orientation*, denoting if the loop was pulled back-to-front (knit) or front-to-back (purl). Stitch edges may cross over one another in a cable stitch. Each KnitGraph is segmented into rows of loops where each row builds on the row below it by pulling its loops through the loops on the row below. We derive KnitGraphs of knit textures by compiling KnitSpeak, a programming language closely patterned after a widely used notation for hand knitting instructions [36, 7]. KnitGIST uses a KnitSpeak compiler [13] which ensures that all textures compiled in KnitSpeak will be machine knittable and will not unravel.

For machine knitted samples, the KnitSpeak compiler then converts the resulting KnitGraphs into automatic knitting machine instructions (i.e.Knitout [14]). All of the presented machine knit samples were knitted on an Shima Seiki SWG91N2 15-gauge v-bed knitting machine using Tamm Petit, a 2/30NM (8,147 yards per pound) acrylic yarn with moderate twist. We used our machine's digital stitch control system to regulate yarn tension and our stitch size was 40 with leading set 25. This is the same machine, yarn, and settings

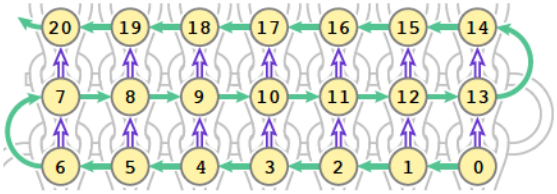


Figure 4: A KnitGraph is a link graph representation of a knitted fabric. Loops (yellow-circles) are connected by yarn edges (green) and stitch-edges (purple).

Table 2: Summary of KnitGraph notation.

Notation	Interpretation
$l \uparrow m$	The stitch edge where loop m is pulled through loop l
$o(l \uparrow m)$	The orientation of the stitch edge $l \uparrow m$
$P(l)$	Set of parent loops l is pulled through
R_i	The row at index i
N_R	Top row-index in a texture
$i(l)$	The index of l in its row

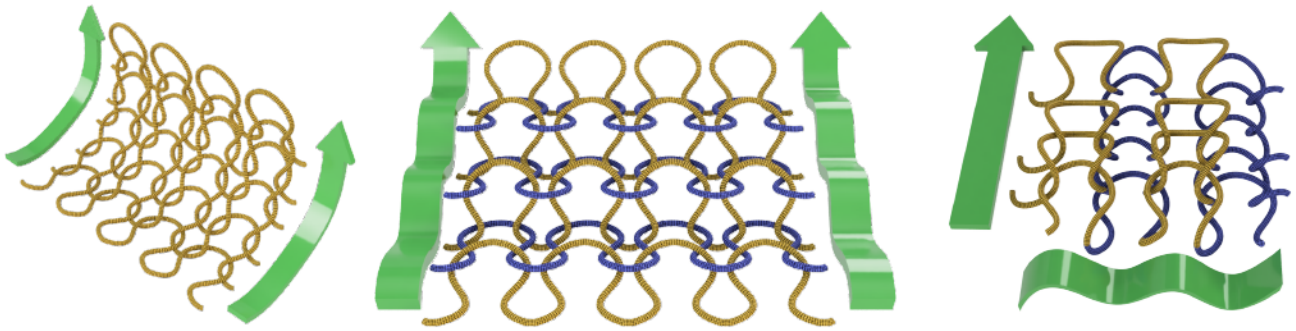
used to create the KnitPick Database [13]. Swatches took between 2 and 30 minutes to knit, depending on the types of stitches and the size of the swatch. Cable and lace patterns took significantly longer because of complex transfers on all rows, while knit purl patterns knit quickly. Hofmann hand knitted samples using 4 worsted weight 100% acrylic yarn, a common hand-knitting yarn, on stainless steel 5mm diameter needles.

KNITGIST LIBRARY

We provide a library of pluggable functions to support texture generation. we provide a KnitSpeak synthesizer which generates knit textures of a specified size. Next, we provide a library of seven parameterizable objectives and five tactics that support the generation of optimized KnitSpeak-textures (Table 3). While this library is far from complete, it still provides a wide range of possibilities, and illustrates the versatility of the extensible KnitGIST framework. We note also, that while this paper considers only a programming interface, the "mix and match" nature of our pluggable framework points to strategies for an end-user interface based on picking constructs and parameters that are understandable in the application domain.

KnitGIST Synthesizer

A synthesizer produces a KnitSpeak texture. In the KnitGIST framework, we use a synthesizer to produce a starting population of textures of a specified width (stitch-count) and height (row count). The synthesizer constructs a Markov-model of stitch relationships from a dataset of KnitSpeak programs [13]. Each stitch type (e.g., knit, purl, increase, decreases, cable) is a state and the probability of being followed by a stitch of another type (moving to the next state) is based on



(a) All-Knits (b) Rows of Knits then Purls (c) Wales of Knits then Purl

Figure 5: Knits (yellow) tend to pull the fabric forward, while purls (blue) tend to pull the fabric back. A fabric with many connected knits will tend to curl (a). Switching between knits and purls will contract that curl causing shrinkage (b and c).

the count of those neighboring stitches in the dataset. A random first stitch is selected and then stitches are added based on the current stitch. Stitches are added to fill rows and rows are added until the target height is met. The advantage of using KnitSpeak is the ease of verifying knittability by compiling the texture.

Library of Objectives for Textures

Objectives define the properties of the knit texture being optimized for. The KnitGIST library includes four physical and three aesthetic objectives (Table 3). The objectives were driven by prominent categories laid out by knitters [36]. An objective scores a texture by comparing a value of interest, v, to a programmer-defined target, t, over some bounds, $[min, max]$. The score is one if the calculated value is equal to the target, and drops off to zero linearly as the value approaches the bounds (Eq. 2). Each objective calculates v differently.

$$S(t, min, max, v) = \begin{cases} 0 & v \notin [min, max] \\ 1 - \frac{t-v}{t-min} & v \leq t \\ 1 - \frac{v-t}{max-t} & t < v \end{cases} \quad (2)$$

The physical objectives presented describe how the resulting texture will modify a physical characteristic of textures: curl, shrinkage, elasticity, and opacity. How much a knit object curls, shrinks, stretches, or blocks light is ultimately dependent on a wide range of manufacturing variables (e.g., machine, machine settings, fiber, gauge). However, texture plays a critical role in modifying the basic properties. For instance, an inelastic fiber can still result in a stretchy fabric if it is knitted into a horizontally elastic rib texture. Alternatively, a thin, lace-weight, yarn can result in airy texture if texture-opacity is reduced. The following physical objectives produce a value, v, between -1 and 1, where zero implies that a texture contributes nothing the the specific property of a knit object. One or negative one implies that the texture maximizes this property of a given knit object. The negative scale implies direction in the cases of curl (curl forward or backwards) and elasticity where a texture may resist stretch. How these modifiers effect specific knitted fiber materials will vary significantly depending on fiber properties.

Estimating Curl for the Curl Objective

Depending on its construction, a knit texture may curl quite significantly, as shown in Figure 1a. Given an estimator for curl, which is mapped to the value, v, for the objective, this objective can help to ensure that a knit texture lies flat or curls up, a property which can be used to enable interesting structural or aesthetic goals.

To estimate the curl of a texture, v_c, it is important to understand why knit textures can curl. Curl is an effect that occurs only in the vertical direction, due to the orientation of stitch edges between child and parent loops. This effect is cumulative. If the curl of stitches in a wale are all in the same direction, the last stitch in the wale is pulled forward causing the texture to curl. When stitches do not curl in the same direction they cancel each other out locally. The effect of the curl of each successive stitch decreases with distance.

We model this recursively (Eq. 3a). If a loop, l, has no child, its curl, $c(l)$, is the average orientation across its parent loops. Otherwise, its curl is that average plus the cumulative curl of its child loop, m, divided by a decay-factor, σ. Empirically, we find 2 to be a good value for σ. To estimate v_c we model how $c(l)$ adds up over the whole fabric. We estimate the cumulative curl of a knit texture v_c as the average curl of loops in the bottom row, R_0, which accumulates the curl of all subsequent rows (Eq. 3b).

$$c(l) = \frac{1}{|P(l)|} \sum_{p \in P(l)} o(p,l) + \begin{cases} 0 & \nexists m | l \uparrow m \\ \frac{1}{\sigma} c(m) & \exists m | l \uparrow m \end{cases} \quad (3a)$$

$$v_c = \frac{1}{|R_0|} \sum_{l \in R_0} c(l) \quad (3b)$$

If a fabric is made up of only knits or only purls it will curl (Fig 5a). However, alternating patterns of knits and purls will prevent the fabric from curling. Curl may cancel out; for example, the forward curl of a row of knits is canceled out by

the backwards curl of a row of purls (Fig 5b). Also, the forward curl of a wale of knits is canceled out by the backwards curl of a wale of purls (Fig 5c). Examples of these effects are demonstrated in the scenario (see Fig 1).

Estimating Shrinkage

When a texture is constructed with alternating orientations of stitch edges, it can shrink significantly as illustrated in Figure 1.e. Given an estimator for shrinkage, this objective can ensure that a texture is compact. Shrunken textures are frequently used at the boundaries of other textures or edges of garments to prevent curl. We estimate texture shrinkage based on the counteracting orientations of knits and purls, which causes stitches to overlap. A second cause of shrinkage occurs when loops are tightened by being pulled through a far away loop; as in cables where loops are crossed over other loops, or in decreases where loops are gathered through a child loop.

Shrinkage is caused by the relationship between neighboring loops in either a vertical (wale-wise) or horizontal (row-wise) direction. We denote the wale wise neighbor of a loop, l, in a given direction, d, which is either vertical, $\updownarrow$, or horizontal, $\leftrightarrow$, as $neighbor(l,d)$ (Eq. 4). The loop n is a wale-wise neighbor to l if and only if n is pulled through l; that is, there is a stitch edge from l to n. Loops l and n are row-wise neighbors if and only if the row-wise index of n is one more than the row-wise index of l and n and l are in the same row.

$$neighbor(l,d) \iff \begin{cases} \exists l \uparrow n & d = \updownarrow \\ i(n) = i(l)+1; n \in R(l) & d = \leftrightarrow \end{cases} \quad (4)$$

With respect to orientation, if two neighboring stitch edges have opposite orientations, they will tend to overlap each other. In other words, when a purl has knits on either side, only the knits are visible, because the knits on each side of the purl overlap it completely. This effect is visible in Figure 5c. This is true whether the stitches are aligned vertically (along a wale, in a parent-child relationship) or horizontally (along a row). Since overlap depends on orientation (knit or purl), we must observe two stitch edges (between two loops each)

Table 3: KnitGIST Library broken into Objectives and Tactics.

Component	Function	Parameters
Objectives	Curl	NA
	Shrinkage	Direction: Horizontal or Vertical
	Elasticity	Direction: Horizontal or Vertical
	Opacity	NA
	Symmetry	Axis Location
	Style	Style-Type: Knit-Purl, Cable, Lace
	Imagery	Region-Map, objectives, objective-Weights
Tactics	Flip Stitch	NA
	Lean Stitch	NA
	Replace Stitch	New Stitch Type
	Mirror Stitch	Axis Location
	Swap Stitch	Alternate KnitSpeak-Texture

where the child loops are either wale-wise or row-wise neighbors. Given loop, l, and its wale-wise or row-wise neighbor, n, we calculate the overlap, s_o, between two loops (l, n) as the difference between the average orientation of the stitch edges between the parents of l ($p \in P(l)$) and the parents of n ($m \in P(n)$) where $o(p \uparrow l)$ denotes the orientation that l is pulled through p. We use the term d to denote the direction (either horizontal or vertical) that determines which neighboring loop, n, is selected. So given, loop l with the set of parent loops $P(l)$ and a neighboring loop n with the set of parent loops $P(n)$, the loops overlap by:

$$s_o(l,d) = \frac{1}{|P(n)|}\sum_{m \in P(n)} o(m \uparrow n) - \frac{1}{|P(l)|}\sum_{p \in P(l)} o(p \uparrow l) \quad (5)$$

Loops overlap by the average of their *width*. The width of a loop depends on the location of its parent loop. When loops are pulled through a parent loop directly below them they have a standard width. As the distance between the parent and child loops is increased (e.g., cable, decrease), the child loop is stretched vertically making it thinner. As the distance between the loop, l, and its parent p increases, l is stretched thin. This distance is the difference between the in-row index of a loop, $i(l)$ and of its parent loop, $i(p)$. The width of the child loop, $w(l)$, is calculated as a factor of this sum of the distances between the loop l and all of its parents, $p \in P(l)$, plus the distance between the rows (i.e.1) (Eq. 6). The width of a loop, l, pulled through parents, $P(l)$, is:

$$w(l) = \frac{1}{1 + \sum_{p \in P(l)} |i(l) - i(p)|} \quad (6)$$

We can calculate the shrinkage between two loops as the average of their widths (Eq. 6) multiplied by the amount they overlap (Eq. 5). So for a loop l with neighboring loop n:

$$s(l,d) = \frac{(w(l)+w(n))}{2} s_o(l,d) \quad (7)$$

Shrinkage of a texture is the average shrinkage across all loops. A programmer-set parameter d, dictates whether to calculate shrinkage either vertically or horizontally by using the appropriate definitions of a neighbor (Eq. 4). So over the whole set of rows of size N_R, we calculate the average shrinkage over all loops, l, in all rows, R_i.

$$v_s(d) = \frac{1}{N_R}\sum_{i=0}^{N_R}\left(\frac{1}{|R_i|}\sum_{l \in R_i} s(l,d)\right) \quad (8)$$

Consider a garter texture, made up of alternating rows of knits and purls (see Fig 5b). The rows will have opposing curl, so the stitches will overlap and the texture will shrink vertically. Alternatively, consider ribbing, made up of alternating wales of knits and purls (see Fig 5c) which shrinks horizontally.

Estimating Elasticity

Textures tend to be either vertically or horizontally elastic, as stretching in one direction shrinks the fabric in the other direction. Texture that do not shrink resist stretching. Similarly, textures that shrink in both directions, resist stretching because stretching the texture one way will cause stitches in the other direction to overlap. Consider again alternating rows of knits and purls (see Fig 5b) and alternating wales (see Fig 5c). The alternating rows shrink vertically but not horizontally so the texture stretches vertically. Alternating wales do the opposite, so the texture stretches horizontally. Elasticity in one direction is the difference between the shrinkage in the target direction and the shrinkage in the opposite direction (Eq. 9) .

$$v_e(d) = \begin{cases} max(v_s(\leftrightarrow) - v_s(\updownarrow), 0) & d = \leftrightarrow \\ max(v_s(\updownarrow) - v_s(\leftrightarrow), 0) & d = \updownarrow \end{cases} \quad (9)$$

Opacity

Opacity is largely defined by pairs of increases and decreases, leaving a gap between them. Yarn-overs are an increase where an extra loop, with no parent loops, adds one loop to a row. When this is combined with a decrease in the same row, it leaves an obvious hole or eyelet in the fabric. We can estimate the opacity of the texture as the density of opaque loops (loops with a parent loop), and non-opaque loops (loops without a parent loop) (Eq. 10). Intuitively, textures with many increase-decrease pairs (i.e.lace) will be less opaque than other textures.

$$v_o = \frac{1}{N}\sum_{i=0}^{N}\left(\frac{1}{|R_i|}\sum_{l\in R_i}\begin{cases} 0 & |P(l)| > 0 \\ 1 & |P(l)| = 0 \end{cases}\right) \quad (10)$$

Symmetry

Symmetry creates aesthetically-balanced texture. We evaluate symmetry across the stitch edges equidistant across an axis in a texture. The programmer chooses the axis location, defaulting to the center. Symmetry is a concept that can be specified by the programmer. Generally, symmetry is binary, but there are many properties that may be symmetrical. By default, we compare three properties of the stitch edge: orientation, depth, and lean. Given a set of symmetry functions, S, which return a value between 0 and 1, comparing two stitch edges, our framework calculates overall symmetry for a pair of edges by averaging the values returned by those functions. Over an entire texture, the symmetry value is the average symmetry value between paired equidistant-stitches across a symmetry axis. Given a horizontal or vertical axis, the set A contains all pairs of edges equidistant from that axis. So a pair of edges $p \uparrow l$ and $p' \uparrow l'$ are in A if l and l' are equidistant from the axis and p is a parent of l and p' is a parent of l'.

$$v_{sym}(A) = \frac{1}{|A|}\sum_{(p,l,p',l')\in A}\left(\frac{1}{|S|}\sum_{sym\in S} sym(p\uparrow l, p'\uparrow l')\right) \quad (11)$$

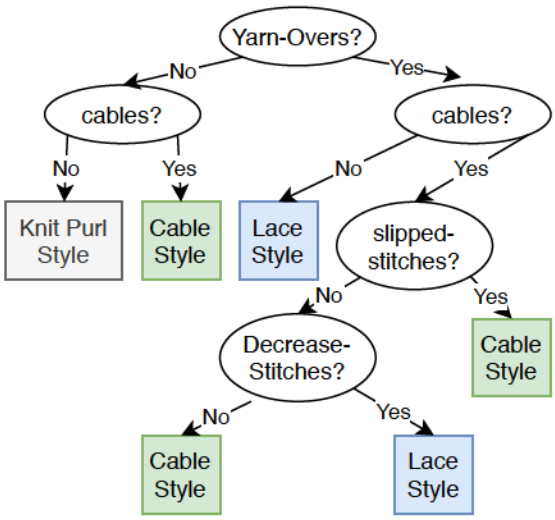


Figure 6: Style classification decision-tree model.

Styles

Styles are the broad categories of knit textures used by most hand-knitters (i.e.knit-purl, cables, lace). Each of these styles have unique physical and aesthetic properties. Knit-purl patterns are the simplest, consisting of only knits and purls. Cable patterns use cable stitches to cross stitches over one another giving the appearance of "traveling-stitches". Lace patterns balance increases and decreases to create eyelets.

We estimate style with a decision tree based on the types of stitches in a texture. We manually labeled 1548 KnitSpeak samples from the Stitch-Maps repository [7] as: ***Knit-Purl, Cable*** or ***Lace*** by examining swatch-photos. We calculated a variety of features from the KnitSpeak text including: stitch-counts, patterns of repeated combinations of stitches, and presence or absence of certain stitches in the texture. We trained a decision tree to classify textures using 25% of the samples in a development set, 50% in a cross-validation set, and the remaining 25% hold-out for final testing. The model produced by the C4.5 decision tree algorithm (confidence interval .15, Minimum of five instances per leaf) had an accuracy of 95.6% ($\kappa = 0.86$) over the withheld set. The best model used features specifying which stitch types were present (Fig 6).

Imagery

Imagery is a way of creating visual effects such as a hexagon using texture. Imagery-objectives allow programmers to apply other objectives over specific 2D regions of the textures. Programmers specify a region-map where a set of loops are mapped to another objective (i.e.curl, shrinkage, elasticity, opacity, symmetry, style). Like all the previous objectives, the programmer provides a target for the region's objective and acceptable bounds. The value of the objective is calculated over the regional subset of loops, rather than the whole texture.

Since it would be tedious to assign loops to a region-map by hand, we provide a painting tool to assign objectives to re-

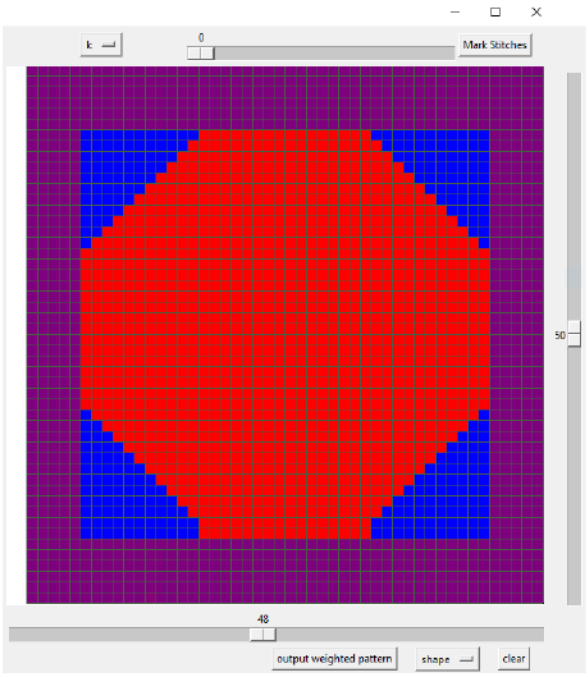


Figure 7: Example of imagery region mapping painting tool.

gions on a grid. Multiple imagery objectives can be defined over differently colored regions with the same sub-objective and different target values. Target values between 0 and 1 are represented as the range of colors between blue and red. For example, as in Figure 7, the blue region will have a target value of 0 and the red region will have a target value of 1. The painting tool can be used multiple times when creating a heuristic map to create regions with different objectives. Previously painted regions can be loaded in, so the programmer does not have to paint the same region repeatedly.

Texture Tactics

Tactics modify one random stitch in a texture per iteration. In our library we modify stitches in five ways: (1) *flip* its orientation (i.e.back-to-front and front-to-back), (2) *lean* it in the opposite direction (i.e.left to right, right to left, centered remains centered); (3) *replace* a stitch with a programmer specified stitch type or set of stitches (e.g., increase-decrease pair, cables); (4) copy a *mirrored* stitch across a symmetry-axis; and (5) *copy* a stitch from another texture in the population by searching for the texture that scores highest on the objective mapped to this tactic. We tend to use the *flip*, *lean*, *replace*, and *mirror* tactics in hill-climbing optimizers where these modifications are likely to affect some objective value. We use *copy* in genetic optimizers where two textures in the population are merged by trading off discovered high-value stitches for separate objectives. Note this is not an exhaustive set of possible stitch-level tactics, however it is effective at producing a wide variety of functional textures.

KnitGIST Design Space

These objectives reveal inherent trade offs between various texture properties. There are four key trade offs across these objectives. First, curl is opposed to shrinkage and, by extension, elasticity. Curl is produced by aligning stitch orientations, while shrinkage is produced by alternating those orientations. The same tactics (flip) are generally effective at producing both, but prioritizing both objectives will create a conflict. Second, there is a trade off between horizontal and vertical elasticity; as a texture stretches one direction, it resists stretch in the other. Third, there are aesthetic trade offs between different styles of textures. Specific types of stitches dominate particular texture styles. Tactics that increase the presence of these stitch types will push a texture into one style category or the other. Further, the fourth trade off is between specific physical properties that are related to stitch types and style. Specifically, opacity is driven by the presence of yarn overs which also is the strongest indicator of texture style.

KnitGIST's set of tactics are largely independent of these trade offs allowing broad exploration of the space. This reduces algorithmic efficiency by not exploiting the properties associated with specific objectives, but ensures exploration of the search space given a range of objectives. By specifying weights on objectives, the programmer makes their priorities explicit which will determine which objectives are prioritized when trade offs arise. Should a programmer define a heuristic map with equal priorities across conflicting objectives, it is unlikely that a solution will be discovered. Instead, the final highest scoring candidate will likely be a compromise between the two conflicting objectives. In such cases, intermediary KnitSpeak candidates are available to the programmer which can be used by the programmer to reconstruct the heuristic map to improve results. Overall, the KnitGIST framework and library are tailored to exploration of a large search space given intuitive objectives and simple tactics, at the cost of algorithmic efficiency and specificity.

DEMONSTRATIONS

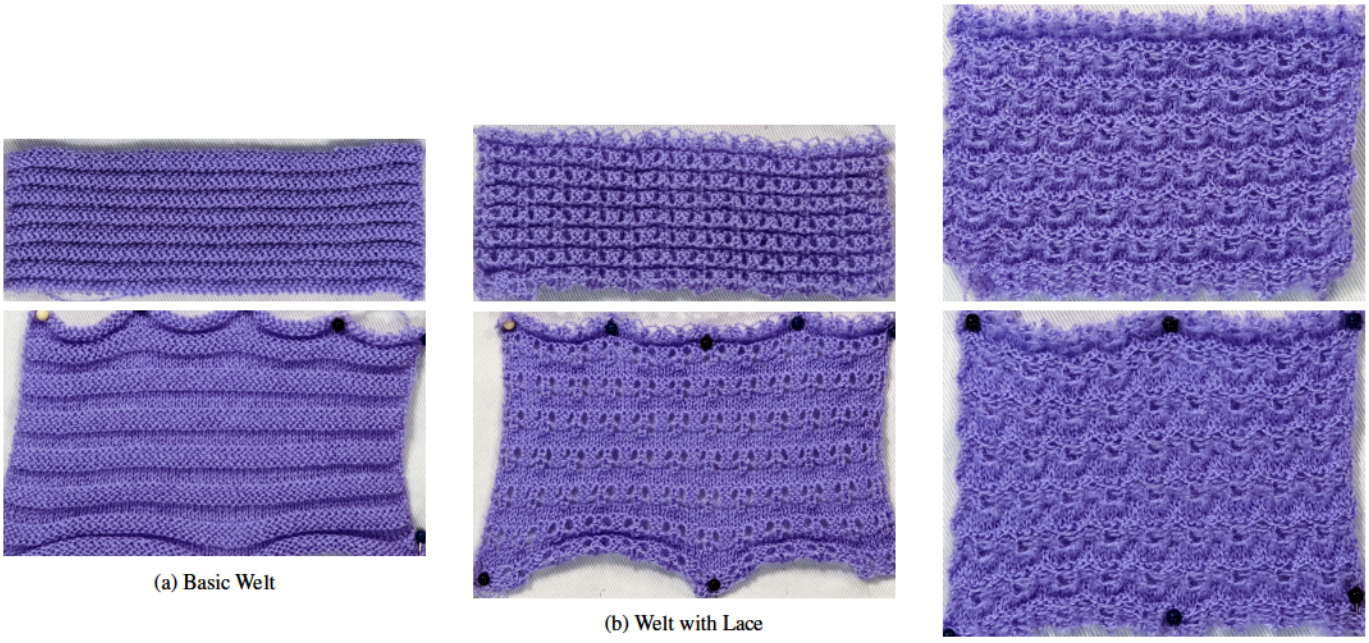
In this section we will construct a series of optimizers for generating functional and attractive textures and finally a whole knit object. Each of these demonstrations make use of the objectives and tactics provided in our library. We combine these objective and tactics in different heuristic maps to demonstrate how KnitGIST can produce a wide range of textures.

Functionality: Generating Functional Textures

Table 4: Welt Heuristic Map

Welt Type	Objective Weight	Objective	Tactic	Tactic Weight
All	3	Forward-Curl Stripe Region	Flip Stitch	1
All	3	Backward-Curl Stripe Region	Flip Stitch	1
All	2	Maximize ↕ Elasticity	Flip Stitch	2
		Mirror Stitch	1	
Lace	1	Lace Style	Replace with increase-decrease pair	1
Cable	1	Cable Style	Replace with cable stitch	1

In our initial scenario, we demonstrated how a simple mapping between objectives and tactics produced a set of canonical knit textures that curl, shrink, and stretch (see Fig 1).



(a) Basic Welt

(b) Welt with Lace

(c) Welt with Cables

Figure 8: We generate welt textures using functional objectives that alternate the direction of curls in horizontal bands. Aesthetic properties are applied to the Lace (b) and Cable (c) welt. The alternating curls maximize vertical elasticity.

In this demonstration we build on this scenario to produce a variety of welting-textures with desired aesthetic and physical properties. Welts are textures with alternating horizontal stripes of stitches with opposite orientations. The constant orientation of each stripe causes the stripe to curl; switching to the opposite orientation causes the texture to curl backwards and shrinks vertically. This produces a vertically elastic texture.

We use three objectives to generate a plain knit-purl welt (Table 4) (Fig. 8a). The first is a region map that marks the stripes that should curl forward. The second maps stripes that curl backwards. The third objective sets a high target vertical-elasticity. The curl objectives map to a flip tactic which may increase curl by aligning stitch orientations. The same tactic is mapped to the elasticity-objective. In addition to the flip tactic, we map a secondary mirror tactic which replaces a stitch with a stitch equidistant across a center-vertical axis. This will increase the vertical symmetry of the texture. While the symmetry objective is not being used, we know that vertically elastic textures tend to be symmetrical, so this approach is a useful heuristic.

Given this structure for generating a basic welt, we can add aesthetic objectives to extend the heuristic map to create cable-welts and lace-welts. To do this we add an objective for either ensuring the texture has a cable style, or a lace style. The cable objective maps to a replace tactic which replaces a stitch with a cable. Respectively, the lace objective maps to a replace tactic with increase-decrease pairs. The resulting cable-welt is shown in Figure 8c, and the lace-welt is shown in Figure 8b. The heuristic map for each of these welts is described in Table 4. Note that the cables and increase-decrease pairs are only present in alternating stripes, even though we did not specify a region for cables or lace. Since the added cables, increases, and decreases all have the same alignment they tended to collect in regions of positive curl. The eyelets and cables appear evenly spaced because they made up nearly every stitch in the positive curl stripes, creating a dense uniform texture.

Aesthetics: Merging Textures and Shape

Table 5: Lace-Hexagon heuristic map

Objective Weight	Objective	Tactic	Tactic Weight
3	Lace Hexagon Region	Copy Stitch from texture	2
		Replace with increase-decrease pair	1
3	Knit Border Region	Copy Stitch from texture	2
		Replace with knit	1
2	Compressive Frame Region	Copy Stitch from texture	2
		Flip Stitch	1
1	↔ Symmetry	Mirror Stitch	1
1	↕ Symmetry	Mirror Stitch	1

Next we demonstrate how to define textures over arbitrary shapes, allowing for more complex texture design. Consider the challenge of creating a lace texture over an arbitrary 2D shape, such as a hexagon. Lace textures are complex because

Figure 9: Resulting hand-knit lace-hexagon texture. Color work added to denote regions and not specified by KnitGIST.

Figure 10: Hand-knit Lamp-shade texture stretched around frame. The texture has alternating vertical stripes of blue lace and green cables. The bottom and top edge are grey regions of backwards curl which grip the top and bottom of the frame.

of the pairing of increases and decreases. While we may normally assume that stitches fall on a grid, increases and decreases change the number of loops on each row. If two adjacent rows have different loop counts, at least one loop will be left with no child loop. This will cause the fabric to unravel. To pair increases and decrease, a designer would normally have to selectively position every pair to avoid unraveling.

In this example we construct the heuristic map to merge textures that score highly on different objectives to generate a final texture which has a lace hexagon bordered by curly all-knits, and a frame of shrunken seed-stitch (alternating knits and purls). The heuristic map consists of three region-map objectives, and vertical and horizontal symmetry objectives. Three regions (i.e.the frame, the hexagon, and the hexagon-border) are drawn out in Figure 7 on our painting tool. The frame region has a regional objective that it be both highly vertically and horizontally shrunk. The hexagon region maps to a regional objective that it be a lace-style texture. Finally, the hexagon border region maps to a regional objective that it curls forward which, as we demonstrated in the scenario, will result in an all knit texture.

All three regions map to a copy tactic. Over time, textures in the population will tend to score highly on some of the objectives, but not all. The copy tactic enables textures that score highly on separate objectives to transfer high-value stitches across the population. The hexagon-region has a secondary tactic of replacing a stitch with an increase-decrease pair to make lace. The hexagon-border region has a secondary tactic to replace a stitch with a knit-stitch, increasing local curl. The frame region has a secondary tactic to flip the orientation of a stitch, which as we have seen in both the last demonstration and the leading scenario, can increase shrinkage. Finally, both symmetry objectives map to mirror tactics which replace a stitch with the stitch that mirrors it across the respective axis. This heuristic map produces the lace hexagon (Figure 9).

Full-Objects: Lacey Lampshade

Now that we have demonstrated how we can use physical and aesthetic objectives to guide the KnitGIST optimizers, we will demonstrate how this can be used to reason about functional knit objects. Consider the properties of a lamp shade. It should allow some light to shine through, but diffuse enough to dampen the bulb. Additionally, lampshades are often conical with a top that is narrower than the base. This can be done by shaping the knit texture but it can also be shaped by a rectangular swatch of fabric that is highly-horizontally elastic which allows the top to remain compressed while the base is stretched wide. Finally, the top and bottom of the texture should bend or curl to cover the top and bottom of the lamp-shade frame. By bending around the frame, the only post-processing required will be to seam the texture into a tube and stretch it over the frame.

In this demonstration, the heuristic map uses four objectives (three functional and one aesthetic choice): (1) the curled-edges region that requires the top and bottom edges of the texture to curl backwards, (2) the low-opacity objective that reduces the opacity of the texture, (3) an objective that maximizes horizontal elasticity, and (4) an additional aesthetic

Table 6: Lampshade Heuristic-Map

Objective Weight	Objective	Tactic	Tactic Weight
4	Curled-Edges Region	Flip Stitch	1
3	Low-Opacity	Replace with Increase Decrease Pair	1
2	Maximize ↔ Elasticity	Flip Stitch	2
		Replace with Cable Stitch	1
1	Cable Style	Replace with Cable Stitch	2
		Replace with Inc Slip-Dec Pair	1

objective that the texture is categorized as a cable pattern which increases the textures complexity by adding complex and attractive stitches. The curl objective maps to the versatile flip tactic. The opacity objective maps to the replace with increase-decrease pair tactic, which will add eyelets that allow light through. We include two tactics to increase elasticity, the standard flip tactic and a low priority objective that adds cable stitches which increase horizontal shrinkage. Finally, the cable-texture objective is mapped to the add cable-stitch tactic, and a tactic that adds *slipped-decrease pairs*. Slipped-decreases change the order that parent loops overlap each other which causes them to be visually similar to cable-stitches (see Fig 6). They are often used in cable textures to hide yarn-overs.

The resulting texture (Fig 10) consists of three noticeable regions which we highlight with different color yarns. The grey region at the top and bottom of the fabric consists of all purl stitches. This causes both ends to curl backwards, making it easier to seam fabric to the lamp shade frame. The green region consists of vertical stripes of cable stitches. The stripes mostly alternate in lean direction, giving the appearance of ropes, though they are not as consistent as a designer might have selected. The light blue regions consists of centered, slipped, triple decreases and two yarn overs. This particular decrease looks similar to cable-stitches, but the added yarn-overs allow more light to shine through.

LIMITATIONS AND FUTURE WORK

KnitGIST lays the groundwork for creating flexible generative design optimizers for constructing a wide variety of knit textures. However, a graphical representation of these textures and an intuitive tool for editing textures while maintaining their knitabilty, remains the logical next step. As KnitGIST stands, this is a programming tool for creating knitting generative design tools. Our library of objectives and tactics provide a basis for generative design optimizers where the programmer does not need to understand how each objective and tactic works. Programmers who develop objectives and tactics require expertise in knitting and a functional understanding of the KnitGraph structure. Other, non-knitting experts can amplify this knowledge by reusing objectives and tactics to create generators. This library is limited to a small set of useful physical and aesthetic objectives, but should be extended to include other objectives and tactics. For example, an extended library could included evaluation of how these modifiers of physical characteristics (e.g., curl, shrinkage, stretch, opacity) effect physical characteristics that are dependent on fiber properties (e.g., stress/strain and tensile breaking strength). Finally, the KnitGIST framework trades off efficiency for extensibility and generality of the optimizers. More efficient optimizers could be constructed, particularly if continuous-optimization methods were applied. However, such efficiency would require extensive domain-knowledge.

CONCLUSION

In this paper, we contribute the Generative Instantiation Synthesis Toolkit for Knitting (KnitGIST) which enables programmers to easily map objectives to tactics which would generate knit textures. We include objectives that assess physical properties (e.g., curl, shrinkage, elasticity, and opacity) and aesthetic properties (e.g., symmetry, style, and imagery) of knit textures and a set of simple tactics which modify random stitches to step towards a texture that scores highly over these objectives.

We present three demonstrations the utility of KnitGIST. First, we use the physical-objectives to generate springy lace and cable textures with varying aesthetic properties. Next, we use KnitGIST to apply a lace texture over an arbitrarily shaped region while maintaining the loop-to-loop connections that ensure the final texture will not unravel. Finally, we combine physical and aesthetic objectives to craft a lace lamp shade which stretches to fit around a lamp shade frame.

The core contribution of KnitGIST is that by optimizing over knit-texture programs, we consistently produce textures that are machine knittable and will not unravel, yet meet user specified goals, all without requiring a detailed knowledge of knitting structures. The KnitSpeak language and compiler manage hard-constraints on knit textures, leaving programmers more freedom to pair objectives and tactics to achieve functional and aesthetic results. This freedom enables modularity which will be valuable when constructing generative knit design interfaces.

ACKNOWLEDGMENTS

Figures 3 and 4 are used with permission of Hofmann et al.[13]. This work was funded by National Science Foundation Grants: IIS-1718651, IIS-1907337, and 2031801.

REFERENCES

[1] Lea Albaugh, Scott Hudson, and Lining Yao. 2019. Digital Fabrication of Soft Actuated Objects by Machine Knitting. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1–13. DOI:http://dx.doi.org/10.1145/3290605.3300414

[2] Fraser Anderson, Tovi Grossman, and George Fitzmaurice. 2017. Trigger-Action-Circuits: Leveraging Generative Design to Enable Novices to Design and Build Circuitry. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. Association for Computing Machinery, New York, NY, USA, 331–342. DOI: http://dx.doi.org/10.1145/3126594.3126637

[3] ANSYS 2020. Topology Optimization Software | ANSYS. (2020). https://www.ansys.com/products/structures/topology-optimization Library Catalog: www.ansys.com.

[4] Autodesk 2020. Inventor | Mechanical Design & 3D CAD Software | Autodesk. (2020). https://www.autodesk.com/products/inventor/overview Library Catalog: www.autodesk.com.

[5] Peter J. Bentley and David W. Corne. *Creative evolutionary systems*. Morgan Kaufmann, San Francisco, CA.

[6] James M. Bern, Kai-Hung Chang, and Stelian Coros. 2017. Interactive Design of Animated Plushies. *ACM Trans. Graph.* 36, 4, Article Article 80 (July 2017), 11 pages. DOI:http://dx.doi.org/10.1145/3072959.3073700

[7] JC Briar. 2013. Stitch Maps. (2013). https://stitch-maps.com/

[8] L Caldas and J Duarte. 2004. Implementational issues in generative design systems. In *First international conference on design computing and cognition*.

[9] Xiang "Anthony" Chen, Ye Tao, Guanyun Wang, Runchang Kang, Tovi Grossman, Stelian Coros, and Scott E. Hudson. 2018. Forte: User-Driven Generative Design. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, New York, NY, USA, Article Paper 496, 12 pages. DOI: http://dx.doi.org/10.1145/3173574.3174070

[10] Jérémie Dumas, An Lu, Sylvain Lefebvre, Jun Wu, and Christian Dick. 2015. By-Example Synthesis of Structurally Sound Patterns. *ACM Trans. Graph.* 34, 4, Article Article 137 (July 2015), 12 pages. DOI: http://dx.doi.org/10.1145/2766984

[11] James Fogarty and Scott E. Hudson. 2003. GADGET: A Toolkit for Optimization-Based Approaches to Interface and Display Generation. In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology (UIST '03)*. Association for Computing Machinery, New York, NY, USA, 125–134. DOI:http://dx.doi.org/10.1145/964696.964710

[12] Grasshopper 2020. Millipede - Addon for Grasshopper. (2020). https://grasshopperdocs.com/addons/millipede.html Library Catalog: grasshopperdocs.com.

[13] Megan Hofmann, Lea Albaugh, Ticha Sethapakadi, Jessica Hodgins, Scott E. Hudson, James McCann, and Jennifer Mankoff. 2019. KnitPicking Textures: Programming and Modifying Complex Knitted Textures for Machine and Hand Knitting. In *Proceedings of the 32Nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. ACM, New York, NY, USA, 5–16. DOI: http://dx.doi.org/10.1145/3332165.3347886

[14] jenny lin. 2017. Knitting with Knitout Part One: Hello World. (Nov. 2017). /posts/2017/11/27/kout1/ Library Catalog: textiles-lab.github.io.

[15] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 215–224. DOI:http://dx.doi.org/10.1145/1806799.1806833

[16] Ayelet Karmon, Yoav Sterman, Tom Shaked, Eyal Sheffer, and Shoval Nir. 2018. KNITIT: A Computational Tool for Design, Simulation, and Fabrication of Multiple Structured Knits. In *Proceedings of the 2nd ACM Symposium on Computational Fabrication (SCF '18)*. Association for Computing Machinery, New York, NY, USA, Article Article 4, 10 pages. DOI: http://dx.doi.org/10.1145/3213512.3213516

[17] Alexandre Kaspar, Liane Makatura, and Wojciech Matusik. 2019. Knitting Skeletons: A Computer-Aided Design Tool for Shaping and Patterning of Knitted Garments. In *Proceedings of 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. ACM, New Orleans, LA.

[18] Rubaiat Habib Kazi, Tovi Grossman, Hyunmin Cheong, Ali Hashemi, and George Fitzmaurice. 2017. DreamSketch: Early Stage 3D Design Explorations with Sketching and Generative Design. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology (UIST '17)*. Association for Computing Machinery, New York, NY, USA, 401–414. DOI:http://dx.doi.org/10.1145/3126594.3126662

[19] Terry W. Knight. 1998. *Designing a Shape Grammar*. Springer Netherlands, Dordrecht, 499–516. DOI: http://dx.doi.org/10.1007/978-94-011-5121-4_26

[20] Knit It Now LLC. 2013. KnitItNow Pattern Library. [Online]. Available from: https://www.knititnow.com/knit/catalog.cfm. (2013).

[21] Sivam Krish. 2011. A practical generative design method. *Computer-Aided Design* 43, 1 (2011), 88 – 100. DOI:http://dx.doi.org/https://doi.org/10.1016/j.cad.2010.09.009

[22] Jiahao Li, Jeeeun Kim, and Xiang "Anthony" Chen. 2019. Robiot: A Design Tool for Actuating Everyday Objects with Automatically Generated 3D Printable Mechanisms. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, New York, NY, USA, 673–685. DOI: http://dx.doi.org/10.1145/3332165.3347894

[23] Jenny Lin, Vidya Narayanan, and James McCann. 2018. Efficient Transfer Planning for Flat Knitting. In *Proceedings of the 2Nd ACM Symposium on Computational Fabrication (SCF '18)*. ACM, New York, NY, USA, Article 1, 7 pages. DOI: http://dx.doi.org/10.1145/3213512.3213515

[24] Jonàs Martínez, Jérémie Dumas, Sylvain Lefebvre, and Li-Yi Wei. 2015. Structure and Appearance Optimization for Controllable Shape Design. *ACM Trans. Graph.* 34, 6, Article Article 229 (Oct. 2015), 11 pages. DOI:http://dx.doi.org/10.1145/2816795.2818101

[25] James McCann, Lea Albaugh, Vidya Narayanan, April Grow, Wojciech Matusik, Jennifer Mankoff, and Jessica Hodgins. 2016. A Compiler for 3D Machine Knitting. *ACM Trans. Graph.* 35, 4 (July 2016), 49:1–49:11. DOI: http://dx.doi.org/10.1145/2897824.2925940

[26] Clare Mountain-Manipon. 2018. How to Grade Knitting Patterns Using a Spreadsheet | Sister Mountain. (May 2018). https://www.sistermountain.com/blog/grade-knitting-patterns-spreadsheet

[27] Chandrakana Nandi, James R. Wilcox, Pavel Panchekha, Taylor Blau, Dan Grossman, and Zachary Tatlock. 2018. Functional Programming for Compiling and Decompiling Computer-Aided Design. *Proc. ACM Program. Lang.* 2, ICFP, Article Article 99 (July 2018), 31 pages. DOI:http://dx.doi.org/10.1145/3236794

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

[29] Vidya Narayanan, Kui Wu, Cem Yuksel, and James McCann. 2019. Visual Knit Programming. *ACM Trans. Graph.* 38, 4 (July 2019).

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

[31] Ravelry 2017. Ravelry: Home. (2017). https://www.ravelry.com/

[32] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2016. Stochastic Program Optimization. *Commun. ACM* 59, 2 (Jan. 2016), 114–122. DOI: http://dx.doi.org/10.1145/2863701

[33] Shima Seiki. 2011. SDS-ONE Apex3. [Online]. Available from: http://www.shimaseiki.com/product/design/sdsone_apex/flat/. (2011).

[34] Mélina Skouras, Bernhard Thomaszewski, Stelian Coros, Bernd Bickel, and Markus Gross. 2013. Computational Design of Actuated Deformable Characters. *ACM Trans. Graph.* 32, 4, Article Article 82 (July 2013), 10 pages. DOI: http://dx.doi.org/10.1145/2461912.2461979

[35] Soft Byte LTD. 2012. DesignaKnit 8. [Online]. Available from: https://softbyte.co.uk/designaknit.htm. (2012).

[36] Lesley Stanfield and Melody Griffiths. 2010. *The Essential Stitch Collection.* The Reader's Digest Association, Inc.

[37] Stoll. 2011. M1Plus pattern software. [Online]. Available from: http://www.stoll.com/stoll_software_solutions_en_4/pattern_software_m1plus/3_1. (2011).

[38] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 287–296. DOI: http://dx.doi.org/10.1145/2491956.2462174

[39] E. Ulu, J. McCann, and L. B. Kara. 2019. Structural Design Using Laplacian Shells. *Computer Graphics Forum* 38, 5 (2019), 85–98. DOI: http://dx.doi.org/10.1111/cgf.13791

[40] Chenming Wu, Haisen Zhao, Chandrakana Nandi, Jeffrey I. Lipton, Zachary Tatlock, and Adriana Schulz. 2019. Carpentry Compiler. *ACM Trans. Graph.* 38, 6, Article Article 195 (Nov. 2019), 14 pages. DOI: http://dx.doi.org/10.1145/3355089.3356518

[41] Yahan Zhou, Evangelos Kalogerakis, Rui Wang, and Ian R. Grosse. 2016. Direct Shape Optimization for Strengthening 3D Printable Objects. *Comput. Graph. Forum* 35, 7 (Oct. 2016), 333–342.