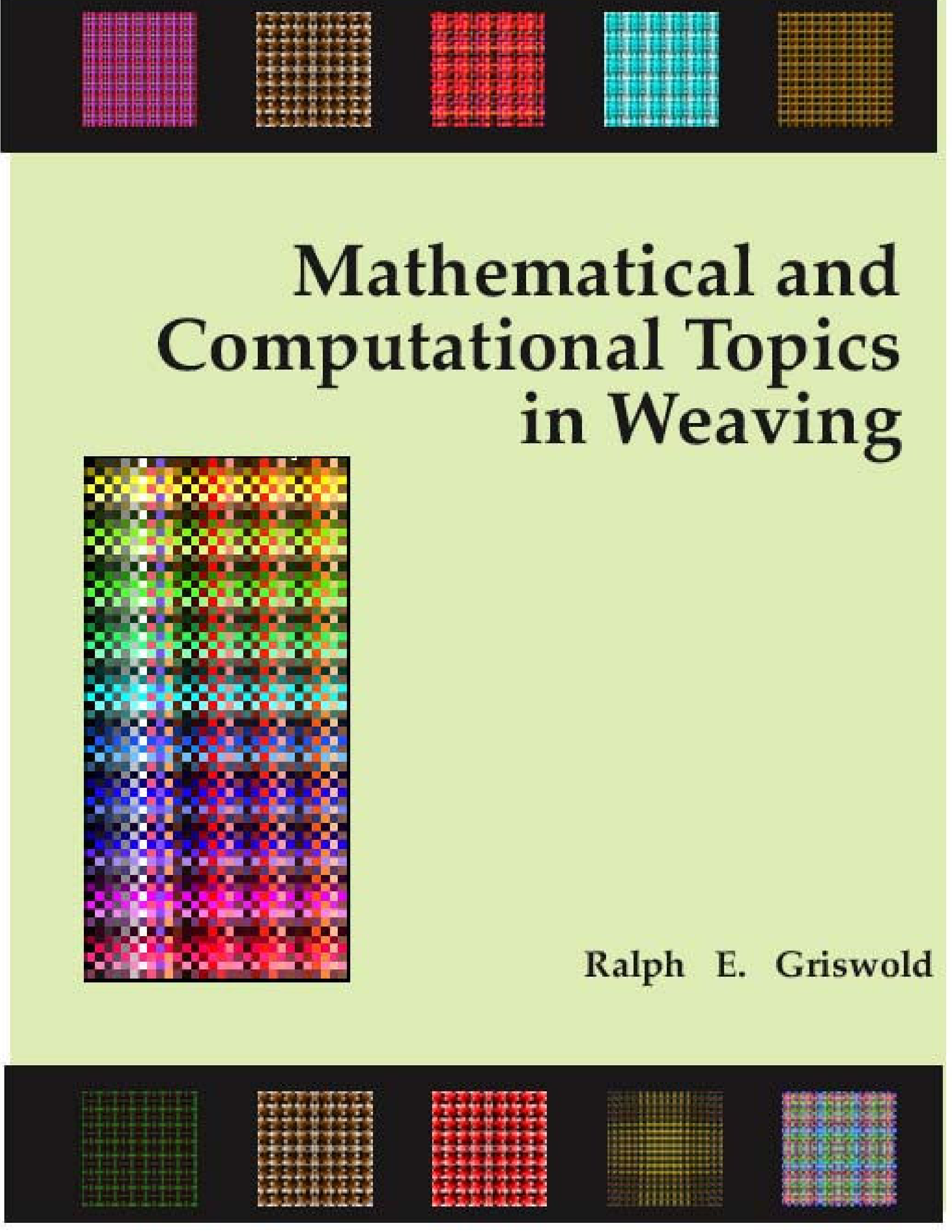


27 Sep 2006

Mathematical and Computational Topics in Weaving

Over a period of years, I've written a variety of article related to the use of elementary mathematics and computation in weaving.

Years ago — more than I'm comfortable with admitting — I decided it would be worthwhile to bring these articles together in a "book". The quotation marks indicate that it might not be a book in the ordinary sense. In any event, it would be published on the Web, free to all.

The problem I had was how to put it all together and have a sensible, coherent result. I've not completely resolved the problem, but I've made progress and the results, with all their flaws, are now available in draft form.

The progress toward a book has largely been the result of encouragement and participation by two skilled weavers, Ruth Blau and Marg Coe. My longtime friend and colleague, Gregg Townsend, has provided invaluable help with the mathematical and computational aspects of the book. When the books is better developed, more detailed and appropriate acknowledgments to them and the many others who helped over the years will be forthcoming.

As to the draft status of the book, some sections are well developed and in penultimate form. Others range from "okay" to downright awful. There are many known errors remaining to be corrected. And many sections are missing. Most noticeably, the "connective tissue" to bring it all together is largely lacking.

As a work in progress, changes will occur frequently and not be specifically announced unless there is a major change.

Comments, notes of errors, and so forth are welcome. But understand that it may take some time to deal with them. Please resist the urge to make suggestions for major changes to the book. I have neither the time nor the energy for these, however meritorious they may be.

The book is available through links to PDFs. See the table of contents that follows. Some links lead directly to PDFs. The links are active; you can just click on them (if this doesn't work, let me know and I'll fix it). Others lead to other links. The absence of a link indicates the section has not been written or is too incomplete to include. Pleaase let me know of bad links.

Navigating is not easy. My priorities are in completing the book, not making it more easily accessible in draft form. I hope, nonetheless, that what's there will be interesting and useful.

Ralph E. Griswold

September 20, 2006

ralph@cs.arizona.edu

Contents

A. Cover

B. Front Matter

1. Acknowledgments
2. Contents
3. Preface
   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/B/Preface.pdf
4. Introduction

C. Terminology and Notation

1. Weaving Context
2. Notation

D. Some Simple Applications of Mathematics to Weaving

1. Twill Counters
   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/D/TwillCounters.pdf
2. Satin Counters
   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/D/SatinCounters.pdf
3. Sequence Drafting
   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/D/SequenceDrafting.pdf
4. Straight Draw Threading Conversion
   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/D/ThreadingConversion.pdf
5. Fabric Analysis
   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/D/FabricAnalysis.pdf

E. A Case Study of a Weaving Technique

1. Introduction
2. Name Drafting
   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/E/Name.pdf

F. Case Studies of Specific Weaves

1. Crackle Weave

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/F/Crackle.pdf

1. Shadow Weave

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/F/ShadowWeave.pdf

G. Patterns

1. Pattern Substitution
2. Cellular Automata
3. Constrained Patterns

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/G/Constraints.pdf

4. Nonlinear Grid Design

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/G/GridLayouts.pdf

5. Operations on Patterns

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/G/PatternOperations.pdf

6 . Pattern Tours

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/G/Tours.pdf

7. Grid Overlays

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/G/GridOverlays.pdf

8. Permutations
9. Line Patterns

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/G/LinePatterns.pdf

10. Complementation
11. Pattern-Extension Schemata
12. Gaussian Primes
13. Pantactic Design

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/G/PantacticSquares.pdf

H. Sequences

1. Introduction

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/Introduction.pdf

2. Residue Sequences

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/ResidueSequences.pdf

3. Simple Integer Sequences

4. Recurrence Relations

   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/RecurrenceRelations.pdf
5. The Fibonacci Sequence
6. Fractal Sequences

   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/FractalSequences.pdf
7. The Morse-Thue Sequence

   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/MorseThue.pdf
8. Signature Sequences

   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/SignatureSequences.pdf
9. Spectra Sequences

   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/SpectraSequences.pdf
10. Chaotic Sequences
11. Continued Fractions

    http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/ContinuedFractions.pdf
12. Farey Sequences

    http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/FareyFractions.pdf
13. Term Replication Sequences

    http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/TermReplication.pdf
14. Algebraic Expressions

    http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/AlgebraicExpressions.pdf
15. Meandering Sequences

    http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/MeanderingSequences.pdf
16. Friendly Sequences

    http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/FriendlySequences.pdf
17. Smarandarche Sequences

    http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/H/Smarandache.pdf

I. Structure

1. Sound Interlacements

   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/I/ProblemDrafts.pdf
2. Color Draftability

   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/I/ColorDraftability.pdf
3. Maximal Color Patterns

   http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/I/MaximalPatterns.pdf

4. Characterizing Weave Structure

J. Formal Approaches

1. Boolean Design

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/J/BooleanDesign.pdf

2. L-Systems

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/J/L-Systems1.pdf

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/J/L-Systems2.pdf

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/J/L-Systems3.pdf

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/J/L-Systems4.pdf

3. Cellular Automata

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/J/CellularAutomata.pdf

4. A T-Sequence Language

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/J/T-Sequences.pdf

K. Examples of Advanced Applications

1. Introduction

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/K/Introduction.pdf

2. Painter's Weaving Language

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/K/PWL.pdf

3. Boolean Design

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/K/BooleanDesign.pdf

4. Color Design

http://www.cs.arizona.edu/patterns/weaving/webdocs/mo/K/ColorDesign.pdf

5. Sequence-Based Design

L. Conclusions

M. Appendices

1. Mathematical Notes
2. Web Resources
3. Gallery

N. References

O. Glossary

P. Index

27 Sep 2006

Preface

By title, I am a professor of computer science. My background is in computer programming language design and implementation, software methodology, and more recently program visualization and graphics programming. I have always implemented my ideas with programs. I am, above all, a programmer.

I retired early to have more time for my intellectual interests, which at the time were computer graphics and investigation of pattern-construction techniques.

Until a few years ago I knew virtually nothing about weaving. There were no weavers among my family and friends. I never encountered weaving in my education — not even Pot Holders 101. I have never seen a loom close hand, much less seen one being used.

I "discovered" weaving accidentally. I was exploring a computational problem related to pattern construction. In the process, I was experimenting with Painter®, an advanced program for graphic artists. I found its weaving feature fascinating and was intrigued by the patterns that could be formed within the constraints posed by weaving. But what really caught my attention was Painter's underlying "programming language" for creating weaving drafts. The nature of this language fascinated me and touched on several professional interests of mine. With this, I was inexorably drawn into weaving.

From that time my life was changed. I dug into the few weaving books that were readily available and bought a copy of a current weaving magazine. This launched me into an intensive effort to learn about weaving.

I found learning about weaving from books more difficult than I expected. I attribute part of this to the fact that there are technical aspects in weaving that must be presented to weavers who may not have technical

backgrounds and no doubt sometimes written by weavers without technical backgrounds either. This leads to imprecision and incompleteness where I wanted precision and completeness. Another thing that I learned in this regard is that most weaving books take for granted some knowledge, such as what it means to "weave tabby" with the result that a complete novice such as I was at the time is left puzzled by apparent omissions.

On the other hand, I learned long ago that there is some benefit in coming new to a field without the preconceptions and presumptions that knowledgeable persons have acquired from others — ignorance has its advantages. Also, the critical novice can question conventional wisdom, which sometimes is misleading if not downright wrong. Accepting conventional wisdom is easy and convenient, but it also tends to make new ideas "unthinkable". I tried to keep these things in mind when learning about weaving.

In addition to reading, I also corresponded by e-mail with several weavers who were willing to help a novice. On the recommendation of one of these weavers, I joined Complex Weavers (with considerable trepidation), initially to gain access to its excellent lending library. But soon I joined several study groups so that I could learn more about some topics from experts.

I explored weaving programs, first trying free ones and demonstration versions of commercial programs, but I finally purchased several programs to see how capable programs worked, what kinds of features they supported, and what their conceptual bases were.

I have learned a great deal about weaving in the last few years, and continually learn more. All this has revealed to me how little I know and will ever know of this vast culture and body of knowledge. But knowledge was not all I wanted. I wanted to do things. And it was and is my perception that there were contributions to be made in my areas of interest.

I had not been long into my learning endeavor when it seemed to me that almost everything I had done professionally was applicable to weaving — a hidden preparation, as it were. And that preparation was in mathematics and computation.

With a different background and at another time in my life, I probably would have bought a loom and taken up actual weaving with a vengeance. But I could not do that and also explore mathematical and computational topics in weaving. I chose the latter. I realize that by not being an actual weaver, I am missing a great deal and that there are gaps in my knowledge and understanding. I made my decision knowing this.

As mentioned earlier, throughout my professional career I have used the process of writing programs to verify my ideas and to increase the depth of my

understanding. In fact, programming is a research technique for me. It exposes hidden assumptions and flaws in reasoning, and almost always leads to new ideas. A program also makes it possible to try things that are too tedious, time consuming and error prone to do by hand.

I have written hundreds of programs related to various aspects of weaving, ranging from simple utilities to a full-blown interactive program for weave design, albeit unlike any of the existing commercial ones.

I started to write down my ideas and results. Like programming, writing is a research tool for me, and for many of the same reasons.

I published my first writings in the *Icon Analyst*, a newsletter for advanced computer programmers. Few of the readers, if any, knew much about weaving, so I started with tutorials, which provided an excellent way for me to clarify my own thinking. Next I described Painter's weaving language, and then went on to some of my own work.

Later I started to write short articles and publish them on the Web: easy, convenient, and readily accessible to others. I spruced up a couple of the less technical articles and published them in *Complex Weavers Journal*.

A couple of years ago, I started to think about writing a book containing the material I had developed. (Richard Feynman, a Nobel Laureate in physics once commented "A professor is a person who doesn't know when to stop talking". I would add that a professor is a person who doesn't know when to stop writing.) I began to see the articles I was publishing on the Web as preliminary drafts of material for a book. By publishing my work in separate articles, I was free of having to worry about how things fit together. For a book, I began to worry about this, andit, not surprisingly, has been the major difficulty I've faced — trying to make a somewhat coherent "whole". Of course in trying to do this, I have learned a great deal and come upon new ideas.

One of the major problems I've had with the whole concept was "Who could possibly use this book?". Most weavers do not have a background or an interest in mathematics and computation. A subsidiary question was to what extent even those few weavers with the necessary background could actually apply the ideas and methods in the book to weave design? To be useful in practice, much of the material in the book requires computer programs — programs that are not supplied. Many, however, are easy for an experiences programmer to write.

Despite all these misgivings, I have decided to publish this book. If it succeeds in giving only a few weavers ideas and inspiration, the effort will have been worthwhile. And as time passes, there will be more weavers with the background to use the material in it.

I have chosen to publish this book on the Web for several reasons. The first and foremost is to make it freely and widely available. As an alternative, paper publication has its advantages and disadvantages, and self publishing is reasonably easy with the present technology. However, a printed book cannot be provided free. There also are numerous administrative problems with paper publication that I would prefer to avoid. If you are wondering why I didn't seek a commercial publisher, the reason is that there simply is not an adequate market to make such a project financially viable.

Here it is, yours for the taking. Look through it, read it, and perhaps dream of the possibilities.

Ralph E. Griswold
Otero House
Tubac, Arizona

June 5, 2002

Tucson, Arizona

September 21, 2006

Twill Counters

Twills are described by *counters* that show what shafts are raised and not raised.

Sidebar on what twill counters are by Marg/Ruth.

Notation

Two notations are in common use for counters. In one, there is a horizontal line with pairs of numbers above and below, alternating, to show shafts raised and not raised (on a rising-shaft loom). For example,

$$\begin{array}{cccc} 2 & & 3 & \\ \hline & 2 & & 1 \end{array}$$

describes an 8-shaft twill counter in which the first two shafts are raised, the next two are not, the next three are, and the last, not.

This form of stacked notation is easy to understand, but it is typographically difficult to produce and needs to be set apart from lines of text. An alternative, linear notation uses separating slashes to indicate the over/under sequence. In this notation, the example above would be written 2/2/3/1.

The difficulty with the linear notation is in keeping track of the over/under order. Nonetheless, the ease with which the linear notation can be written and used in text generally makes it the favored notation.

Twill Tie-Ups

In *regular twills*, the twill counter appears rotated by one for each successive row of the corresponding tie-up. Figure Ω.1 shows the tie-up for the twill counter in the example above.

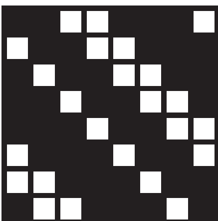


Figure 1. A 2/2/3/1 Twill Tie-Up

In this example, the rotation is to the right. This is called a *right twill* and is the one usually shown in examples. Rotation also can be to the left, producing a *left twill*.

As the twill counter is rotated to the right, parts of it move off the right end and onto the left end. The third row of the tie-up in Figure 1, taken as a twill counter, would look like this:

$$\frac{1 \quad 2 \quad 2}{1 \quad 2}$$

This is not a valid counter because there are more terms above the line than below. What has happened is that the 3 on the top of the original counter has been split into two parts: 2 at the end and 1 at the beginning.

The second row of the tie-up in Figure 1, on the other hand, would look like this as a counter:

$$\frac{\quad 2 \quad 3}{1 \quad 2 \quad}$$

This violates the convention that twill counters start with raised shafts.

The fourth row of the tie-up in Figure 1 corresponds, structurally, to a valid twill counter:

$$\frac{3 \quad 2 \quad}{\quad 1 \quad 2}$$

This counter is, in fact, equivalent to the original twill counter in our example:

$$\frac{2 \quad 3 \quad}{\quad 2 \quad 1}$$

The difference is that the 3/1/2/2 counter is a rotated version of the 2/2/3/1 counter.

Since rotation of a counter in multiples of two produces equivalent counters, the question is how to tell rotated counters apart, or better, how to pick a standard form.

Most authors pick the form that is, in a loose sense, the "smallest" — the one starting with the smallest number. Thus, 2/2/3/1 is smaller than 3/1/2/2. This is easier to see if the slashes are removed: 2231 is smaller than 3122.

In the case of counters with more parts, the standard one can be obtained by forming all rotations by multiples of two and picking the smallest of the results.

The Number of Twill Counters

There are no real twill counters for 2 shafts, although 1/1, which corresponds to plain weave, is an acceptable twill counter structurally. We'll include it in what follows so that we don't have to make exceptions for it constantly.

For 3 shafts, there are two twill counters, 1/2 and 2/1. Four 4 shafts, there are four: 1/1/1/1, 1/2, 2/2, and 3/1, 1/1/1/1 is simply a doubling of the 2-shaft 1/1.

As the number of shafts increases, it becomes increasingly difficult to figure out all the possible twill counters, especially if doing it by hand. In fact, mistakes can be found in this regard in old weaving books. For example, Posselt's *Technology of Textile Design* [1] omits some of the twill counters for 6 and 8 shafts.

Whether working by hand or using a computer, a systematic method is needed. And, as the number of shafts gets large, the method needs to be efficient.

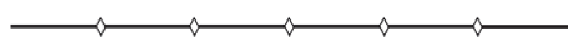
A Method for Determining Twill Counters

The sum of the numbers in a twill counter add up to the number of shafts being considered. For example, for 4 is the sum of smaller numbers in six ways: 1 + 1 + 1 + 1, 1 + 1 + 2, 1 + 2 + 1, 1 + 3, 2 + 2, and 3 + 1. Of these, only the ones with an even number of terms correspond to twill counters: 1 + 1 + 1 + 1, 1 + 3, 2 + 2, and 3 + 1.

In mathematics, expressing a number as the sum of smaller (positive) numbers is called a *composition* [2]. In compositions, as in twill counters, order matters: 1 + 3 is different from 3 + 1. (If order doesn't matter, so that 1 + 3 and 3 + 1 are considered to be the same, these are called *partitions*.)

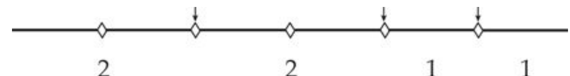
So the problem of finding twill counters is equivalent to the problem of finding compositions with even numbers of terms.

A convenient way to formulate the problem is to treat the number of shafts, n as a line with n segments of equal length, each segment amounting to one part of n. For $n = 6$, the line, with diamonds connecting the segments, looks like this:



Notice that there are five connection points, one less than the number of segments.

A composition can be obtained by selecting connection points. For example, the composition $2 + 2 + 1 + 1$ has the selected connection points as shown by the arrows:



Notice for *m* segments, only *m* – 1 connection points are selected.

The composition, looked at in this way, can be represented by a bit pattern in which 0 means a connection point is not selected, and 1 means a connection point is selected. Therefore, the composition in the example above has the bit pattern 01011.

Note: It you're not interested in mathematics, just skip the next few paragraphs. They are not important.

In general, for n shafts, there are $n - 1$ connection points, 2^{n-1} possible patterns, and $2^{n-1} - 1$ compositions (since an all-0 pattern, which represents n itself, is not considered to be a composition). The number of m-segment compositions of n is given by the binomial coefficient

$$\binom{n-1}{m-1} = \frac{(n-1)!}{(n-m)! \times (m-1)!}$$

For twill counters, compositions with an even number of parts (an odd number of connection points) are required. The number of these is given by

$$\sum_{m \text{ odd}, < n} \binom{n-1}{m-1}$$

For example, for $n = 6$, the number of twill counters is

Sidebar on on mathematical notation by Ralph.

$$\binom{5}{5} + \binom{5}{3} + \binom{5}{1} = 1 + 10 + 5 = 16$$

Some of these consist of repeats of counters for a smaller number of shafts.

In terms of bit patterns, only those with 5, 3, or 1 1s correspond to twill counters. There are few enough of these to list them here:

11111	01101
11100	01011
11010	00111
11001	10000
10110	01000
10101	00100
10011	00010
01110	00001

To convert binary pattern to a counter, proceed as follows:

1. Start at the left.
2. Remove 0s up to the first 1.
3. The corresponding counter number is the number of zeros removed + 1 (just 1 if there are no zeros).
4. Repeat steps 2 and 3 until there are no more 1s.
5. The last counter number is the number of 0s remaining plus 1 (just 1 if there are no remaining zeros).

As an example, consider the pattern 10110.

There are no zeros before the leftmost 1, so the first counter number is 1, and starting to construct the counter, we have 1/.

What remains is 0110. Now there is one zero before the first 1, so the next counter number is 2, and the evolving counter is 1/2.

What remains is 10. There are no 0s before the 1, so the next counter number is 1 and the evolving counter is 1/2/1.

All that remains is 0. Since there are no more 1s, the last counter number is 2 (one remaining 0 plus 1), and the complete twill counter is 1/2/1/2.

Note that if you just want counters with a specific number of counter numbers, you can use patterns that have that number of 1s less 1. For example, for 6-shaft counters with only four numbers (counters of the form $i/j/m/n$), you only need to decode

11100	10101

11010   10011
11001   01110
10110   00111

The method given here produces twill counters that are not in standard form that duplicate ones in standard form. For example,

00111 → 3/1/1/1
11100 → 1/1/1/3

Here the first twill counter is not in standard form but is a rotation of the second, which is in standard form.

It is necessary to remove twill counters that are not in standard form using the method described earlier. For 6 shafts, four of the counters are not in standard form, leaving a total of 12 twill counters:

/1/1/1/1/1/1
/1/1/1/3
/1/1/2/2
/1/1/3/1
/1/2/1/2
/1/2/2/1
/1/5
/2/1/2/1
/2/4
/3/3
/4/2
/5/1

Proper and Inherited Twill Counters

As mentioned above, some twill counters produced by this method may consist of repeats of twill counters for a smaller number of shafts. For example, 1/2/1/2 from the worked-out example above is a repeat of the 3-shaft twill counter 1/2.

Counters that are repeats of counters for a smaller number of shafts are called *inherited counters*. The rest are called *proper counters*.

There is nothing wrong with an inherited counter; it's just that it comes from a smaller number of shafts.

The number of inherited counters depends on the divisors of the number of shafts. For example, 6 has divisors 2 and 3. Thus, all of the (proper) counters for 2 shafts and 3 shafts are inherited for 6 shafts.

If a divisor itself has inherited counters, these are included in *its* divisors. For example, for 12 shafts, counters are inherited from 2-, 3-, 4-, and 6-shaft counters. The inherited 6-shaft counters are included in the proper 2- and 3-shaft counters, which are inherited for 12 shafts by virtue of its divisors 2 and 3.

How Many Twill Counters Are There?

The number of twill counters increases rapidly with the number of shafts. Here are the numbers through 20 shafts:

shafts	*proper*	*inherited*	*total*
2	1	0	1
3	2	0	2
4	3	1	4
5	6	0	6
6	9	3	12
7	18	0	18
8	30	4	34
9	56	2	58
10	99	7	106
11	186	0	186
12	338	12	350
13	630	0	630
14	1161	19	1180
15	2182	8	2190
16	4080	34	4114
17	7711	0	7711
18	14543	57	14600
19	27594	0	27594
20	52377	109	52486

Note that if the number of shafts is a prime, there are no inherited twill counters because there are no divisors.

For more than 8 or so shafts, there are so many twill counters that it doesn't make sense to list them all — how could they all be used?

Satin Counters

Few aspects of weaving require more mathematics than simple arithmetic. When a subject does require more, authors of books on weaving sometimes provide descriptions that are anything but clear, even to a person with some knowledge of mathematics. And sometimes the descriptions are incomplete or even incorrect.

Part of the problem is that the authors think they are writing for an audience most of whom not only know little mathematics beyond the most basic but also often are hostile to or fearsome of mathematics. (Hostility is a good cover for fear, the latter being socially less acceptable.) Of course, authors themselves may have the same problems with mathematics. Part of the problem comes from trying to express in words things for which ordinary language is inadequate.

How (or How Not) to Determine Satin Counters

Sidebar on what satin counters are by Ruth/Marg

Satin counters provide an example. Here are five quotations on the subject from sources dating from 1888 to 1994.

E. A. Posselt, author of many books on weaving and textiles in the late eighteenth and early nineteenth centuries, in 1882 writes [1]:

> *Divide the number of harness for the satin into two parts, which must neither be equal nor the one a multiple of the other; again it must not be possible to divide both parts by a third number.*

Harness is used here as a collective noun.

Charles Z. Petzold, writing in 1900, gives this (incomplete) rule [2]:

> *The mathematical formula is found by dividing the number of harnesses of the desired sateen into two parts. The numbers thus found should not be equal, neither should they be a multiple of each other.*

Ann Sutton, writing in 1982, describes determining satin counters in this way [3]:

> *Divide the number of ends (or shafts) on which the satin … is to be woven into two unequal parts, so that one shall not be a measure of the other, nor shall it be divisible by a common number.*

"Measure of the other" is British English and means (I think — my problem)

"divisible by the other".

S. A. Zielinski, in his massive *Master Weaver Library*, says [4]:

Find two numbers which give a sum equal to the number of frames. None of these numbers can be 1; the two numbers cannot divide one another, or by any other number at the same time.

Madelyn van der Hoogt, explains what a satin counter cannot be [5]:

The satin counter cannot be 1, or the interlacement forms a twill. It cannot be one fewer than the number in the unit … , or the interlacement forms a twill in the opposite direction. The counter cannot share a divisor with the number in the unit, or some warp threads interlace more than once and others not at all.

Grammatical errors, tortured prose, questionable meaning, missing parts, and definition by elimination aside, what does this all mean?

Mathematics to the Rescue (?)

Integer Division and Prime Numbers

The result of the division of two integers (whole numbers) such as 3 / 2 is usually taken to be 1-1/2 *[will fix typography]* or 1.5; that is 1 plus a fractional part. In true integer division, the fractional part is omitted and the result is just 1. If there is no fractional part, the second integer is said to evenly divide the first. For example, in 4 / 2, 2 evenly divides 4, and 2 is said to be a divisor of 4.

A prime number is one that has no divisors other than 1 and itself. An example is 7. The smallest prime is 2, the only even prime. The first few primes are 2, 3, 5, 7, 11, 13, 17, … The is no limit to the number of primes. This is sometimes stated as "the number of primes is infinite".

Two integers are relatively prime if they have no common divisor. For example, 5 and 12 are relatively prime, but 4 and 12 are not, since 4 divides both 4 and 12.

A mathematician might state it this way:

For n shafts, find relatively prime i and j such that $1 < i < n$, i

$+ j = n$, and $1 < j < n$.

The mathematician talks in terms of "variables"— i, j, and n — and describes the conditions they must satisfy. The condition "$1 < i < n$" can be phrased in plain English as "i is greater than 1 and less than n" and the phrase "$i + j = n$" as "i and j add to n" In fact, it might seem clearer as "$j = n - i$", but the mathematician prefers to talk in terms of constraints (requirements), for which "$i + j = n$" is appropriate.

All that having been said, most readers would understand the mathematical statement above except that they probably would not know what "relatively prime" means, which is the conceptual core of the definition.

There are many variations on the so-called mathematical description. (Actually, mathematicians are capable of and sometimes take delight in using arcane symbols and convoluted prose to make what really is simple into something that is incomprehensible to the layperson and requires effort even for other mathematicians to understand [6].)

It is possible to provide a definition of satin counter in simple mathematical terms along with an explanation of the core concept that is intelligible to most readers. Simple examples, especially of things that work, as opposed to those that do not, are a great help in understanding such things. Unfortunately, many mathematicians consider it beneath them to do this in their writing. A famous computer scientist openly stated that the use of examples is a sign of intellectual weakness.

A Table of Satin Counters

Having spent more than a page on the problems with describing satin counters, I'll finish by giving a table.

In a sense, a table aren't that bad: there are not that many different counters for the number of shafts that are available for hand looms and there's no need for a weaver to compute them.

Here is a table for 2 to 24 shafts. Only the smaller of the two counter pairs is given; the other is easy to determine by simple subtraction. For example, for 13 shafts and the small counter of 4, the large counter is $13 - 4 = 9$. Incidentally, most descriptions of satin counters say that the smaller is preferable, or at least more often used, without saying why. Indeed, why?

shafts	*small counters*	*number*
2		0
3		0
4		0
5	2	1
6		0

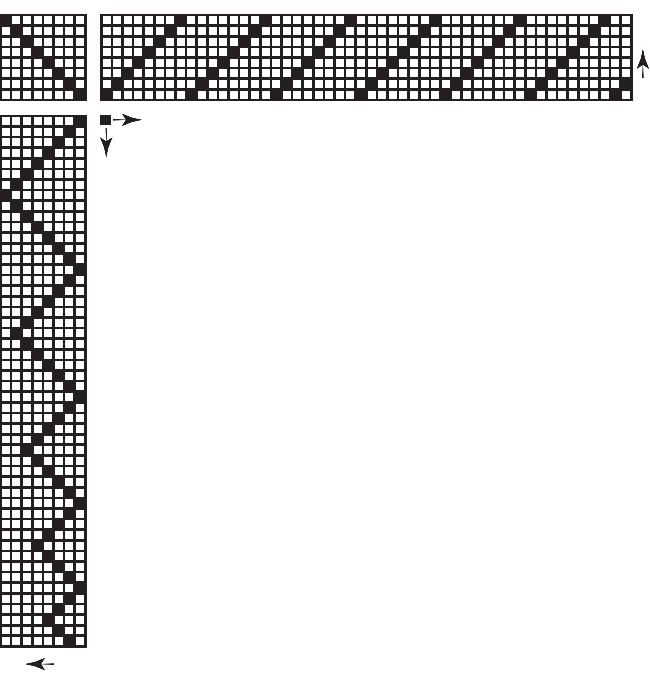
7	2 3	2
8	3	1
9	2 4	2
10	3	1
11	2 3 4 5	4
12	5	1
13	2 3 4 5 6	5
14	3 5	2
15	2 4 7	3
16	3 5 7	3
17	2 3 4 5 6 7 8	7
18	5 7	2
19	2 3 4 5 6 7 8 9	8
20	3 7 9	3
21	2 4 5 8 10	5
22	3 5 7 9	4
23	2 3 4 5 6 7 8 9 10 11	10
24	5 7 11	3

As is well known to weavers, true satin requires at least five shafts and cannot be woven with six shafts. By the way, if the number of shafts is a prime, $p > 2$, any number $2 \le i \le p - 1$ is a valid counter: If $i + j = p$, i and j must be relatively prime — otherwise a common factor would divide p.

06 Mar 2006 D2: SatinCounters

Drafting With Sequences

Shafts and treadles in drafts are numbered for identification. The numbers of the shafts through which successive warp threads pass form a sequence, as do the numbers of the treadles for successive picks. Consider the following draft, in which the arrows indicate the orientation:



The threading is an upward straight draw. The sequence is:

1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8,
1, 2

The treadling sequence is more complicated:

1, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4, 3, 2, 1, 2, 3, 4, 5, 6, 7, 6, 5, 4,
3, 2, 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, 4,
3, 2

These two sequences, in combination with the tie-up, define the structure of the weave.

Threading and treadling sequences often have distinctive patterns, as in the

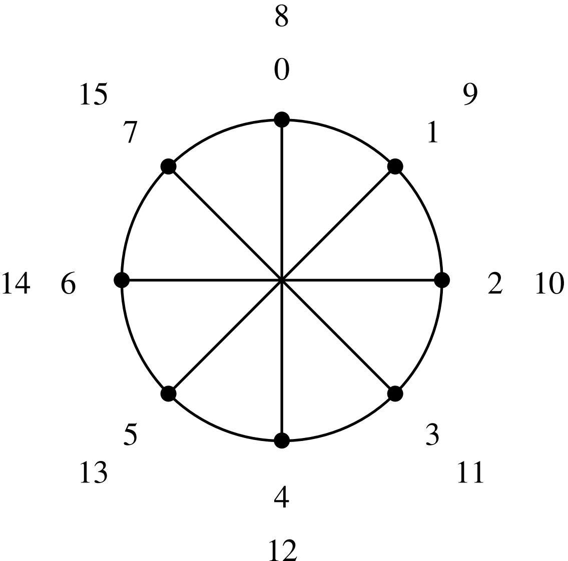
05 Jul 2006 D3: SequenceDrafting

repeat for the threading sequence above. In the case of a repeat, it's only necessary to know the basic unit, which is indicated by brackets:

[1, 2, 3, 4, 5, 6, 7, 8]

Modular Arithmetic

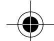
Since looms have a fixed number of shafts and treadles, the sequences usually are most easily understood in terms of modular arithmetic, sometimes called clock or wheel arithmetic, in which numbers go around a circle clockwise, starting with 0. If there are 8 shafts, there are 8 equally spaced points on the circle from 0 to 7:

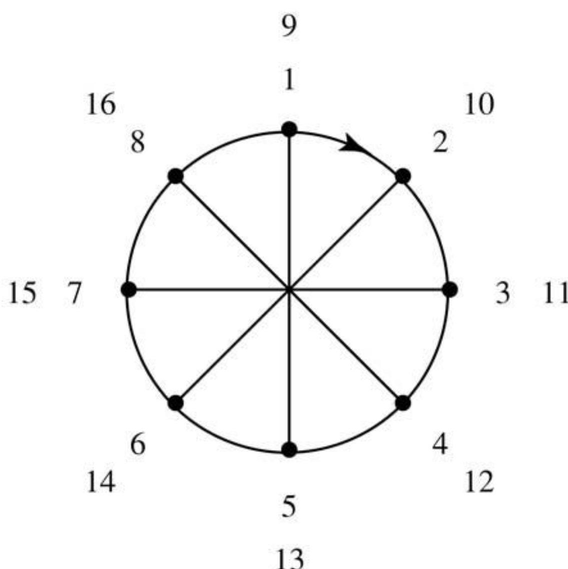


The numbers on the inner circle are those that exist in the modular arithmetic. Continuing beyond 7, as shown in the outer ring, the numbers wrap around the wheel. Numbers on the same spoke are equivalent. For example, 0 and 8 are equivalent, 1 and 9 are equivalent, 2 and 10 are equivalent, and so on. Another way to look at it is that when 9 is introduced into modular arithmetic with 8 shafts, it *becomes* 1, and so on.

Shaft Arithmetic

Although modular arithmetic uses the number 0 as a starting point, most persons count from 1. This is used for numbering shafts and treadles and can easily accomplished by rotating the wheel counterclockwise by one position:





Notice that 1 and 9 are still equivalent, as are 2 and 10, and so on. If there are 8 shafts, there are 8 positive numbers. 0 has gone away, but it will be back.

For sequences, shafts and treadles are handled the same way, so it is called *shaft arithmetic*, with the understanding that it applies to treadles also. Of course, most facts about shaft arithmetic hold for ordinary modular arithmetic.

In shaft arithmetic, an upward straight draw for 8 shafts is described by the positive integers in sequence:

1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, …

and wrapped around the shaft circle to produce

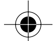
1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 4, 5, 6, 7, 8, …

The point is that an upward straight draw comes from the most fundamental of all integer sequences, the positive integers in order.

Drafting with Sequences

The idea behind drafting with sequences is that many sequences have interesting patterns, which often become more interesting in shaft arithmetic. In fact, many sequences show repeats when cast in shaft arithmetic. When this is the case, the entire sequence can be represented by the repeat. For example, the shaft sequence for an upward straight draw for 8 and 10 shafts are represented by

[1, 2, 3, 4, 5, 6, 7, 8]



and

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

respectively.

Note: Not all sequences produce repeats in shaft arithmetic. For example, the prime numbers, which are divisible only by 1 and themselves, do not show a repeat in shaft arithmetic (or in any other arithmetic).

Patterns in Sequences

Sequences may produce interesting woven patterns when they are used for threading and treadling.

There are many, many well-known integer sequences. The Fibonacci sequence, which has many connections in nature, design and mathematics, is one of the best known and most thoroughly studied of all integer sequences. The Fibonacci sequence starts with 1 and 1. Then each successive number (*term*) is the sum of the preceding two:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,…

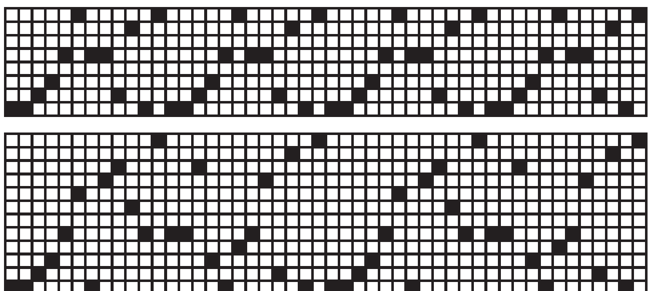
As the sequence continues, the numbers get very large. For example, the 50th term in the Fibonacci sequence is more than 12 billion. Shaft arithmetic brings this sequence under control. For 8 shafts, the result is

1, 1, 2, 3, 5, 8, 5, 5, 2, 7, 1, 8, 1, 1, 2, 3, 5, 8, 5, 5, 2, 7,
1, 8, 1, 1, 2, 3, 5, 8, 5, 5, 2, 7, 1, 8, …

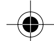
There is a repeat, so the entire sequence can be represented by

[1, 1, 2, 3, 5, 8, 5, 5, 2, 7, 1, 8]

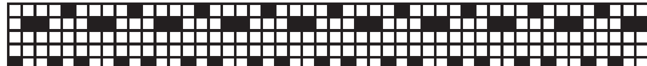
Patterns in sequences are more easily seen if they are plotted, as in the grids used in weaving drafts. For 8 and 12 shafts, the Fibonacci sequence looks like this:



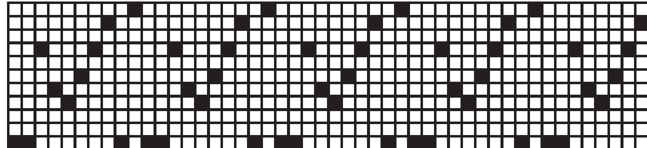
Here are some other simple sequences and what they look like for various numbers of shafts.



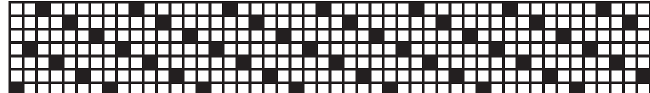
The squares for 5 shafts:



The cubes of the Fibonacci numbers for 11 shafts:

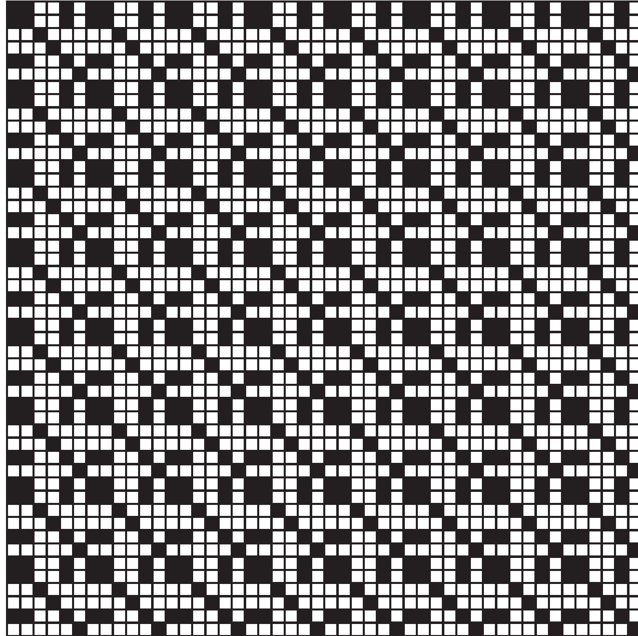


Every third positive integer for 7 shafts:

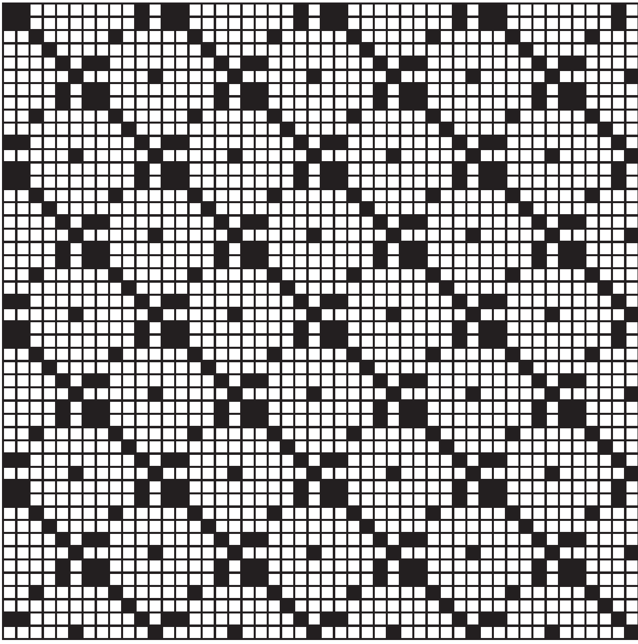


The patterns such sequences produce in weaves depend on many factors. To keep things simple to begin with, direct tie-ups and treadling as drawn in (that is, the same sequence for the threading and the treadling) are used. Even in this very limited framework, interesting woven patterns abound.

Here is a drawdown for a few repeats of the Fibonacci sequence for 4 shafts.



The pattern looks quite different for 8 shafts, although there are structures in common:

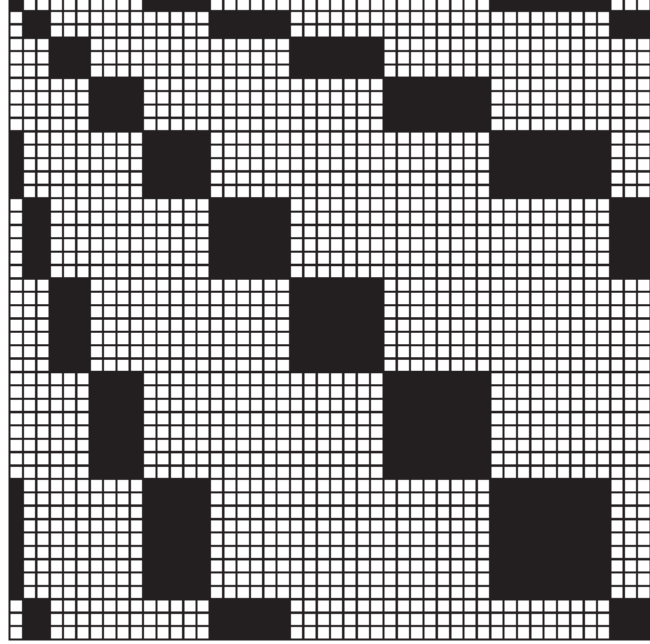


A simple sequence that produces interesting patterns is the "multi" sequence, which starts with a single 1 and is followed by 2 copies of 2, 3 copies of 3, and so on:

1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 6, …

Note that there are no repeats in shaft arithmetic for this sequence, since the "width" of the repeated integer blocks constantly increases.

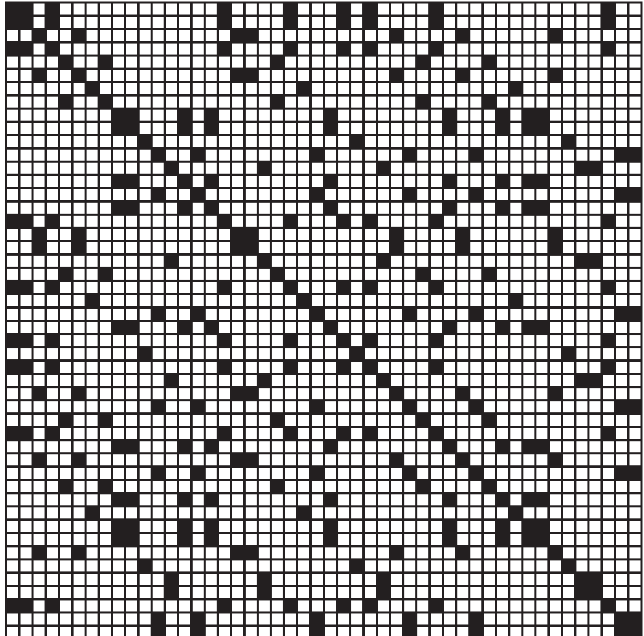
The drawdown for the multi sequence for 4 shafts is:



One way to produce interesting sequences is to combine other sequences, such as interleaving the terms of two sequences. For example, interleaving the positive integers and the Fibonacci sequence produces

1, 1, 2, 1, 3, 2, 4, 3, 5, 5, 6, 8, 7, 5, 8, 5, 1, 2, 2, 7, 3, 1, 4,
8, 5, 1, 6, 1, 7, 2, 8, 3, 1, 5, 2, 8, 3, 5, 4, 5, 5, 2, 6, 7, 7, 1,
8, 8 …

The drawdown for 8 shafts is:



Other tie-ups and threading sequences and treadling sequences that are different produce all kinds of interesting results.

Creating interesting weaves by drafting with sequences requires judicious selection and combination of sequences, the number of shafts and treadles, and tie-ups. An understanding of the properties of the sequences used may help, but a little luck and some experimentation also can lead to pleasant surprises. The process is a nice combination of artistic sense, creative talent, a modicum of arithmetic, and finding the hidden structures that abound in integer sequences.

Finding Interesting Integer Sequences

Interesting integer sequences can come from many sources. It helps if you have a computer with a program that can do simple arithmetic so that you can invent your own. There also are many on-line sources of sequences. By far the most extensive one is the "Encyclopedia of Integer Sequences" (EIS) [1]

Beware, though — this site contains a lot of esoteric mathematical material and its vastness can be overwhelming. It's like a "Haystack from Hell", but the needles to be found within are made of precious metals.

Getting Shaft Sequences

There are shaft sequences for a few integer sequences and various numbers of shafts on Reference 2.

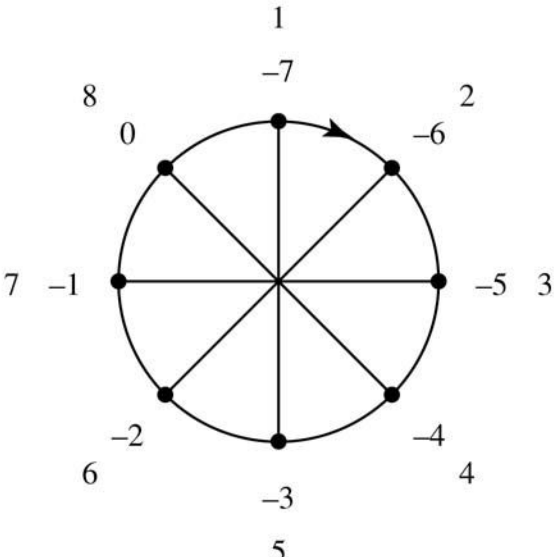
These sequences provide an easy way to start, but you'll want more if you decide you're really interested in drafting with sequences.

You can find many integer sequences ready made, but in order to do your own drafting, you need to be able to convert them to shaft sequences for different numbers of shafts. The method is simple: Divide each term by the number of shafts and take the reminder. For example, for 8 shafts, the reminder of 13 divided by 8 is 5, which is the shaft number for 13. That gives you the corresponding term in the shaft sequence. It helps if you have a program or calculator that can do integer arithmetic and produce remainders.

There's one more complication — 0 and negative numbers. The way to deal with these is indicated by looking at what happens when you have negative integers in increasing sequence as they cross over to the positive integers:

…, –7, –6, –5, –5, –3, –2, –1, 0, 1, 2, 3, 4, 5, 6, 7, 8, …

Now think of the modular wheel and what happens if you wrap this sequence of numbers around it. For 8 shafts, it looks like this:



In other words, –1 becomes 7, –2 becomes 6, and so on. Note that 0, which haas been been hiding, becomes 8.

Perhaps you now see the integer sequence that produces a downward straight draw:

0, –1, –2, –3, –4, –5, –6, –7, –8, –9, –10, …

All that's needed to convert a non-positive remainder to a shaft number is to add it to the number of shafts. For –1, for example,

$8 + (-1) = 7$

Despite this long-winded discussion, getting shaft sequences from integer sequences is not difficult at all.

The interesting part remains — trying it and designing drafts.

Straight-Draw Threading Conversion

An earlier version of this article appeared in Complex Weavers Journal, No. 74, January 2004.

Views of the Problem

Suppose you have a straight-draw warp on *n* shafts (for example, 8) and want to use it to weave patterns designed for fewer shafts (say 4). The crucial point was that the pattern must come from a draft with an *n*-end repeat (say 8).

Observation: The pattern for *any* draft with a threading that has an *n*-end repeat can be woven with a straight-draw threading on *n* shafts.

For example, given an 8-shaft straight-draw threading (which has an 8-end repeat), patterns with drafts for 2 through 8 shafts (not just 4) can be woven, provided the drafts have 8-end threading repeats. (Actually, this applies to more than 8 shafts, although if a draft has more than 8 shafts and has an 8-end threading repeat, not all the shafts are used.) Notice that any 8-shaft draft with an 8-end threading repeat can be woven with a straight-draw threading; it's just a matter of rearranging the rows in the tie-up.

The problem can be turned around. Instead of assuming a straight-draw threading on *n* shafts and looking for drafts with *n*-end threading repeats to convert, consider the problem of converting drafts to drafts with straight-draw threadings without knowing, *a priori*, how many shafts would be required.

The two views are, of course, equivalent. It is just easier to formulate the problem by starting with a draft to convert.

Note that *any* draft can be converted to a draft with a straight-draw threading — the problem is that if the repeat is not small (or if there is no repeat), the number of shafts required may be impossibly large.

A Procedure

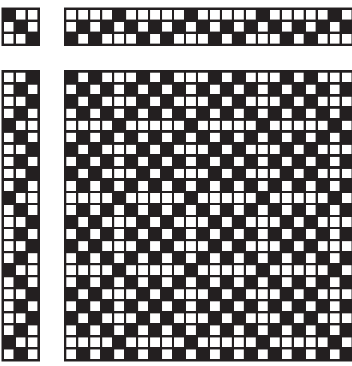
The first thing to do is to determine the repeat in the threading of the draft to be converted (the *original draft*). This often can be done by inspection, but care is needed to make sure the smallest repeat is found. Short of inspection, increasingly longer initial parts of the threading can be tried to see if, when repeated, they match the whole threading.

Suppose the repeat has length *n*. Then the new draft will have *n* shafts and a straight-draw threading. All that's left is to get the tie-up for the new draft (the treadling is the same as for the old draft).

Getting the tie-up is easy. Just start at the beginning of the threading for the original draft and add the corresponding row of its tie-up to the new tie-up, continuing end-by-end through the repeat.

An Example

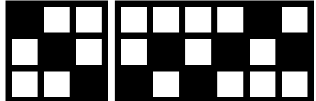
Here's a simple example: a 3-shaft draft with a threading repeat of length 6.



Original Draft

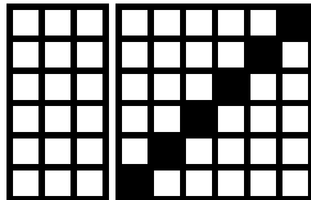
The new draft will, of course, have 6 shafts, so start with a blank 6-row tie-up.

Putting the original tie-up and the threading repeat next to each other helps in visualizing the process:



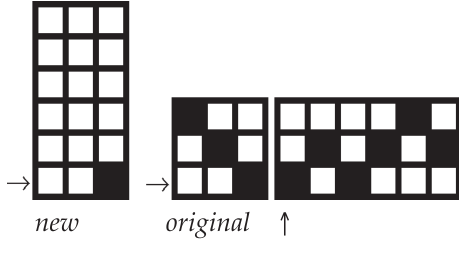
Original Tie-Up and Repeat

The new tie-up with the straight-draw repeat looks like this:



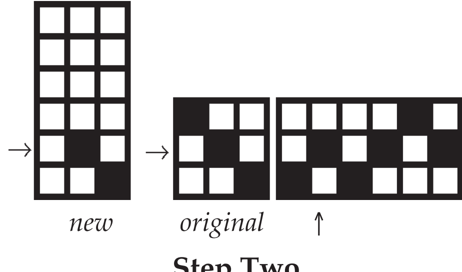
The Initial Setup

Start at the beginning of the repeat, looking at the first end. It is on shaft 1, so copy row 1 of the original tie-up to row 1 of the new tie-up:



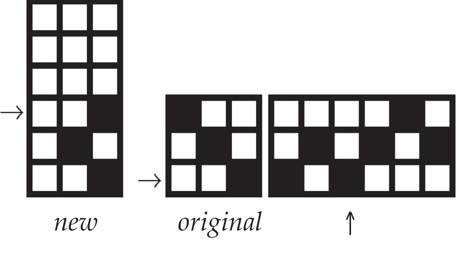
Step One

Now we go on to the second end of the repeat. It is on shaft 2, so copy the second row of the original tie-up to the next row of the new tie-up:



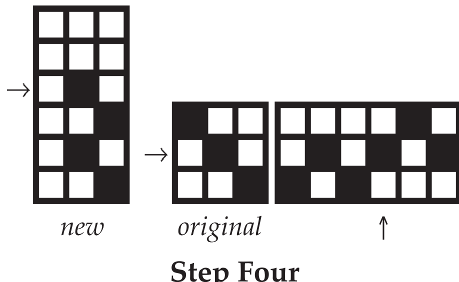
Step Two

The third end is on shaft 1, so copy the first row of the original tie-up to the next row of the new tie-up:



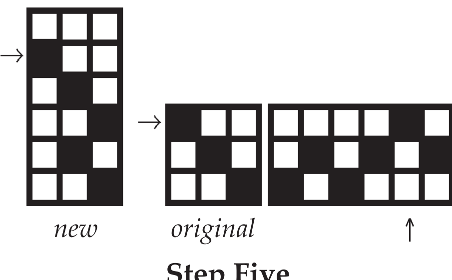
Step Three

End 4 is on shaft 2, so copy row 2 of the original tie-up to the next row of the new tie-up:



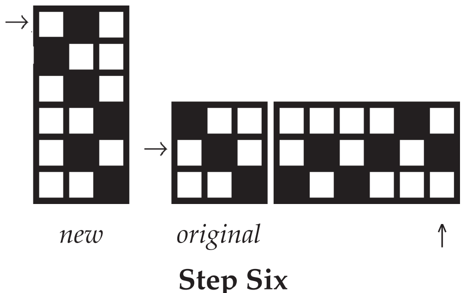
Step Four

We're getting there. For end 5, we copy row 3 of the original tie-up to the next row of the new-tie-up:



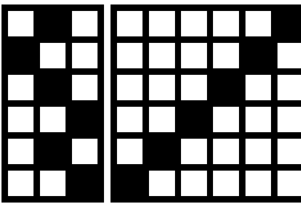
Step Five

To complete the process, we copy row 2 of the original tie-up to the top row of the new tie-up:



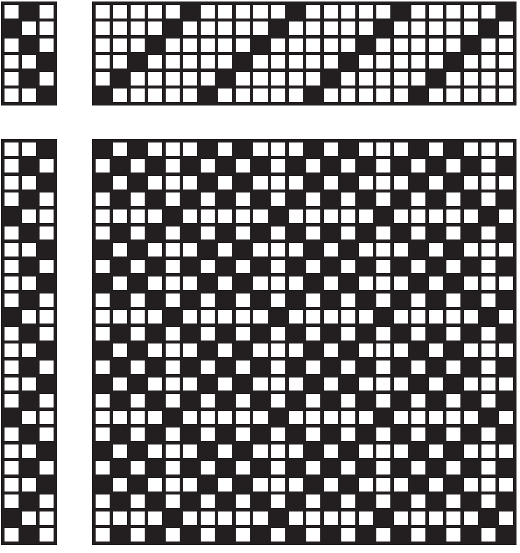
Step Six

Here's what the new tie-up and threading repeat look like:



New Tie-Up

The new draft looks like this:

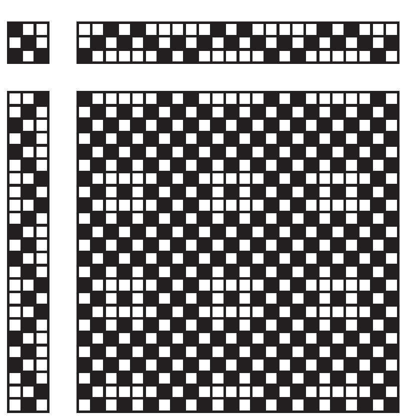


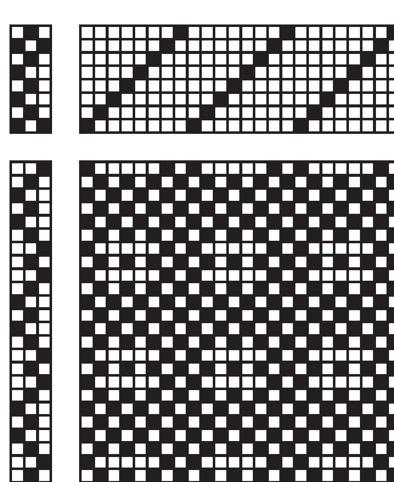
New Draft

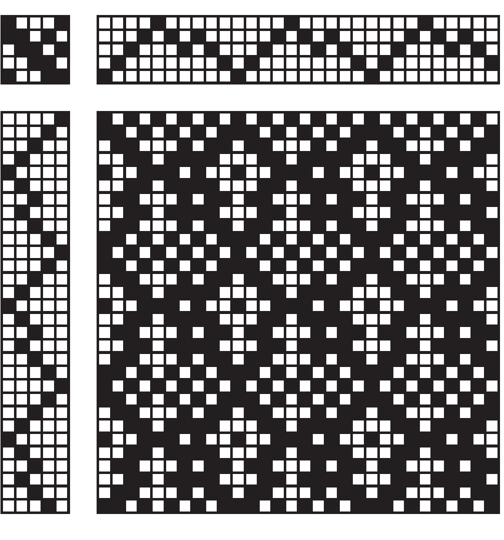
The Appendix contains examples of the conversion of several other drafts.

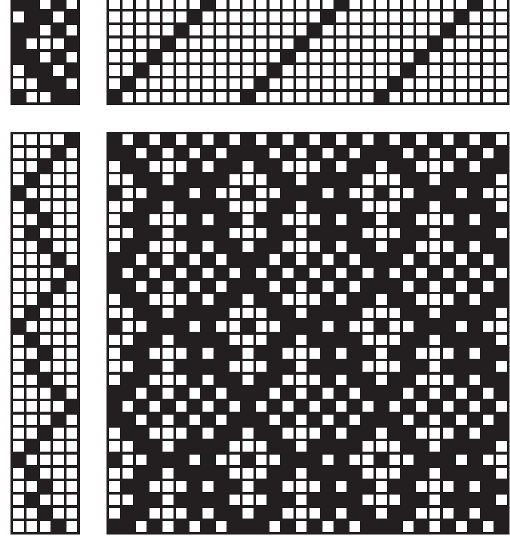
Conversion Examples

original *new*









Fabric Analysis —From Drawdown to Draft

Nature uses only the longest threads to weave her patterns, so each small piece of her fabric reveals the organization of the entire tapestry.

— Richard Feynman

Fabric analysis is the process of determining how a farbic was woven. The first steps are:

- determining the interlacement of the warp and weft threads
- producing a draft from the interlacement

A particular method for doing fabric analysis may intermix these two processes, constructing the draft as the interlacement is determined. The two processes can be done separately, however, and there are advantages to separation:

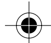
- Determining the interlacement can be difficult. It requires a knowledge of weaving and careful visual examination of the fabric.
- Producing a draft from the interlacement is a mechanical task of an entirely different nature. It can be done by a person who is unfamiliar with weaving or by a computer program. A computer program is, of course, fast, but it offers a more important advantage: accuracy.

This section describes a method of going from a drawdown to a draft and shows how it can be done by a program.

Drawdowns

Various systems of notation are used for describing interlacement, but for the purposes of producing a draft, they are equivalent. The conventional drawdown grid, in which black squares indicate intersections where warp threads are on top and the white squares where weft threads are on top, is most widely used. Figure Ω.1 shows an example.

Such a drawdown is designed to make it easy for a person to see the interlacement (and any patterns that may exist in it). For a computer program, a drawdown is just a rectangular array of zeros and ones with ones indicating where warp threads are on top and zeros indicating where weft threads are on top, as shown in Figure Ω.2.

05 Jul 2006  D5: FabricAnalysis

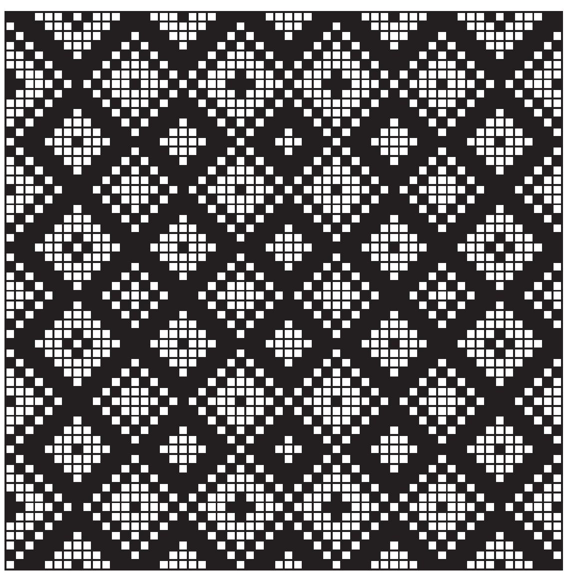
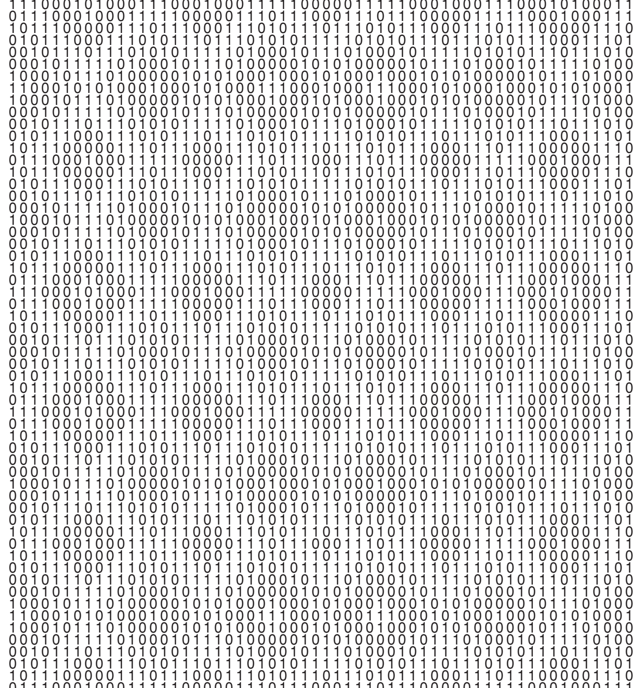


Figure Ω.1. A Drawdown



```
1110001010001110001000111110000011111000100011100010100011
0111000100011111000001110111000111011100000111110001000111
1011100000111011100011101011101110101110001110111000001110
0101110001110101110111010101111101010111011101011100011101
0010111011101010111110100010111010001011111010101110111010
0001011111010001011101000001010100000101110100010111110100
1000101110100000101010001000101000100010101000001011101000
1100010101000100010100011100010001110001010001000101010001
1000101110100000101010001000101000100010101000001011101000
0001011111010001011101000001010100000101110100010111110100
0010111011101010111110100010111010001011111010101110111010
0101110001110101110111010101111101010111011101011100011101
1011100000111011100011101011101110101110001110111000001110
0111000100011111000001110111000111011100000111110001000111
1011100000111011100011101011101110101110001110111000001110
0101110001110101110111010101111101010111011101011100011101
0010111011101010111110100010111010001011111010101110111010
0001011111010001011101000001010100000101110100010111110100
1000101110100000101010001000101000100010101000001011101000
0001011111010001011101000001010100000101110100010111110100
0010111011101010111110100010111010001011111010101110111010
0101110001110101110111010101111101010111011101011100011101
1011100000111011100011101011101110101110001110111000001110
0111000100011111000001110111000111011100000111110001000111
1110001010001110001000111110000011111000100011100010100011
0111000100011111000001110111000111011100000111110001000111
1011100000111011100011101011101110101110001110111000001110
0101110001110101110111010101111101010111011101011100011101
0010111011101010111110100010111010001011111010101110111010
0001011111010001011101000001010100000101110100010111110100
0010111011101010111110100010111010001011111010101110111010
0101110001110101110111010101111101010111011101011100011101
1011100000111011100011101011101110101110001110111000001110
0111000100011111000001110111000111011100000111110001000111
1110001010001110001000111110000011111000100011100010100011
0111000100011111000001110111000111011100000111110001000111
1011100000111011100011101011101110101110001110111000001110
0101110001110101110111010101111101010111011101011100011101
0010111011101010111110100010111010001011111010101110111010
0001011111010001011101000001010100000101110100010111110100
1000101110100000101010001000101000100010101000001011101000
0001011111010001011101000001010100000101110100010111110100
0010111011101010111110100010111010001011111010101110111010
0101110001110101110111010101111101010111011101011100011101
1011100000111011100011101011101110101110001110111000001110
0111000100011111000001110111000111011100000111110001000111
1011100000111011100011101011101110101110001110111000001110
0101110001110101110111010101111101010111011101011100011101
0010111011101010111110100010111010001011111010101110111010
0001011111010001011101000001010100000101110100010111110100
1000101110100000101010001000101000100010101000001011101000
1100010101000100010100011100010001110001010001000101010001
1000101110100000101010001000101000100010101000001011101000
0001011111010001011101000001010100000101110100010111110100
0010111011101010111110100010111010001011111010101110111010
0101110001110101110111010101111101010111011101011100011101
1011100000111011100011101011101110101110001110111000001110
0111000100011111000001110111000111011100000111110001000111
```

Figure Ω.2. Drawdown Data

Observations

The number of treadles required is the number of *different* row patterns. The number of shafts required is the number of *different* column patterns.

In a large, complicated drawdown, it's difficult for a human being to determine the different row and column patterns. That's where a computer program comes in.

Comment: You can rearrange the rows and columns of a drawdown or delete duplicates without affecting the number of treadles and shafts required.

The Process

Creating a draft from a drawdown is a three-step process. The first two steps, which are central to the approach used here, can be done in either order.

The first step identifies the different rows and assigns a treadle number to each. The sequence of treadle numbers for the rows in the drawdown gives the treadling sequence.

The second step does the same thing for the columns to assign shaft numbers to the different columns and produce the threading sequence.

Comment: For the purposes of getting a workable draft, it doesn't matter which treadles and shafts are assigned to the different rows and columns. A systematic method, however, such as working from right to left or left to right, usually produces a better-organized draft.

The third step is to produce the tie-up that relates the treadles and shafts according to the drawdown.

To see how the first two steps might go by hand, look at Figure Ω.3, which shows treadle and shaft assignments to the upper-left portion of the drawdown we've been considering.

```
                 shafts
treadles   1 2 3 4 5 6 7 8 7 6 ...

    1      1 1 1 0 0 0 1 0 1 0 0 0 1  ...
    2      0 1 1 1 0 0 0 1 0 0 0 1 1  ...
    3      1 0 1 1 1 0 0 0 0 0 1 1 1  ...
    4      0 1 0 1 1 1 0 0 0 1 1 1 0  ...
    5      0 0 1 0 1 1 1 0 1 1 1 0 1  ...
    6      0 0 0 1 0 1 1 1 1 1 0 1 0  ...
    7      1 0 0 0 1 0 1 1 1 0 1 0 0  ...
    8      1 1 0 0 0 1 0 1 0 1 0 0 0  ...
    7      1 0 0 0 1 0 1 1 1 0 1 0 0  ...
    6      0 0 0 1 0 1 1 1 1 1 0 1 0  ...
   ...                ...
```

Figure Ω.3. Treadle and Shaft Assignments

It is easy to see that the beginnings of the first eight rows are different and

therefore they are different patterns and each gets a separate treadle. The ninth row starts out like the seventh row and looking at the complete drawdown in Figure Ω.1, they are the same. Similarly, the tenth row is the same as the sixth. The same is true of the columns.

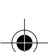
In this example, there are only eight different row patterns and eight different column patterns. They can be distinguished by the first three digits they contain. But another drawdown might be more complex, irregular, and not so easily analyzed by hand.

Programs

In this section, we'll show programs for determining the loom resources required and producing a draft from a drawdown.

You don't need to be a programmer to get an idea of what's going on. If you're not a programmer, just read through what follows and ignore the details.

How easy it is to write program to analyze a drawdown depends to a considerable extent on the programming language used. In our examples here, we'll use Icon [1, 2], a high-level programming language designed for the manipulation of strings of characters (like row and column patterns) and structures (like lists of patterns).



If you're a programmer but not familiar with Icon, just browse through what follows, note the comments, and imagine how you'd do it in your favorite programming language.

First, a word about strings and data structures in Icon. Strings are sequences of characters. A data structure is a collection of values that are organized in a particular way. We'll use three kinds of data structures in the programs that follow:

A *list* is a sequence of values. The values may be strings or other types of values.

A *set* is an unordered collection of unique values. A set can be created from a list; any duplicate values in the list are discarded.

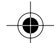
A *table* is like a set, except that it has unique keys with which values can be associated.

Strings, lists, sets, and tables are created and modified as a program runs.

Computing Treadle and Shaft Requirements

Here's a little program that reads a drawdown in the form of rows of zeros and ones such as shown in Figure Ω.2. It writes out the number of treadles and shafts required.

```
procedure main()
```



```
  # Read the drawdown and put it in a list.
  rows := []                              # empty list to start
  while put(rows, read())           # add the row patterns
  # Write the number of treadles needed.
  write(*set(rows), " treadles needed" )
  # Rotate the drawdown 90 degrees to put the
  # columns in the place of rows.
  cols := rotate(rows)
  # Write the number of shafts needed.
  write(*set(cols), " shafts needed" )
end
```

The conversion of the list rows to a set, using set(rows), creates a set of rows without the duplicates. The operation * produces the number of members in the set. The procedure rotate() is shown in the complete program listing in Appendix A. The number of shafts required is then determined in the same way as the number of treadles.

The output of the program for the data shown in Figure Ω.2 is

```
8 shafts needed
8 treadles needed
```

Comment: A drawdown must, of course contain at least one full repeat. If it contains more, the extra rows and columns are just duplicates and do not affect the result.

Producing a Draft

To produce a draft, a little more work is required. Here's a program.

```
procedure main()
  # Read the drawdown and put it in a list.
  rows := []                              # empty list to start
  while put(rows, read())           # add the row patterns
  cols := rotate(rows)                # list of columns
  # Compute the treadling sequence.
  number := 0                             # treadle counter
```

```
treadles := table()                          # table of row patterns

# Build a table of the different row patterns and
# assign a shaft number to each.

every treadles[!set(rows)] := (number +:= 1)

# Compute the threading sequence the same way.

shafts := table()
number := 0

every shafts[!set(cols)] := (number +:= 1)

# Create the tie–up.

tieup := table()

every row := key(treadles) do {
      tie_line := repl("0", *shafts) # no ties to start
   every i := 1 to *row do           # go through row
     if row[i] == "1" then           # tie if warp on top
       tie_line[threading[i]] := "1" # for rising shed
   tieup[treadles[row]] := tie_line  # add tie-up line
   }

# Write the treadling sequence.

write("Treadling sequence:")

every writes(treadles[!rows], " ")
write()

# Write the threading sequence.

write("Threadling sequence:")

every writes(shafts[!cols], " ")
write()

# Write the tie–up.

  every i := 1 to *treadles do
   write(tieup[i])

end
```

The operator ! generates the members of a set. Each one becomes a key in a table, with which a number is associated (+:= 1 increments the value of number). To produce the sequences, the rows and columns are generated and used as keys to the tables, which in turn produces the numbers.

The tie-up is created using a table. key(treadles) produces the row patterns. Each line of the tie-up starts with zeros, indicating the absence of ties. Then for every position of the row pattern that is one, the position in the row of the corresponding shaft is set to one to indicate a tie.

The sequences are produced from left to right, since that is the most natural and easiest way to program the process.

The output for the data shown in Figure Ω.2 is:

```
Treadling sequence:
1 2 3 4 5 6 7 8 7 6 5 4 3 2 3 4 5 6 7 6 5 4 3 2 1 2 3 4 5 6 5 4 3 2 1 2 3 4 5 6 7 6 5 4
3 2 3 4 5 6 7 8 7 6 5 4 3 2
Threading sequence:
1 2 3 4 5 6 7 8 7 6 5 4 3 2 3 4 5 6 7 6 5 4 3 2 1 2 3 4 5 6 5 4 3 2 1 2 3 4 5 6 7 6 5 4
3 2 3 4 5 6 7 8 7 6 5 4 3 2
Tie-up:
11100010
01110001
10111000
01011100
00101110
00010111
10001011
11000101
```

Note that the treadling and threading sequences are the same: The draft is treadled as drawn.

Figure Ω.4 shows a conventional draft produced from the results given by the program above.

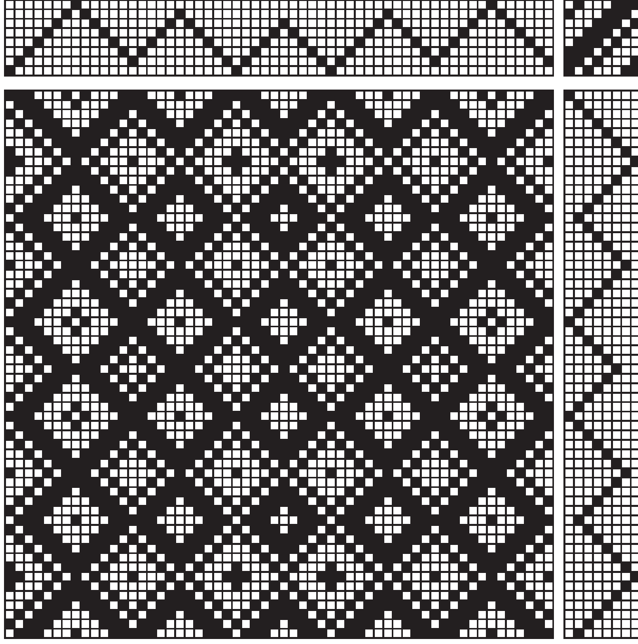


Figure Ω.4. The Draft

Related Issues

WIFs

The program that produces the draft information can be modified to produce a WIF [3], which can be imported into a weaving program and exchanged with other weavers.

This is not difficult to do, but WIF is a verbose format and the code needed to produce a WIF is considerably longer than the code to analyze the drawdown.

The Appendix A shows the complete program for going from the raw data to a WIF. Appendix B shows the WIF for the data shown in Figure Ω.2.

Drafts from Images

Any black and white image* can be considered to be a drawdown — the black pixels (individual dots) correspond to where imaginary warp threads are on top and the white pixels correspond to where imaginary weft threads are on top.

To get a draft for weaving the equivalent of such an image, it only is necessary to convert the pixels in the image to patterns of zeroes and ones as shown in Figure Ω.2.

This requires a little bit of computer graphics. In Icon, it looks like this [4]:

```
WOpen("image=design.gif")                # window for image

width := WAttrib("width")                # dimensions
height := WAttrib("height")

rows := []

# Get the row patterns

every y := 0 to height – 1 do {
  row := ""
  every p := Pixel(0, y, width, 1) do
    if ColorValue(p) == "0,0,0" then row ||:= "1"
      else row ||:= "0"
  put(rows, row)
  }
```

The value "0,0,0" corresponds to a black pixel. Other pixels are assumed to be white. The rest of the program is the same as before.

Comment: Some weaving programs, such as SwiftWeave [5], provide a facility for going from black-and-white images to drafts. SwiftWeave calls it a drawup.

A draft created from an image can be used as a profile draft or a threading draft. For a threading draft, modifications may be needed to make it weavable.

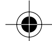
Although any two-color pattern can be converted to a draft, the problem is the loom resources required — the number of shafts and treadles — which exceed the capacity of your loom — or any loom (recall that the number of treadles required is the number of different rows and the number of shafts required is the number of different columns).

If you want to try this, pick images that are small, with straight lines, and lots of duplicate rows and columns. Horizontal and vertical symmetry come free.

Appendix C shows an image and the drawdown produced by this method.

On-Line Resources

The programs described here, along with sample data and images, are available on the Web [6].



Name Drafting

Many handweavers simply weave from the large number of drafts that are available in books and magazines about weaving. These weavers may make minor modifications, but the designs they weave are the creations of others. The measure of "real" handweavers is the desire and ability to create their own designs. But how to start?

A type of weaving known as *name drafting* often is recommended for this situation. (Name drafting also is known as code drafting, commemorative drafting, and personalized design.)

Although name drafting is naive in concept, it does provide an easy bridge between copying the work of others and creating new designs.

Mapping Strings into Threading Sequences

The basic idea is simple: A string of characters — a word, or more often, a phrase or sentence — is named to make a threading sequence. The string may be the name of a loved one or a famous person (hence the term name drafting), a motto, an epigram, or anything else that strikes a weaver's fancy.

The coding assigns a shaft number to each character of the selected string. Although any method of associating shafts with characters could be used, only a few appear in the literature [1-7] and weavers generally are instructed to use one of these.

Three codings commonly are used for four shafts:

letters	*shaft*	
ABCDEFG	1	Table 1
HIJKLMN2		
OPQRSTU	3	
VWXYZ	4	

ABCDEF 1	Table 2	
GHIJKL	2	
MNOPQR	3	
STUVWXYZ	4	

AEIMQUY	1	Table 3
BFJNRVZ	2	
CGKOSW	3	
DHLPTX	4	

Suppose the string chosen is

JACOB ANGSTADT

and Table 1 is used. The resulting sequence is

2, 1, 1, 3, 1, 1, 2, 1, 3, 3, 1, 1, 3

If Table 2 is used, the sequence is

2, 1, 1, 3, 1, 1, 3, 2, 4, 4, 1, 1, 4

and if Table 3 is used, the sequence is

2, 1, 3, 3, 2, 1, 2, 3, 3, 4, 1, 4, 4

Note that the blank between Jacob and Angstadt is ignored; more on this later.

One problem in choosing a mapping between characters and shaft numbers is whether some shafts will be underutilized or not used at all.

In the examples above, if Table 1 is used, shaft 4 is absent in the threading sequence.

Weavers doing name drafting often try different tables for a chosen string to see which one gives the best results.

Using only coding tables specified in the literature is an example of the dominating role of rote among unsophisticated weavers.

There are strong statistical patterns in the frequency in which characters appear in written text (usually considered only in terms of letters). None of the tables above is close to being balanced for ordinary English text. A subsequent section will give some coding tables that use letter-frequency information in an attempt to balance shaft usage.

Nonetheless, any predefined mapping can be defeated by a particular string — not to mention the fact that the string chosen may not contain as many *different* characters as there are shafts. In practice, strings are chosen to work around such problems.

Modifying Sequences for Weaving

Name drafts usually are woven in overshot. A technical requirement for overshot is that the shaft numbers alternate between odd and even, called *alternating parity*. When a sequence does not meet this requirement, it is modified by adding incidentals.

For example, in the sequence given earlier,

2, 1, 3, 3, 2, 1, 2, 3, 3, 4, 1, 4, 4

there are four places where incidentals are needed. They could be added as follows, where the incidentals are underlined:

2, 1, <u>2</u>, 3, <u>4</u>, 3, 2, 1, 2, 3, <u>4</u>, 3, 4, 1, 4, <u>1</u>, 4

Of course, adding incidentals increases the length of the sequence.

There are various formulas used for adding incidentals. In this example, an incidental is one more than the preceding value, with 4 wrapping around to 1.

Other Aspects of Name Drafting

Name drafts usually are reflected about their centers to add symmetry and increase the visual appeal of the resulting weaves. For the example above, the resulting threading sequence would be

2, 1, 2, 3, 4, 3, 2, 1, 2, 3, 4, 3, 4, 1, 4, 1, 4,
1, 4, 4, 4, 3, 4, 3, 2, 1, 2, 3, 4, 3, 2, 1

Note that the last term in the original sequence is not included in the reflection; that would violate the alternating parity requirement.

In overshot weaves, the tie-up usually is a twill. Different tie-ups often have a dramatic effect on the weave. Again, it is a matter of experimentation.

Examples of drafts based on the sequence above are given at the end of this section.

Name Drafting in Perspective

Certainly name drafting is an *ad hoc* mechanism for producing threading sequences. It is telling that only letters are considered and that upper- and lowercase letters always are taken to be equivalent without anything being said. This is akin to the problem of a person who is not familiar with computing and has trouble with the fact that a blank is just a much of a character as X.

To weavers, however, name drafting can serve a real purpose, which is indicated by the alternative term “commemorative drafting”. The string chosen may have a meaning that is personal to the weaver, resulting in a weave embodying this meaning.

This aspect of name drafting is sometimes forgotten, however. A recent article on name drafting [6] described the author's attempts to find a string that produced an attractive weave, finally settling on "The Random House Dictionary" as the result of glancing at a nearby bookshelf. The resulting weave was attractive, but it hardly carried a special meaning, as the author admitted.

Another Method of Obtaining Alternating Parity

Alternating parity can be obtained by associating odd-even shaft pairs with the rows in a name table. For Table 3, it might look like this:

AEIMQUY	1,2	Table 3
BFJNRVZ	2,3	
CGKOSW	3,4	
DHLPTX	4,1	

Then the appropriate shaft can be chosen as the threading sequence develops. For our example, the result is

2, 1, 4, 3, 2, 1, 2, 3, 4, 1, 2, 1, 4

With this method, the length of the sequence is just the number of letters in the string used. This method for obtaining alternating parity will be used in subsequent sections.

Crackle Weave

Crackle weave, a version of point twill, offers many possibilities for interesting patterns and, if done in the standard manner, has maximum floats of three and makes a strong cloth.

Block Design

Blocks

Conventional crackle weave design is based on blocks with 3 shafts and 4 ends. For 4 shafts, the blocks are

A 1, 2, 3, 2

B 2, 3, 4, 3

C 3, 4, 1, 4

D 4, 1, 2, 1

Adjustments at Block Boundaries

Blocks can be arranged in any sequence, although some sequences produce better results than others. Examples of block sequences are

AAABBBCCCDDD

and

ABCDCBABCDCBA

Incidentals are inserted or ends removed to meet the structural requirements of crackle weave. Berta Frey [1] lists these rules (abbreviated and paraphrased here):

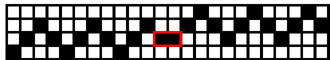
1. An odd/even progression of shafts must be maintained.
2. The 3-shaft character must be maintained; incidentals can be added or ends removed to achieve this.
3. There may be no more than three threads on two adjacent shafts (for example, 2, 1, 2, 1 is not allowed).
4. There may be no more than 4 shafts before direction changes.

Although incidentals can be inserted and ends removed in various ways,

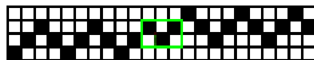
the rules established by Harriet Tidball [2] are logical, systematic, and now generally used in crackle weave design. If an incidental is needed after a block, it is put on the shaft that is one less than the last thread of the block:

A	1
B	2
C	3
D	4 (wrapping around from 1)

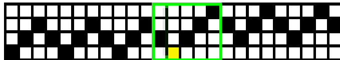
If the same block is used in succession, as in AAA, no incidentals are required. Going from one block to the next, as in AAABBB, however, there is a problem:



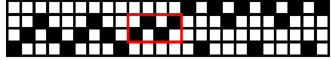
The adjacent duplicates are outlined in red. There are two choices. One is to delete one of the duplicates:



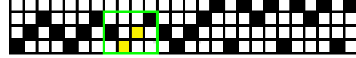
The area where the duplicate was removed is outlined in green. The other choice is to insert an incidental, shown in yellow:



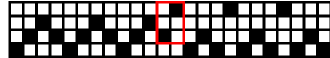
If a block is skipped, as in AACCC, Frey's Rule 3 is violated:



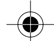
Incidentals for block A and the skipped block B need to be inserted

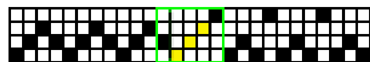


The same principle applies for skipping two blocks, as in AAAADDDD:



where the incidentals for blocks A, B, and C need to be inserted:





It is worth noting that only four essentially different situations occur at block boundaries:

- Either an end can be deleted or an incidental inserted
- No change is needed.
- An incidental is needed to connect the shafts.
- Two incidentals are needed to connect the shafts.

There are, of course, the horizontal reflections of these, to which the same rules apply.

More Shafts

Crackle weave is not limited to 4 shafts; more shafts can be used. For example, for 6 shafts, there are six blocks:

A 1, 2, 3, 2
B 2, 3, 4, 3
C 3, 4, 5, 4
D 4, 5, 6, 5
E 5, 6, 1, 6
F 6, 1, 2, 1

The same rule for incidentals applies.

Crackle also can be woven on an odd number of shafts.

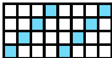
If more that 4 shafts are used, there are more different situations that may arise at block boundaries.

Motif Along a Path

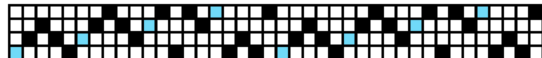
There is another way to view the design process for crackle weave.

Note that blocks B, C, and D are simply successively upwards shifted versions of block A, with wrap-around from top to bottom. There is only one *motif*, which can be taken to be block A.

This motif can be placed as successive points along a *path* to give the same result as using different blocks in succession. For example, if the path is a straight draw, as in



the result is the same as using the block sequence ABCDABCD:



The first threads of successive motifs are shown in blue to emphasize the path.

Incidentals are, of course, handled in the same way as for blocks.

The advantage of motif-along-a-path design is that different motifs and paths can be tried independently. For example, the motif 1, 2, 3, 2, 1, 2, 3, 4, 3 is equivalent to the block sequence AB with the required incidental. But by using the combination motif, design can be done in terms of a single motif.

If a motif not corresponding to combinations of blocks is used, such as 1, 2, 3, 4, 3, 2, the result may be floats longer than 3 and not true crackle. But many more patterns are possible and if care is taken, the resulting cloths will be sound.

Tie-Ups

Crackle is a twill weave, a variety of point twill. As such, twill tie-ups normally are used. In order to keep the maximum float length to 3, expected of conventional crackle weaves, twill counters must be less than 3.

The following tie-ups work well for crackle:

4 shafts:

2/2

5 shafts:

1/2/1/1

6 shafts:

1/1/2/2
1/2/1/2
1/2/2/1
2/1/2/1

08 May 2006

F1: Crackle

7 shafts:

1/2/1/1/1/1
2/2/2/1

8 shafts:

1/1/1/1/2/2
1/1/1/2/1/2
1/1/1/2/2/1
1/1/2/1/1/2
1/1/2/1/2/1
2/2/2/2 (same as a 4-shaft twill)

9 shafts:

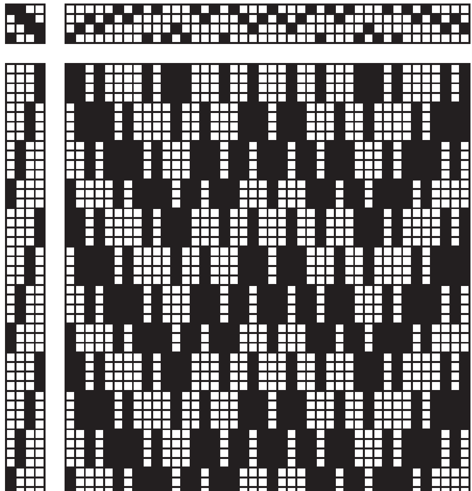
1/1/1/2/2/2
1/1/2/1/2/2
1/1/2/2/1/2
1/1/2/2/2/1
1/2/1/2/1/2
1/2/1/2/2/1
1/2/2/1/2/1
2/1/2/1/2/1

10 shafts:

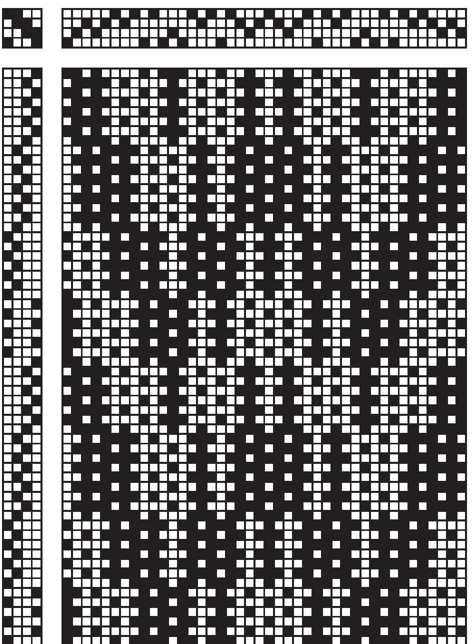
1/1/2/2/2/2
1/2/1/2/2/2
1/2/2/1/2/2
1/2/2/2/2/1
2/1/2/1/2/2

Treadling

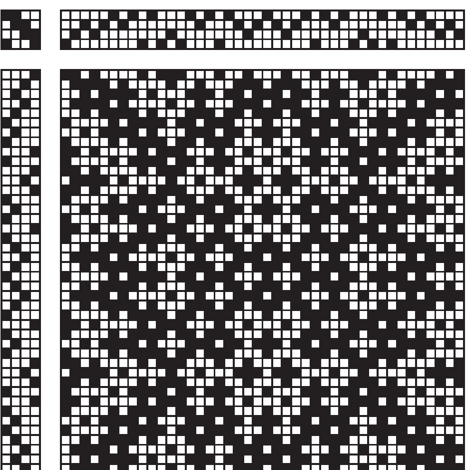
Traditional crackle threading uses the same treadle for all the threads of one block, with pattern picks alternating with binding picks. Here is an example, in which the binding picks are not shown:



Summer and Winter is another conventional treadling for crackle, with two treadles alternating, the first block woven on treadles 4 and 3, the second on 3 and two, the third on 1 and 2, the fourth on 1, and 4, and then repeating. Here is an example:



Crackle also can be treadled as drawn in. This is the treadling now most commonly used. Here is an example:

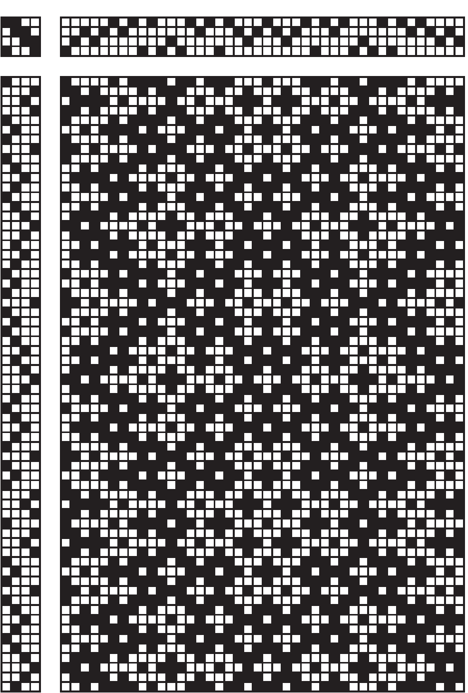


See Reference 2 for more information on conventional crackle treadling.

Many other treadlings have been tried, including straight draws, point twill, and advancing twills.

Crackle treadling is an area for experimentation, which may produce interesting results.

One particularly interesting method is to use one crackle sequence for threading and a different crackle sequence for treadling. Here is an example:



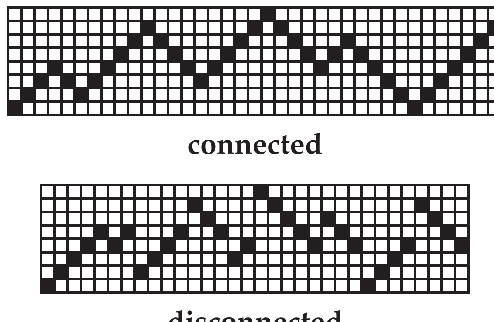
Path Design

Except for the path, all is formular in conventional crackle weave design: the 1,2,3,2 motif and specified ways for fixing problems that may occur between the boundaries of the successively placed motif. The only design element is the path.

In design, a path is not constrained to a specified number of shafts. Adaptation to a specified number of shafts is done after the motif is placed along the path.

Path Properties

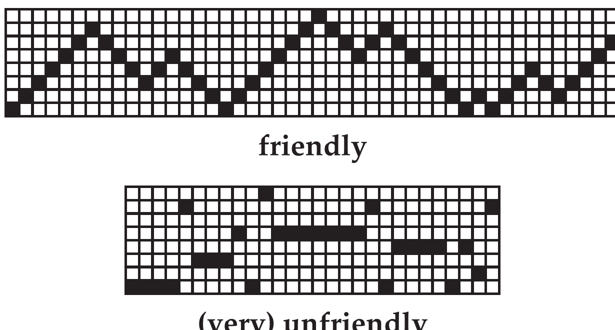
Paths can be classified in a variety of ways. For example, a path may be connected or disconnected, as shown in these examples:



connected

disconnected

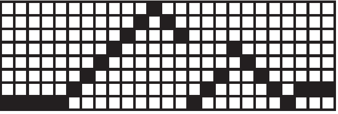
A path may be “friendly” or “unfriendly” [2]. A friendly path is one in which each value is one greater or one less than the preceding one. Friendly paths are, of course, connected. Here are examples of friendly and unfriendly paths:



friendly

(very) unfriendly

Unfriendly paths may, of course, have friendly segments, as in



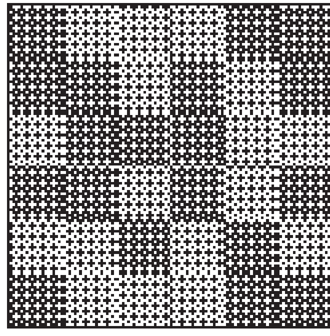
(somewhat) unfriendly

The *extent* of a path is the difference between its largest and smallest values, plus 1. Without losing anything (or, as a mathematician would say "without loss of generality"), the smallest value can be taken to be 1, so that the extent of a path is given by its largest value.

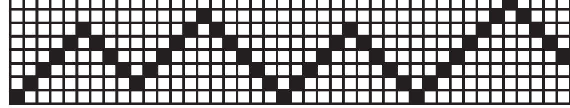
Design Considerations

Paths with "flats", runs of the same value, as in

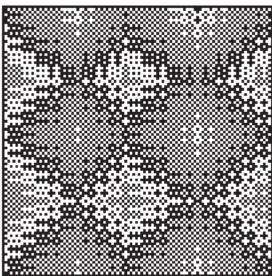
produce blocky, rectilinear patterns. The path shown above produces this drawdown for 4 shafts and a 2/2 twill tie-up, treadled as drawn in:



Friendly paths can produce varied patterns. An example is

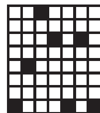


A drawdown based on placing the motif on this path is

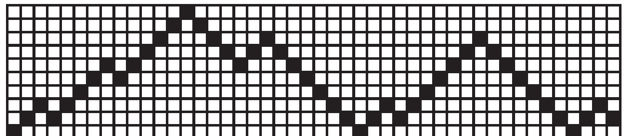


Paths with large extents are capable of producing more elaborate patterns than paths with small extents. However, the number of shafts used limits the possibilities once the motif has been placed along the path.

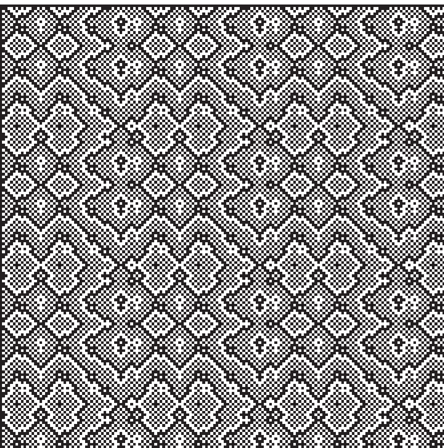
Perhaps surprisingly, disconnected paths are capable of producing the most dramatic patterns. Here is an example of a disconnected path:



Although the path is short, placing the motif along it produces a much longer sequence because of the incidentals needed to connect separated motifs:



A drawdown with this sequence adapted to 8 shafts is



The reason disconnected paths can produce elaborate patterns is the runs

introduced by the incidentals needed to connect separated motifs. One problem with disconnected paths is that they may result in floats longer than 3. Nonetheless, they often produce sound fabrics and the marvelous patterns that can result may more than compensate for the increased float lengths.

Horizontal reflection of a threading sequence to produce a palindrome usually produces a pattern that is more attractive than the unreflected sequence. However, since the motif itself is not symmetric, the result of placing a motif along a palindromic *path* is not a true palindrome. Palindromic enhancement is best done after the motif is placed along the path.

Results of Motif Placement

When the motif is placed along a path, the maximum value in the resulting sequence is two greater than the extent of the path, since the motif goes up two above the largest path point.

Once the motif is placed along the path, modular reduction [3] can be used to bring the sequence within the range of the number of shafts to be used.

Experimenting with Crackle Design

The number of possible paths for all but trivial situations is very large. If the range of a path is n and its length is k, there are n^k possible paths. For example, if n is only 4 and the k is only 8, there are $4^8 = 65{,}536$ possible paths.

Even with a program to run through the possibilities, it is impractical to create, much less evaluate, all crackle weaves drafts of even modest extent.

But that is the challenge of intelligent, artistic exploration — to find gems in vast mountains of debris.

[The dependency of this material on Painter's Weaving Language raises serious questions about the order of presentation in M.O.]

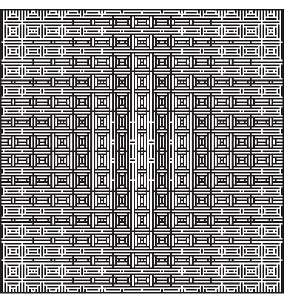
Shadow Weave

Reference Ω.1, which describes the weaving language in MetaCreation’s [get current company name] Painter application, is prerequisite to the material that follows. An example given there is shadow weave [2, 3]. A textual form of the Painter draft is

Shadow Op Art	*name*
1–8–2–828–3–82128–4–8214128–5–821434128–6–82143634128782143656 34128|,1	*threading*
1–8–2–828–3–82128–4–8214128–5–821434128–6–82143634128782143656 34128|,1	*treadling*
KW–>183	*warp colors*
WK–>183	*weft colors*
1010101001010101101010010101011010100101010110101001010101101010	*tie-up*

W and K stand for white and black, respectively.

This draft produces the weave



A Shadow Weave

Weaves of this type produce the appearance of shadows (which are more obvious on actual woven fabrics than in images) by alternating light and dark threads in reverse orders in the warp and weft.

The threading and treadling expressions for shadow weaves typically are the same — treadled as drawn in, as is the case here. Therefore only the threading expression is needed.

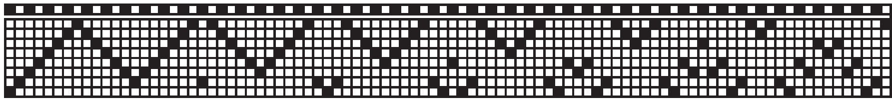
The threading expression is a (true) palindrome. This follows from the fact that the pattern palindrome operator, |, has very low precedence and the expression groups like this:

((1–8–2–828 …436341287821436563412 8)|,1)

The 1 concatenated at the end converts a pattern palindrome into a true one. The weave looks better when repeated if this last character is omitted, leaving a

pattern palindrome.

The threading expression consists of a sequence of domain runs — "ups and downs" — between other shaft sequences. This is easier to understand graphically than in terms of numbers. This figure shows the threading for the first half of the sequence. The bar at the top shows the colors.



The Threading

The operand of the pattern palindrome operator has a definite structure:

1–1–2–2–3–3–4–3–5–5–6–8214363412878214365634128

where the small numbers have their own structure: [Lost a font here and used "small numbers" as a temporary work-around.]

1 = 8
2 = 828
3 = 82128
4 = 8214128
5 = 821434128

Note that these all are true palindromes.

After –6–, the pattern appears to break down, although there are similarities with the earlier parts. In fact,

8214363412878214365634128

is equivalent to

82143634128–7–8214365634128

So the result is

1–1–2–2–3–3–4–3–5–5–6–6–7–7

with the continuation of the palindromes between:

6 = 82143634128
7 = 8214365634128

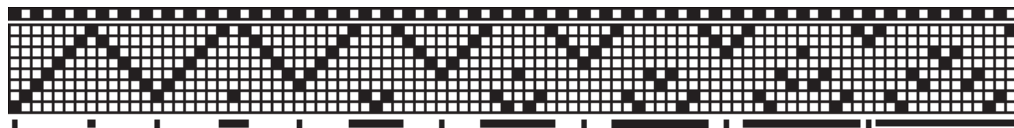
These palindromes can be represented using pattern forms, which makes the underlying structure more evident:

1 = [!8]
2 = [8!2]

3 = [82!1]
4 = [821!4]
5 = [8214!3]
6 = [82143!6]
7 = [821436!5]

The sequence 8241365 runs not only across but also down the center of these palindromic forms — patterns within patterns.

One way to view the overall pattern is as a sequence of anchors for domain runs, which are connected by palindromes. The following figure shows the threading draft with the anchors indicated by vertical bars and the palindromes by horizontal bars.



Threading Draft Showing Anchors and Palindromes

There several questions at this point. The first ones that come to mind are:

- If this pattern is modified in various ways, what kinds of weaves result?
- Is the threading pattern somehow special or just one of a class of patterns that produce interesting weaves?
- If so, how can this class be characterized?

Start with the first question, take the domain runs as given, and concentrate on the sequence of anchors and palindromes. For this, it is easier to deal with character sequences. Digits will be used for labeling the shafts and the letters A though G to label the palindromes. Thus, the sequence can be represented as

1A2B3C4D5E6F7G

In terms of pattern forms, this is an interleaving:

[1234567~ABCDEFG]

More formally, label the anchor sequence $\mathcal{A}$ and the palindrome sequence $\mathcal{P}$, giving

[$\mathcal{A}$ ~ $\mathcal{P}$]

Given transformations τ_1 and τ_2 on sequences, consider

[$\tau_1(\mathcal{A})$~ $\tau_2(\mathcal{P})$] *general transformations*

One possibility is coupling the anchors and the palindromes, that is $\tau_1 \equiv \tau_2$:

$[\tau_1(\mathcal{A}) \sim \tau_1(\mathcal{P})]$ *coupled transformations*

An example of this, using our original notation, is the permutation

6–6–3–3–1–1–4–4–5–5–2–2–7–7

Another possibility is using the identity transformation ι on one but not the other component:

$[\tau_1(\mathcal{A}) \sim \iota(\mathcal{P})]$ *anchor transformations*

or

$[\iota(\mathcal{A}) \sim \tau_2(\mathcal{P})]$ *palindrome transformations*

Respective examples are the permutations

5–1–4–2–3–3–2–4–1–5–7–6–6–7

and

1–5–2–6–3–7–4–4–5–3–6–2–7–1

This is not limited to permutations. Examples of transformations that are not permutations are the coupled transformation

1–1–2–2–3–3–4–4–4–4–3–3–2–2

and this transformation, which increases the length of the sequence

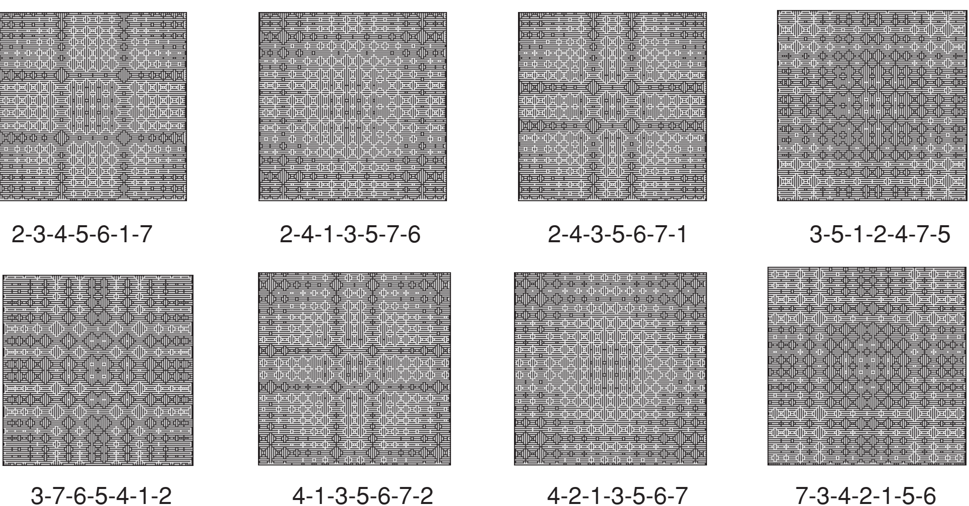
1–5–2–6–3–7–4–4–5–4–6–2–7–1–1–5–2–6

It is, of course, impossible to explore all such transformations. For permutations alone, there are $14! \cong 8.7 \times 10^{13}$ possibilities for the general case.

There are, however, only $7! = 5{,}040$ permutations for the coupled anchor and palindrome cases. All the anchor-sequence permutations to give a feel for how the weaves differ.

No two of the weaves are the same, although many are so similar that the differences cannot be detected without detailed examination. There is some difference in the size of the weaves. This is to be expected, since the lengths of the domain runs change when the anchors do. The size is determined solely by the first anchor. If the first anchor is i, then the weave is $180 + 2i$ threads on a side.

All are visually attractive, at least to us, and the range of design variations is relatively small. The 10 weaves that follow represent the visual extremes. The pattern is aesthetically robust.


2-3-4-5-6-1-7
2-4-1-3-5-7-6
2-4-3-5-6-7-1
3-5-1-2-4-7-5
3-7-6-5-4-1-2
4-1-3-5-6-7-2
4-2-1-3-5-6-7
7-3-4-2-1-5-6

Constrained Patterns

Constraints limit what is possible. With respect to interlacement patterns, constraints impose both color and structural limitations. Constraints can take many forms. Expressed in terms of drawdowns, examples are:

1. The number of white cells and black cells must be equal.
2. No more than four consecutive cells in any row and column can be the same color.
3. Every cell must have at least one adjacent cell of the opposite color.
4. Constraints 1, 2, and 3 all must be satisfied.

Constraint 1 is a global constraint and is equivalent to requiring that a weave be balanced. This constraint cannot be satisfied by a drawdown with an odd number of cells. That is, of all drawdowns, only ones with even dimensions can possibly satisfy this constraint.

Constraint 2 is more local and in more familiar terms limits float length.

Constraint 3 is local; it specifies a property that all neighborhoods must have. [Cross-reference cellular automata.]

Constraint 4 requires that three constraints be simultaneously satisfied. It is called a *constraint set*. In this sense, Constraints 1, 2, and 3 are constraint sets containing only one constraint.

Constraint Analysis

Given a pattern, it generally is easy to determine if it satisfies a given constraint set. For example, whether or not a pattern satisfies Constraint Set 1 can be determined just by counting black and white cells. Similarly, whether or not a pattern satisfies Constraint Set 2 can be determined by examination or using the float-analysis feature of a weaving program.

Constraint Set 3 requires a little more work, since it may be necessary, in general, to examine a large number of individual neighborhoods.

And, of course, determining whether or not a pattern satisfies Constraint Set 4 can be determined by checking each of the constraints in its constraint set.

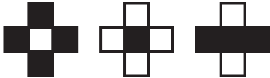
It is important to realize that there are constraint sets that no patterns satisfy. For example, a constraint set that contains Constraint 1 and a constraint that patterns must have an odd number of cells cannot be satisfied — it is *unsatisfiable*. In this example, it is obvious that the two constraints are mutually exclusive. In general, it may be difficult to determine whether or not a constraint set can be satisfied any pattern.

Neighborhood Constraints

As far as weave structure is concerned, neighborhood constraints are interesting, since they have a strong effect on appearance.

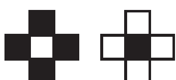
Neighborhood constraints can be characterized by *neighborhood templates*. As with drawdown automata, there are many kinds of neighborhoods that can be used. The von Neumann 5-cell neighborhood [2] is used in what follows. This neighborhood is small enough to be computationally tractable but large enough to characterize a wide range of structural characteristics.

Neighborhood constraints can be pictured like this:



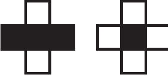
None of these constraints *taken alone* is satisfiable, simply because they all require every cell to be the same color (white in the first, black in the other two) while simultaneously requiring that them to be surrounded by cells of other colors.

Taken in combination in constraint sets, however, they may be satisfiable. For example, the constraint set consisting of the templates



is satisfied by plain weave (and only plain weave).

On the other hand, the constraint set consisting of the templates



is unsatisfiable because it requires every cell to be black and at the time to have adjacent white cells.

Neighborhood constraints can be looked at in several ways:

- Does a pattern satisfy a given neighborhood constraint set?
- What neighborhood constraint set does a given pattern satisfy?
- What patterns satisfy a specific neighborhood constraint set?

The first question is easy to answer: as mentioned above, it's only necessary to compare the cell neighborhoods to the templates in the constraint set.

The second question also is generally easy to answer by cataloging the

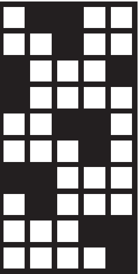
neighborhoods of all cells, although there are some issues to be addressed, such as how to handle cells at the borders that do not have complete neighborhoods.

The third question is, in general, much harder to answer. It is, nonetheless, interesting. For example, it would be interesting to know what patterns satisfy the same neighborhood constraint set that a 2/2 twill does. The problem is hard because there is no is known way to construct patterns from constraint sets that does not involve a large amount of computation.

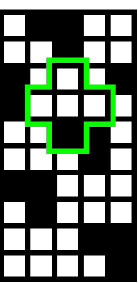
Neighborhood Analysis

In the first article on constraints, we introduced the concept of neighborhood constraints [1]. In this article, we'll look at the problem of determining the neighborhood constraint set of a pattern.

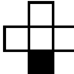
Consider the following pattern:



All that is necessary to determine the constraint set for this pattern is to examine every cell and record the template for its neighborhood.



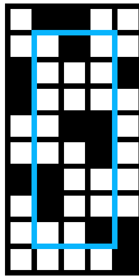
For example, the template for the cell outlined above is



This process is straightforward except for cells at the edges, which have incomplete neighborhoods. There are several ways to handle such cells:

1. Don't include the edge cells in the analysis.
2. Assume that the pattern repeats so that the edges wrap around.
3. Don't assume the pattern repeats (for example, the Morse-Thue carpet does not [2]) but include partial neighborhoods of the edge cells.

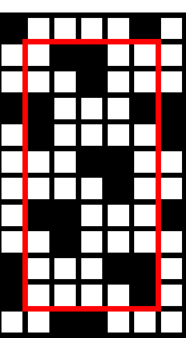
Method 1 amounts to analyzing a sub-pattern, shown by the blue outline below:



The problem with this approach is that the constraint set obtained may not be complete. For example, the unit motif for plain weave is a 2 × 2 pattern:

This pattern only has edge cells. If they are ignored, there is no constraint set at all, which obviously is incorrect.

Method 2 can be handled by augmenting the pattern, adding cells around the edges that correspond to what would appear if the pattern were contained in a repeat:

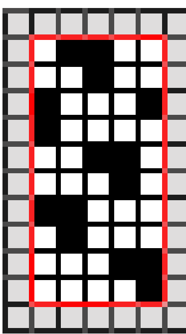


Now the analysis can proceed for the cells enclosed in the red rectangle above;

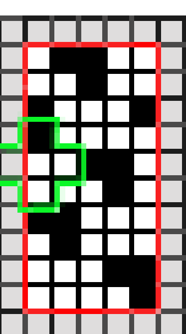
effectively there are no edge cells.

This method is fine for repeating patterns, but it produces erroneous results for aperiodic patterns such as the Morse-Thue carpet.

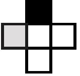
Method 3 tries to deal with this situation by adding edges with unknown cell colors:



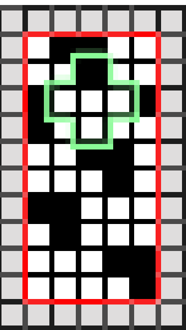
In this case, an edge cells such as the one outlined below has a neighborhood template with a cell of unspecified color:



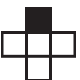
Here is that template:



It can be added this partial constraint to the set. But note that there is other cells in the pattern with complete templates that have the same three cells as the partial constraint:



This neighborhood,



"covers" the incomplete one, it is not necessary to keep the incomplete one.

If partial constraints remain after analyzing all cells, one possibility is to just "force them" by arbitrarily coloring the unspecified cells.

What to do about an aperiodic patterns is an open question. One can analyze a portion of it using Method 3. But how can one tell if the constraint set obtained is complete? Would analyzing a larger portion add to the constraint set?

In the case of the Morse-Thue carpet, analyzing a modest portion yields a constraint set with 18 templates. Analyzing larger portions do not increase the size of the constraint set. It seems reasonable, examining the method by which the Morse-Thue carpet is constructed, that this constraint set applies to the entire, unlimited pattern.

But for other patterns, such as random ones, there is no basis for such an assumption. In fact, the constraint set for a randomly generated pattern may include all 32 possible constraints.

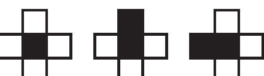
On the other hand, what is the point of trying to determine the neighborhood constraint set for a pattern without structure?

Representing Constraint Sets

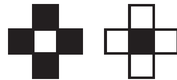
Neighborhood constraint sets can be represented in several ways. For human understanding, graphical methods work best. For computer processing, textual representations or numerical codes are more appropriate.

Graphical Representations

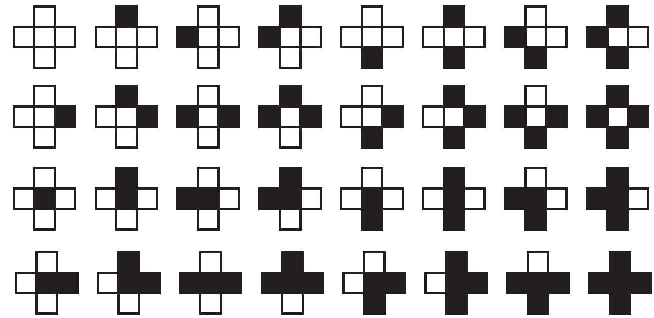
In previous articles, we showed templates as neighborhoods laid out according to their natural geometrical interpretation, as in



Any constraint set then can be represented by a collection of such template images. For example, the constraint set for plain weave is



If a constraint set is large, this kind of representation takes a fair amount of space for images of a size sufficient to be readily understood. For example, the constraint set containing all constraints is



A less useful but more compact graphical representation is as a bar of 32 cells, each cell corresponding to one of the constraints. If a cell is in a constraint set, it is colored gray, otherwise white. Gray is used so the black dividing lines can be used as a guide to cell position. For example, the cell bar for the plain-weave constraint set is

The problem with the cell-bar representation is that the templates are coded by position, so that to determine the constraints, it's necessary to know where individual templates are in the bar and the order of the templates (which is as shown in the image for all templates, reading left to right and top to bottom. Determining the actual templates in this way is tedious and error prone, so the cell-bar representation is not appropriate for that purpose. It is suitable, however, for getting an idea of the number of constraints in a set and comparing

patterns of different constraint sets.

Textual Representations

The graphic representation as a series of templates has a natural counterpart as a list of 5-bit binary strings in which a bit is 1 if the corresponding cell in the neighborhood is black and 0 otherwise.

A convention is needed to determine the order of the bits in the bit string. The convention we'll use here numbers the cells starting with the center cell and continuing clockwise around the outer cells:

```
   5
4  1  2
   3
```

Therefore the plain-weave constraint set has the textual representation

01111

10000

(Since this represents a set, the order of the constraints is not important, but a useful convention is to order the binary strings by magnitude, as we have done in this example.)

A more compact textual representation of constraint sets is as 32-bit binary strings in which a bit is one if the corresponding constraint is in the set and 0 otherwise. For example, the plain-weave constraint set represented in this way is

00000000000000011000000000000000

Note that although the cell-bar representation is difficulty for a human being to interpret in its entirety, the 32-bit binary string representation presents no problem for a program: It's just another decoding task of the kind that programs have to handle all the time.

Numerical Codes

A variation on the textual representations is to think of bit strings as base-2 integers. These base-2 integers then can be converted to conventional base-10 integers. For example, in terms of 5-bit constraints, the plain-weave constraint set has the numerical codes

15

16

while the 32-bit representation has the numerical code 98305.

For computer programs, these are just other ways of encoding and present no more problems than the textual forms. Base-10 integers have advantages for programming in some situations because all commonly used programming languages support integer arithmetic. However, the equivalents of 32-bit bit strings can be very large: as large as 4,294,967,296, which is beyond the range of integer arithmetic in most programming languages.

Numerical codes are somewhere between graphical representations, suitable for human beings, and textual representations, suitable for computers. For human beings, they do have value as labels, if arbitrary, and are only about one-third the length of the corresponding binary strings, as well as being easier to differentiate than bit strings.

von Neumann Constraint Pattern Catalog

[This section has not been published on the Web.]

Wolfram [1] has shown that only 171 repeating patterns are needed to characterize all the von Neumann neighborhood constraint sets [2]. (Rotations, reflections, and color reversals are omitted.)

The pages that follow show unit weaves for these patterns in the order given in Reference 1. Below each pattern are its dimensions and, if weavable from a drawup, the loom resources required. At the bottom is a cell bar showing the constraints involved [3].

Some of the patterns that are unweavable as drawups can be drafted using color-and-weave effects. Obvious examples are the stripes.

All of the patterns that are weavable from drawn-up drafts “hang together” [4]. [Cross reference; consider changing “weavable”to “draftable” here and elsewhere.]

1 1x1	2 5x5 5x5	3 2x4	4 12x12 12x12
5 1x3	6 6x2	7 1x2	8 3x4
9 3x3 3x3	10 6x3 3x3	11 5x5 5x5	12 4x4 4x4
13 5x10 5x10	14 3x9	15 8x8 8x8	16 4x4 4x4

17	18	19	20
3x3	9x9 9x9	3x12	4x4 4x4
21	**22**	**23**	**24**
8x4 4x4	5x4 5x4	4x8 4x8	7x7 7x7
25	**26**	**27**	**28**
5x10 5x10	4x3 3x3	9x3 6x3	12x3 6x3
29	**30**	**31**	**32**
6x6 6x6	3x2	4x2 2x2	10x10 10x10

33	34	35	36
3x6 3x6	7x14 7x14	4x3	3x3

37	38	39	40
8x8 8x8	9x18 9x18	3x5	11x11 11x11

41	42	43	44
15x5 15x5	15x5 15x5	6x9 6x9	12x4 10x4

45	46	47	48
17x17 17x17	3x9 3x9	16x20 16x20	12x3

49	50	51	52
13x13 13x13	15x5 10x5	10x5 10x5	4x16 4x16
53	**54**	**55**	**56**
10x10 10x10	5x15 5x15	7x20 7x20	20x7 20x7
57	**58**	**59**	**60**
14x14 14x14	4x6 4x6	4x3 3x3	20x20 20x20
61	**62**	**63**	**64**
20x20 20x20	5x10 5x10	15x3	14x14 14x14

65	66	67	68
7x14 7x14	12x4 12x4	20x20 16x20	20x20 20x20
69	**70**	**71**	**72**
14x7 14x7	15x15 15x15	7x14 7x14	9x18 9x18
73	**74**	**75**	**76**
6x6 6x6	20x9 18x9	2x2 2x2	3x3 2x3
77	**78**	**79**	**80**
16x20 16x20	14x14 14x14	15x3 6x3	12x3 6x3

81	82	83	84
17x20 17x20	5x15 5x15	16x8 16x8	20x20 20x20

85	86	87	88
14x14 14x14	4x6	6x3	7x14 7x14

89	90	91	92
12x3	11x11 11x11	10x4	5x15 5x15

93	94	95	96
8x8 8x8	18x9 18x9	9x20 9x20	20x20 20x20

97	98	99	100
4x5	7x14 7x14	20x20 20x20	6x9 6x9
101	102	103	104
12x12 12x12	7x14 7x14	15x5 10x5	13x13 13x13
105	106	107	108
20x11 20x11	17x17 17x17	6x6 6x6	6x4 6x4
109	110	111	112
9x18 9x18	20x9 18x9	10x10 10x10	12x12 12x12

113 7x14 7x14	114 4x8 4x8	115 20x20 20x20	116 15x3
117 8x16 8x16	118 18x3	119 15x3	120 4x8 4x8
121 5x15 5x15	122 4x20 4x20	123 4x12	124 20x20 17x20
125 14x7 14x7	126 4x12 4x12	127 3x6	128 14x7 14x7

129	130	131	132
13x13 13x13	20x20 18x20	6x4 6x4	20x8 20x8
133	**134**	**135**	**136**
20x4 14x4	16x20 16x20	20x17 17x17	13x13 13x13
137	**138**	**139**	**140**
9x20 9x20	20x20 17x20	16x4 10x4	8x8 4x8
141	**142**	**143**	**144**
12x4 5x4	12x6 12x6	7x14 7x14	17x20 17x20

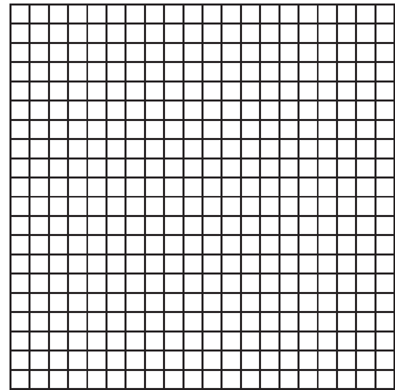
145	146	147	148
20x20 18x20	20x20 20x20	20x11 20x11	20x5 13x5
149	**150**	**151**	**152**
20x17 17x17	20x20 20x20	20x11 20x11	7x14 7x14
153	**154**	**155**	**156**
4x10 4x10	18x6 15x6	20x20 20x20	4x10 4x10
157	**158**	**159**	**160**
5x20 5x20	10x6 4x6	16x16 16x16	18x9 18x9

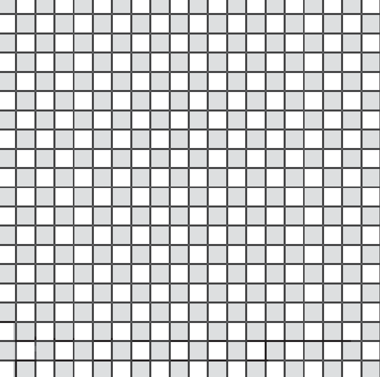
27 May 2006 G1: Constraints

161 11x20 11x20	162 12x6 9x6	163 7x14 7x14	164 20x10 20x10
165 4x16 4x16	166 20x20 20x20	167 6x9 6x9	168 20x20 20x20
169 20x11 20x11	170 4x1	171 3x18	

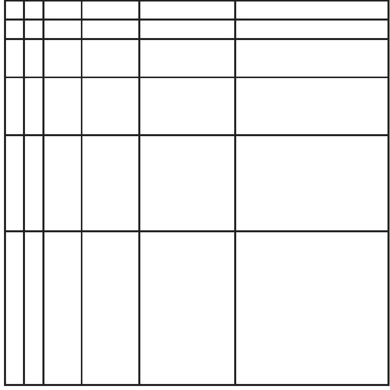
Nonlinear Grid Design

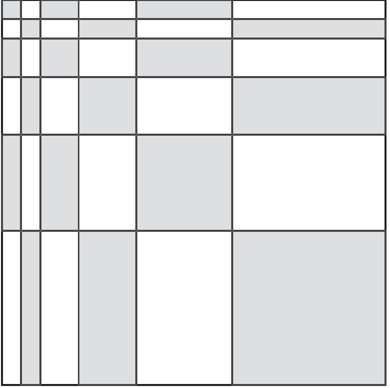
Many design techniques use a linear grid of square cells:



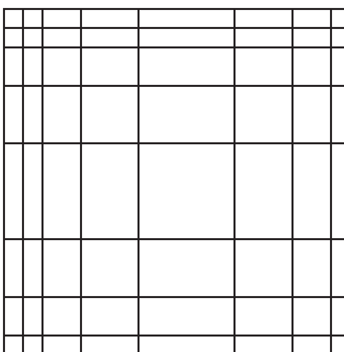


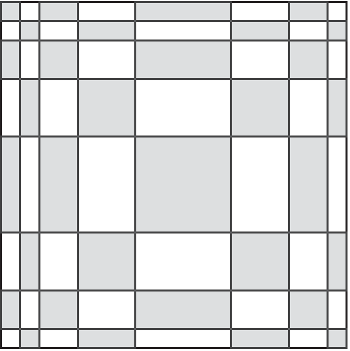
Interesting results can be obtained by using nonlinear grids in which not all cells are square. An example is the Fibonacci grid in which the widths in heights increase according to the Fibonacci sequence: 1, 1, 2, 3, 5, 8, 13, …



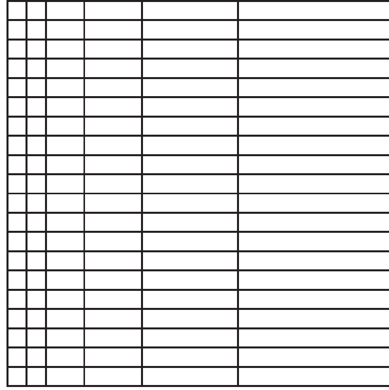


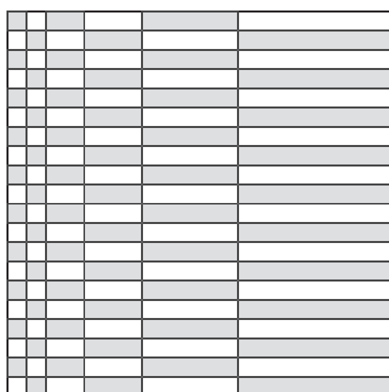
Since the values of the Fibonacci sequence increase rapidly, the cells quickly get too large for interesting designs. An alternative is to take the first few values and then reflect them to get a symmetric grid:





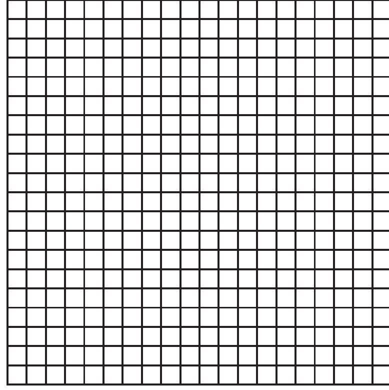
Grids can be characterized by two sequences, one for the widths and the other for the heights, and they need not be the same. For example, the Fibonacci sequence could be used for widths and a constant sequence for heights:



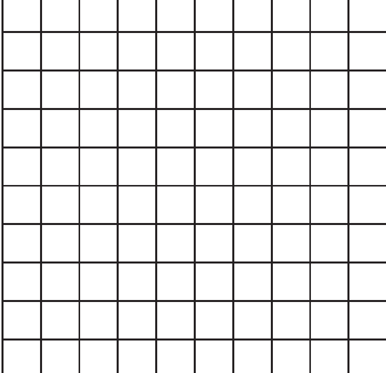


Other aspects of grids are the scaling and resolution: how large a cell actually is. In the grids in this article, the scaling factor is 10 and the resolution is 100 per inch an inch. For example, a width specification of 1 produces a width of 1/10th inch.

Here is a linear grid with a scaling factor of 10 and a resolution of 100:



Changing the scaling factor to 20 produces



Fractal Grids

Fractal sequences can be used as the basis of nonlinear grids. By their nature, fractal sequences do not repeat and so fractal grids do not tile seamlessly if repeated. However, many fractal grids appear to be regular yet with just enough difference to be interesting.

Morse-Thue Grids

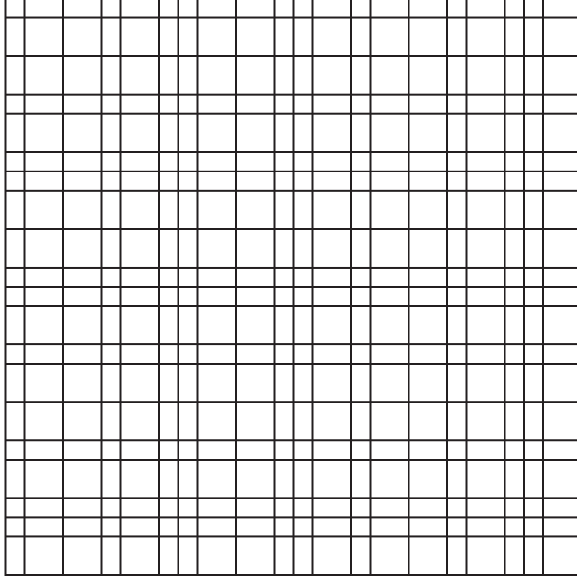
The most famous fractal sequence is the binary Morse-Thue sequence [1]. It goes like this:

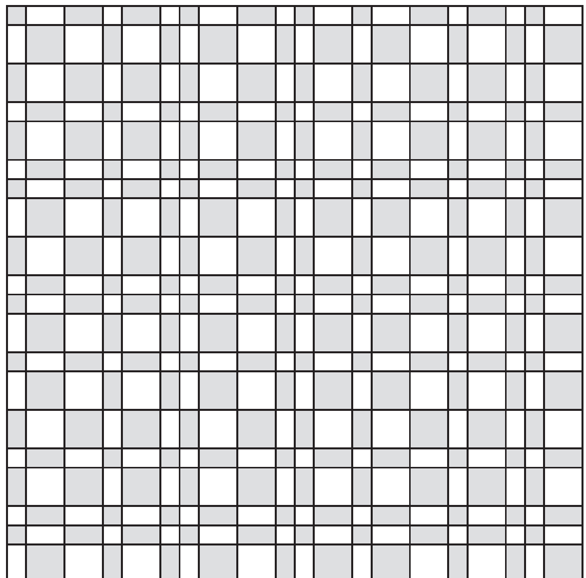
0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, …

In order to make this sequence suitable for widths and heights, 1 can be added to each value:

1, 2, 2, 1, 2, 1, 1, 2, 2, 1, 1, 2, 1, 2, 2, 1, …

The resulting grid is

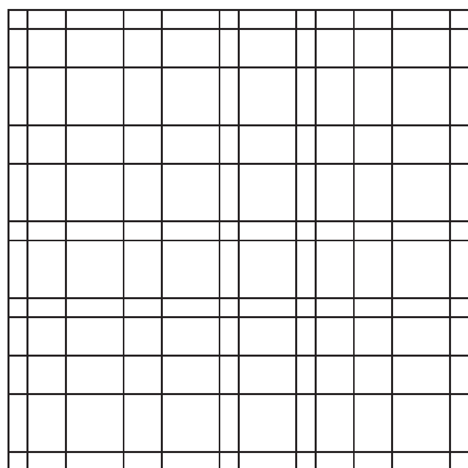


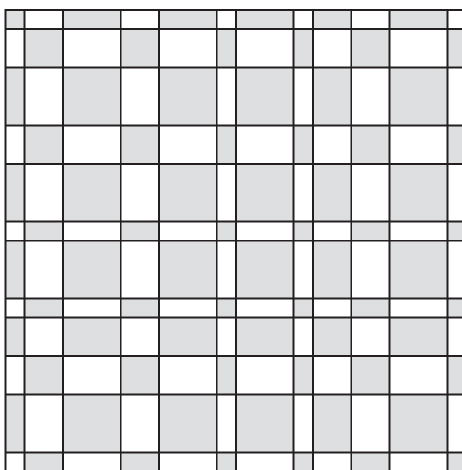


The Morse-Thue sequence can be generalized to base 3, base 4, and so on to give a greater variety of widths and heights. For example, the base-3 Morse-Thue sequence, incremented by 1, starts out as

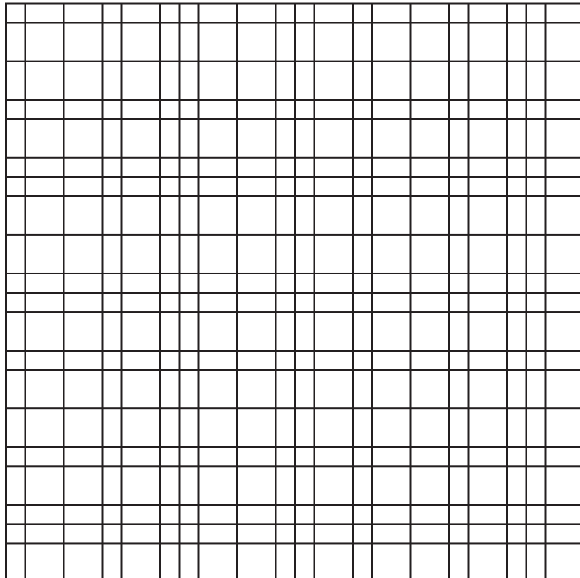
1, 2, 3, 2, 3, 1, 3, 1, 2, 2, 3, 1, 3, …

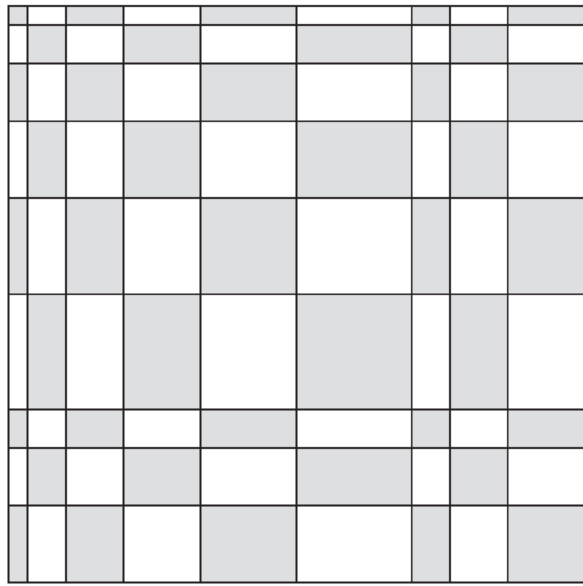
and produces the grid





The base-6 Morse-Thue sequence produces this grid:



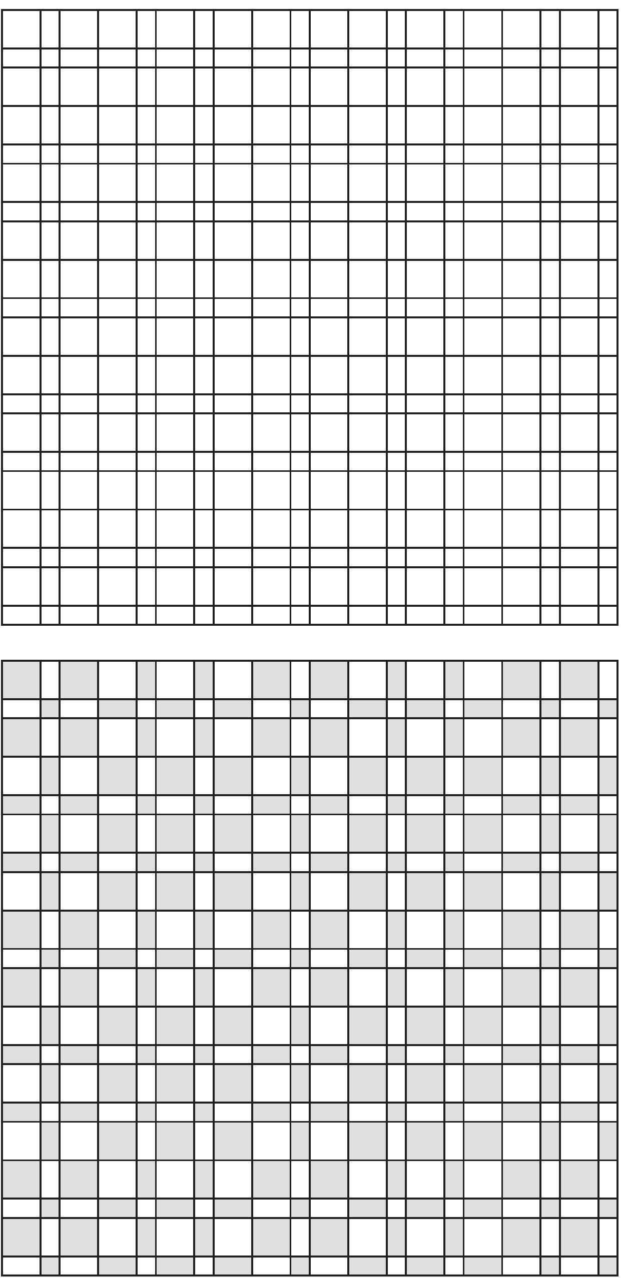


Rabbit Grids

Another binary fractal sequence is the rabbit sequence [2], which incremented by 1 starts out as

2, 1, 2, 2, 1, 2, 1, 2, 2, 1, 2, 2, 1, 2, 1, 2, …

The grid for this sequence is



As with other sequence-based nonlinear grids, the horizontal and vertical sequences can be different.

There are dozens of other fractal sequences that are suitable for making nonlinear grids. Go to the Web site given in Reference 3 and search for the keyword fractal.

These are only a few of an unlimited number of possibilities. And it's easy to make your own. Start with linear grid paper and rule off widths and heights according to the sequences you want.

Operations on Patterns

Many interesting patterns can be created by operations on other patterns, changing and combining them in various ways.

Patterns can be represented as grids of cells. When a pattern is interpreted as a drawdown, black cells indicate where the warp is on top and white cells where the weft is on top. Figure Ω.1 shows an example. [Covered elsewhere.]

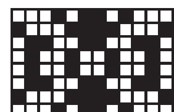


Figure Ω.1. A Drawdown Pattern

Cells in lines across the pattern from top to bottom are called columns, while cells in lines from left to right are called rows. The word lines is used for both in situations in which orientation is not important. See Figure Ω.2.

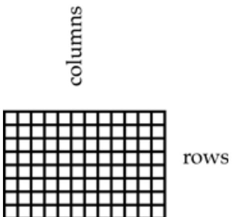


Figure Ω.2. Columns and Rows

A variety of operations can be performed on such patterns. They can be changed by geometrical transformations, such as rotation. Two patterns can be concatenated (adjoined) to form a larger pattern. A portion of a pattern can be replaced by another pattern. The rows and columns can be rearranged. And a pattern can be turned over to show its back side, as in the back of a woven fabric.

Notation

Uppercase italic letters, like *P*, *Q*, and *R*, are used to name patterns for the purpose of identification.

Some operations on patterns require integer values. Lowercase italic letters, such as *i*, *j*, and *k*, are used for these.

Various symbols are used to stand for operations on patterns.

Pattern Properties

Two properties of patterns are important in many operations:

- width, the number of columns, in a pattern P is denoted by $\omega(P)$
- height, the number of rows; in a pattern P is denoted by $\eta(P)$

Sometimes it is useful to know the total number of cells in a pattern. The number of cells in a pattern P is denoted by $\sigma(P)$. $\sigma(P) = \omega(P) \times \eta(P)$. Figure Ω.3 illustrates these properties.

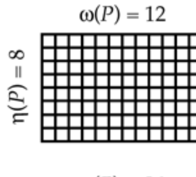


Figure Ω.3. Pattern Dimensions

Another important property of a pattern is the number of different colors it has. For a pattern P, the number of colors is denoted by $\kappa(P)$. For drawdowns, $\kappa(P) = 2$.

For "color drawdowns" (patterns in which the colors of the warp and weft threads are shown), $\kappa(P)$ may be greater than 2. Figure Ω.4 shows a 6-color pattern.

Figure 4. $\kappa(P) = 6$

$\kappa(P)$	number of colors

Geometrical Transformations

There are two kinds of geometrical transformations that can be performed on patterns: rotations and flips.

Rotations

A pattern can be rotated in increments of 90°: 90°, 180°, and 270°. Rotation by 360° leaves the pattern unchanged. Rotation is measured in the clockwise direction by convention.

The rotations of a pattern P by 90º, 180º, and –90º (270º) are denoted by ↴P, ↻P, and ↰P, respectively.

Figure Ω.5 shows the rotations of a square pattern and Figure Ω.6 shows the rotations of an oblong pattern.

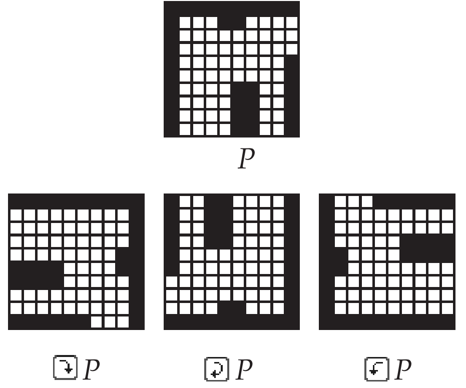


Figure Ω.5. Rotations of a Square Pattern

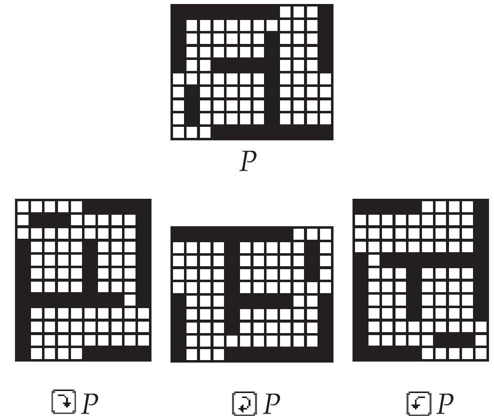


Figure Ω.6. Rotations of an Oblong Pattern

Note that:

$\eta(P) = \omega(↴P) = \omega(↰P)$

$\omega(P) = \eta(↴P) = \eta(↰P)$

$\eta(P) = \eta(↻P)$

$\omega(P) = \omega(↻P)$.

Flips

There are four flips:

- horizontal, around a vertical axis
- vertical, around a horizontal axis
- right, around the left diagonal (from the upper-left corner to the lower-right corner)
- left, around the right diagonal (from the upper-right corner to the lower-left corner)

These flips of a pattern P are denoted by ↔P, ↕P, ⤢P, and ⤡P, respectively.

Figure Ω.7 shows these flips for a square pattern and Figure Ω.8 shows them for an oblong pattern.

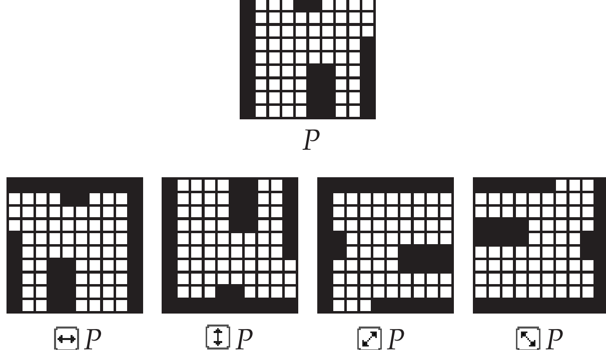


Figure Ω.7. Flips of a Square Pattern

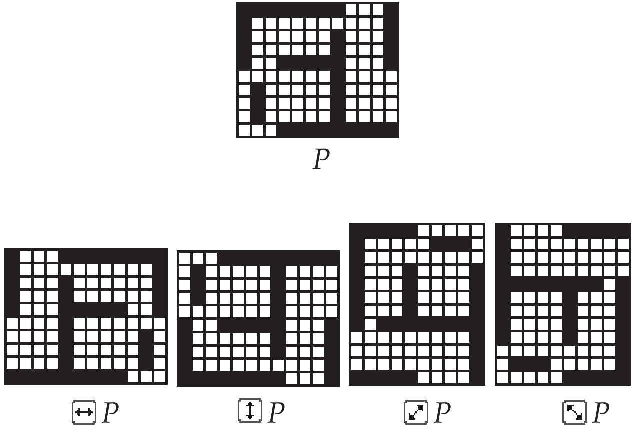


Figure Ω.8. Flips of an Oblong Pattern

05 Jul 2006 G3: PatternOperations

Note that:

$$\eta(P) = \eta(\boxed{\leftrightarrow}P) = \eta(\boxed{\updownarrow}P)$$

$$\omega(P) = \omega(\boxed{\leftrightarrow}P) = \omega(\boxed{\updownarrow}P)$$

$$\eta(P) = \omega(\boxed{\nearrow}P) = \omega(\boxed{\nwarrow}P)$$

$$\omega(P) = \eta(\boxed{\nearrow}P) = \eta(\boxed{\nwarrow}P)$$

Compound Operations

For rotations in increments of 90°, only one operation, $\boxed{\curvearrowright}P$, is needed. Applying it twice results in rotation by 180°, and applying three times results in rotation by 270°:

$$\boxed{\curvearrowright}\boxed{\curvearrowright}P = \boxed{\circlearrowright}P$$

$$\boxed{\curvearrowright}\boxed{\curvearrowright}\boxed{\curvearrowright}P = \boxed{\curvearrowleft}P$$

There also are relationships between rotations and flips. For example,

$$\boxed{\curvearrowright}\boxed{\nwarrow}P = \boxed{\leftrightarrow}P$$

In fact, all the geometrical operations can be obtained by using just $\boxed{\curvearrowright}$ and $\boxed{\nearrow}$ or by using just $\boxed{\nearrow}$, $\boxed{\leftrightarrow}$, and $\boxed{\updownarrow}$. For a discussion of these relationships, see Reference 2.

Although the relationships between the geometrical operations are interesting, for pattern construction, it's more convenient to have the whole set available.

Concatenation

Concatenation is the adjoining (juxtaposition) of two patterns to form a larger one. There are two forms of pattern concatenation: horizontal, in which patterns are adjoined at their vertical edges, and vertical, in which patterns are adjoined at their horizontal edges.

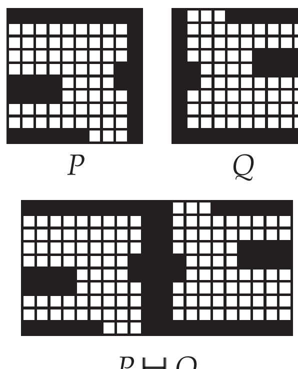
Horizontal and vertical concatenation are denoted by the symbols $\mapsto$ and I, respectively.

In order for concatenation to be possible, the adjoining edges must be of the same length:

For the horizontal concatenation of patterns P and Q, $\eta(P) = \eta(Q)$.

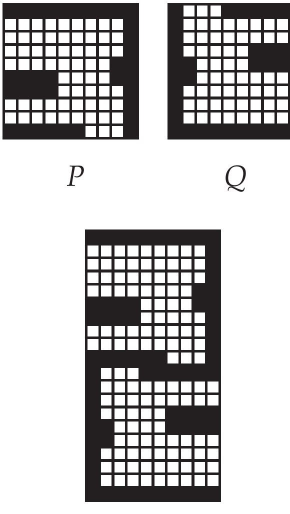
For the vertical concatenation of patterns P and Q, $\omega(P) = \omega(Q)$.

Figures Ω.9. and Ω.100 show examples of horizontal and vertical concatenation.



$P \vdash\!\dashv Q$

Figure Ω.9. Horizontal Concatenation



$P \,\mathrm{I}\, Q$

Figure Ω.10. Vertical Concatenation

Duplicate Edges

A potential design problem arises when the adjoining edges in concatenation are the same, cell by cell. This produces a duplication at the boundary, which may be undesirable for aesthetic and structural reasons. Therefore, if the adjoining edges are the same, one edge is discarded. Figure Ω.11 shows an example.

05 Jul 2006 G3: PatternOperations

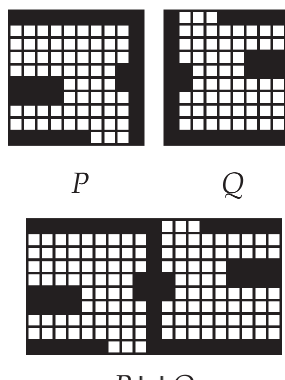


Figure Ω.11. Concatenation with Duplicate Removal

Duplicate removal is automatic in concatenation of patterns. If duplicate removal is not desired, the operations $\vdash\!\dashv_{+}$ and $\top\!\!\bot_{+}$ can be used.

Figure Ω.12 shows an example of horizontal concatenation without duplicate removal.

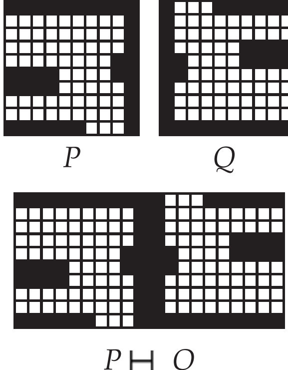


Figure Ω.12 Concatenation Without Duplicate Removal

Note: Duplicate edges are removed only at adjoining boundaries. Any other duplicate rows or columns are not affected.

Repetition and Extension

Repetition is a common way to extend a small pattern to make a larger one. Repetition is a special case of concatenation.

Like concatenation, repetition can be horizontal or vertical. The notation $P \mapsto i$ to denotes the repetition of P horizontally i times. Similarly, the notation $P \overline{\downarrow}\, i$ denotes the repetition of P vertically i times.

Figures Ω.13 and Ω.14 illustrate examples of these operations.

05 Jul 2006 G3: PatternOperations

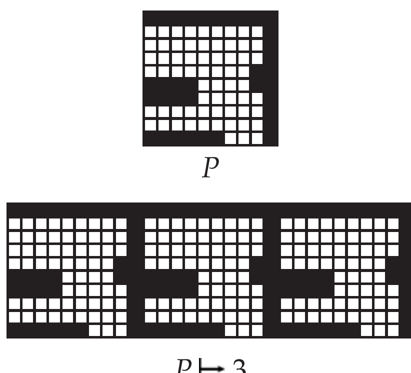


Figure Ω.13. Horizontal Repetition

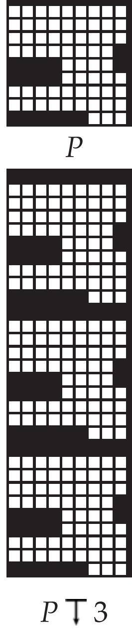


Figure Ω.14. Vertical Repetition

Duplicate Edges

As in concatenation, duplicate edges at boundaries are discarded by the repetition operations. Figure Ω.15 shows an example.

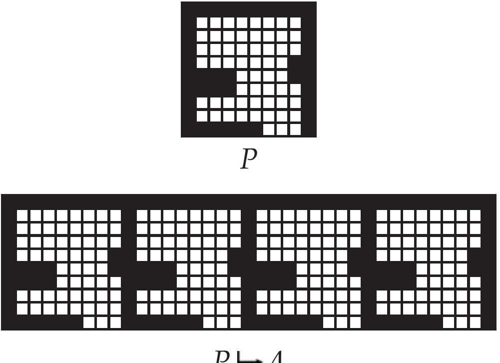


Figure Ω.15. Repetition With Duplicate Removal

Duplicate removal is automatic in the repetition of patterns. If duplicate removal is not desired, the operations $\mapsto_+$ and $\top_+$ can be used.

Figure Ω.16 shows an example of horizontal repetition without duplicate removal.

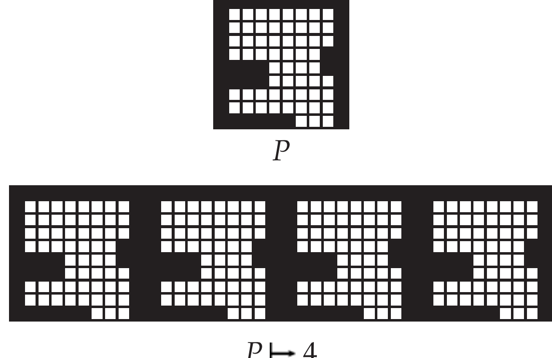


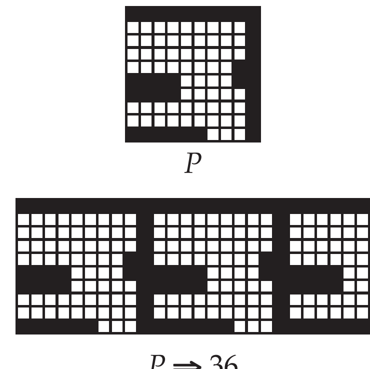
Figure Ω.16. Concatenation Without Duplicate Removal

Note: Duplicate edges are removed only at adjoining boundaries. Any other duplicate rows or columns are not affected.

Extension

Sometimes it is useful to extend a pattern by repetition to a width or height that is not an even multiple of that dimension of the pattern. The operations $P \Rightarrow i$ and $P \Downarrow i$ extend P by repetition to a total of i columns and i rows, respectively.

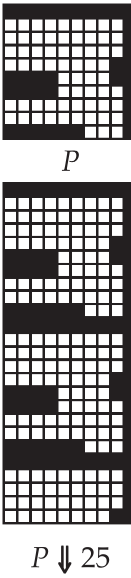
Figures Ω.17 and Ω.18 show examples of extension.



P

$P \Rightarrow 36$

Figure Ω.17. Horizontal Extension



P

$P \Downarrow 25$

Figure Ω.18. Vertical Extension

Note: If i is less that the dimension of the pattern in the given direction, the pattern is truncated at the right or bottom, accordingly.

As with repetition, duplicate edges at the boundaries of patterns are removed. The operations $P \Rightarrow_{+} i$ and $P \Downarrow_{+} i$ do not remove duplicate edges.

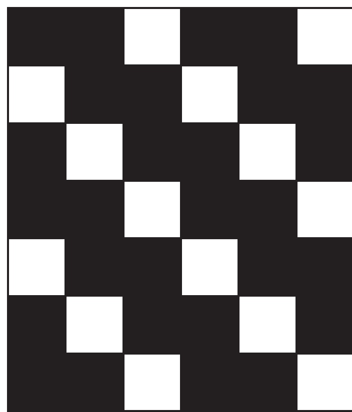
Summary of Operations

$\omega(P)$	width
$\eta(P)$	height
$\sigma(P)$	number of cells
$\kappa(P)$	number of colors
⟳ P	rotation by 90º
⤾ P	rotation by 180º
⟲ P	rotation by –90º
⬌ P	horizontal flip
⬍ P	vertical flip
⤢ P	right flip
⤡ P	left flip
⊢⊣	horizontal concatenation
⌶	vertical concatenation
⊢⊣$_+$	horizontal concatenation without duplicate removal
⌶$_+$	vertical concatenation without duplicate removal
↦ i	horizontal repetition
↧ i	vertical repetition
$\Rightarrow i$	horizontal extension
$\Downarrow i$	vertical extension
↦$_+$ i	horizontal repetition without duplicate removal
↧$_+$ i	vertical repetition without duplicate removal
$\Rightarrow_+ i$	horizontal extension without duplicate removal
$\Downarrow_+ i$	vertical extension without duplicate removal

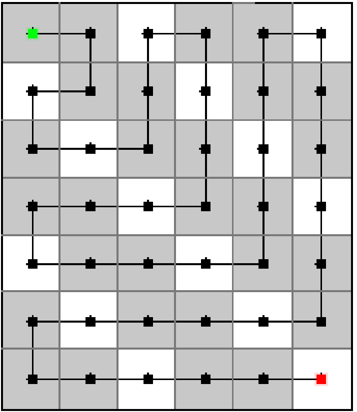
[More to come, but not yet written.]

Pattern Tours

The patterns considered here are black and white and represented by a rectangular grid of cells. Here is a typical pattern: [Redundant]



A sequence of cell locations is called a *path*. A path that includes every cell of a grid exactly once is called a *tour*. The focus here is on tours. Here is an example of a tour on the pattern shown above:

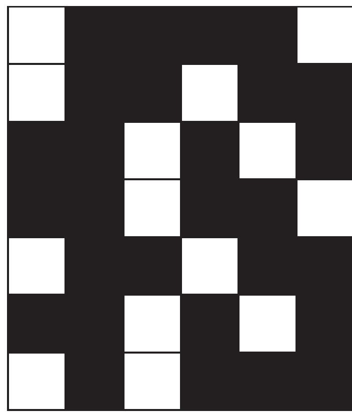


The green dot indicates the start of the tour and the red dot, the end.

The sequence of colors of the cells along a tour is called a *band*. Here is the band for the example above:

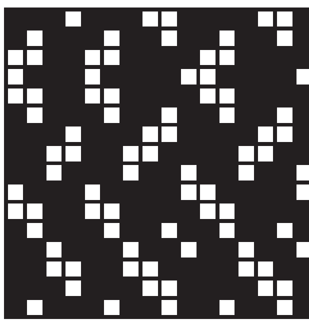
A tour and the corresponding band completely characterize a pattern in the sense that the tour and band can be used to construct the pattern.

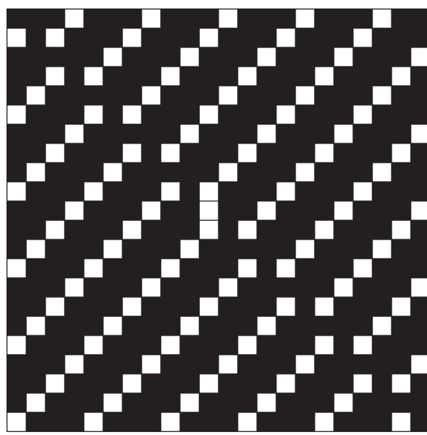
The process of producing a band from a pattern and a tour on it is called *reading out* the band. Conversely, given a blank grid, a tour, and a band, a pattern can be constructed by *reading in* the band. A band read out according to one tour and written in according to a different tour generally produces a different pattern. For example, here is the pattern that results from reading in the band show previously with the tour in reverse order:

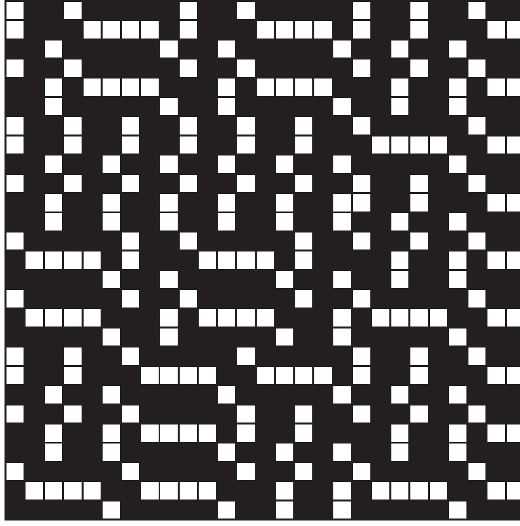


Tours and bands can be constructed independently of any pattern. This allows the possibility of constructing interesting and perhaps unexpected patterns.

Here are three patterns produced by tours and bands created mathematically but independently.







These are just some results from the first experiments. They are by no means the most interesting patterns that can be created by the methods described here.

Perspective

A tour serves to distribute the colors of the band over all cells in a grid.

The number of possible tours for all but trivially small grids is enormous. If there are k cells, there are k possibilities for the first cell on the tour. This eliminates one cell but leaves $k - 1$ possibilities for the second cell. Therefore the number of possible tours for a grid of k cells is $k \times (k - 1) \times (k - 2) \times \ldots 3 \times 2 \times 1 = k!$ (k factorial). For an 8×8 grid, there are 64 cells and the number of possible tours

[Sidebar on factorials.]

is 64!, which is

126,886,932,185,884,164,103,433,389,335,161,
480,802,865,516,174,545,192,198,801,894,375,
214,704,230,400,000,000,000,000

[Sidebar here or elsewhere — or in in appendix — on immense numbers]

Obviously, tours must be chosen with some plan or concept in mind, for a tour is the geometry from which the eventual pattern is crafted. A random tour is extremely unlikely to produce attractive results.

The problem of tour design is complicated by the fact that it alone does not determine the pattern; the band plays a strong role in the final result. This makes tour design challenging and interesting.

There are certain things that can be looked for in tours. One is some kind of pattern in the tour itself. A jumbled, chaotic tour, even if far from random, is unlikely to produce attractive results *unless* the band is developed along with the tour. This is possible — take any pattern and any tour, however disorganized, and read out the band. This band, when read in by the tour, will reproduce the original pattern. Some uses for tours and bands produced in this way are described in a subsequent section, but the first concern is designing tours independently of bands.

While designing tours and bands independently requires both the application of some principles and some serendipity, it is the core of a process for getting attractive and unusual patterns.

If tours are taken alone, their design needs to be guided so that the tours themselves are attractive and interesting.

Tour Design

There are several properties to keep in mind when designing tours.

Symmetry: Symmetric designs are attractive to human beings (although no one knows exactly why). Various kinds of symmetries are possible for tours.

Repetition: Repetition of a unit within a design can be useful in tours just as it is in tilings, weaves, and other kinds of artistic constructions.

Variety: A little variety or an element of surprise can break an otherwise monotonous design and be aesthetically pleasing.

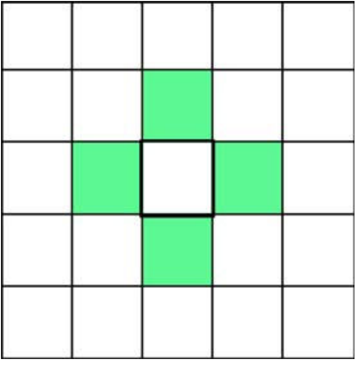
Continuity: Since a tour is a path in the general sense, there is some value in continuity. For example, the locations on a tour may be adjacent (their cells sharing a side). There are various kinds of continuity and technical names for them. The problem is discussed in another section in the context of constraints.

Tour Constraints

Constraints can be useful in design; they limit what is possible in a systematic way that prevents accidental aberrations, and they provide for a certain degree of regularity.

The kind of constraints here are local ones that allow tours to be built step-by-step. Such constraints might limit the distance from one location on the tour to the next or limit the possible directions in which the next location may lie.

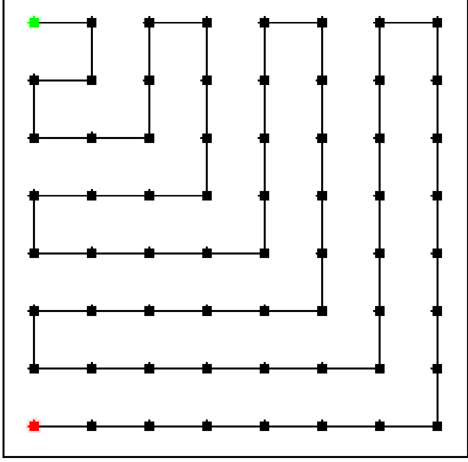
The neighborhood concept from cellular automata [2] is a useful model for this kind of constraint. For example, the von Neumann neighborhood looks like this:

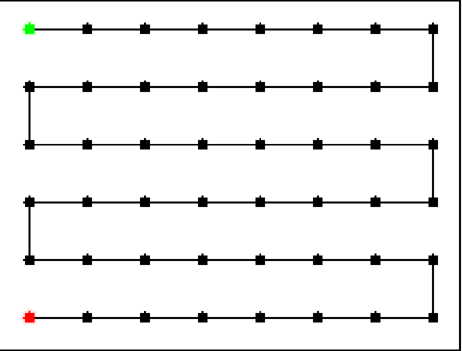


Viewed as a constraint, this neighborhood requires the location following the central location to be one cell away, horizontally or vertically. Staying put is not an option, since a location cannot appear twice in a tour. Locations that are off the grid, when a cell is at an edge, obviously are excluded. Similarly, locations that already are on the tour are forbidden.

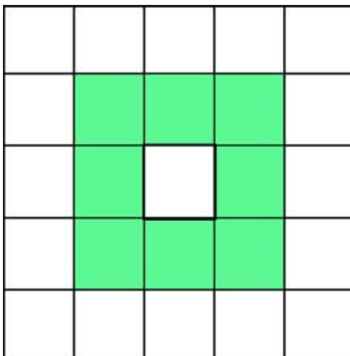
We'll call a tour constructed using this neighborhood a *von Neumann tour*. Von Neumann tours have a special kind of continuity, called *unicursal* in graph theory. Von Neumann tours also are *planar*, meaning there are no crossings on a line drawn along the tour [?].

Here are examples of von Neumann tours:

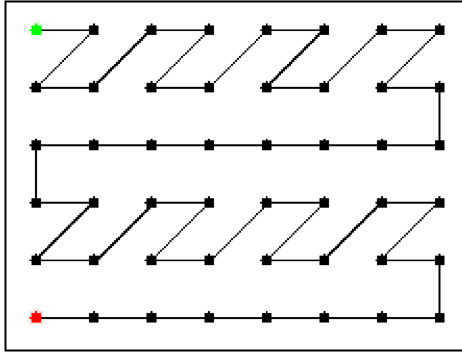


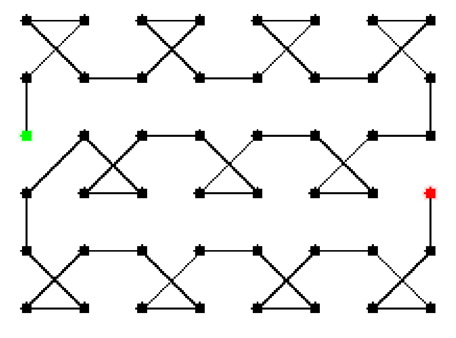


The Moore neighborhood allows more freedom of movement:

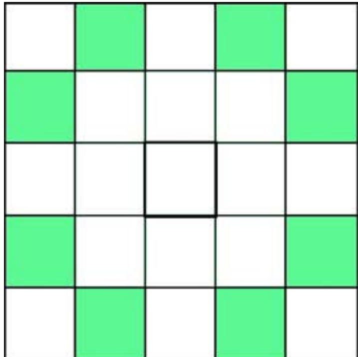


Moore tours include von Neumann tours as a subset, but they allow considerably more variety, including diagonal moves and non-planar tours. Here are some examples of Moore tours that are not von Neumann tours:



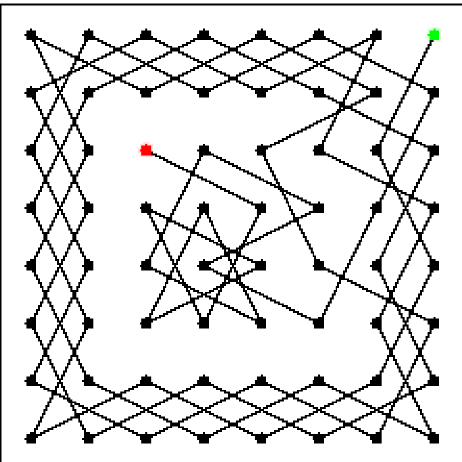


The legal moves of chess pieces can be described as neighborhoods. For example, the knight has the neighborhood



Note that the knight "jumps"; it cannot move to an adjacent cell.

Knight's tours are sufficiently interesting and difficult to design that they have occupied the attention of chess players and mathematicians for centuries. Here is an example:

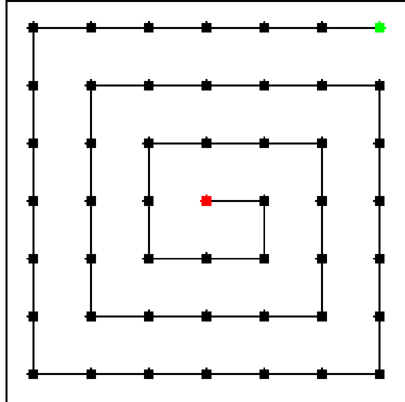


Neighborhoods provide constraints that limit the nature of tours. In constructing tours using neighborhoods, there are, of course, choices. One thing that can happen is getting into a situation in which no further move is possible. There are ways of dealing with this problem, which we'll discuss in a later section.

Tour Classification

Tours can be classified roughly according to the methods by which they can be constructed and places they can be found. In many cases, a tour will fit into more than one category, depending on how it is created or viewed.

Algorithmic Tours: These tours are constructed according to a fixed set of rules applied in a well-defined fashion. An example of an algorithmic tour is a square spiral:

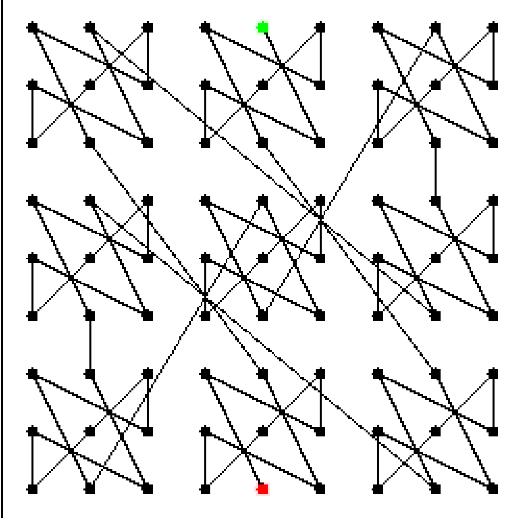


Note that this also is a von Neumann tour.

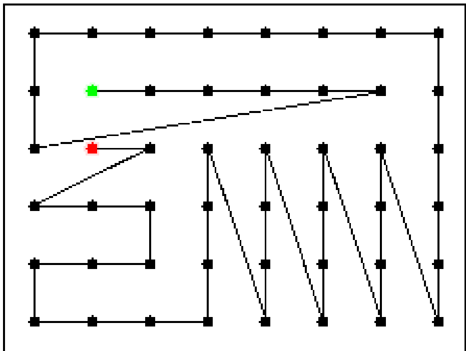
The geometry of this kind of tour is obvious, as are possible variations on it. You might find it illuminating to devise an algorithm that produces square-spiral tours.

Neighborhood-Constrained Tours: These tours are discussed above. They come in great variety and will be the subject of subsequent articles.

Numerically Derived Tours: These tours are derived from numerical problems and puzzles that are not directly related to tours but that nonetheless can be interpreted as tours. An example is this tour derived from a magic square, in which the sums of the rows, columns, and main diagonals are all equal:



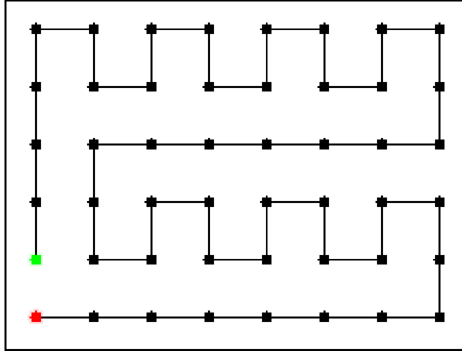
Miscellaneous: There is the inevitable "other" category containing tours that do not fit elsewhere. Here is an example of a tour that was constructed by hand:



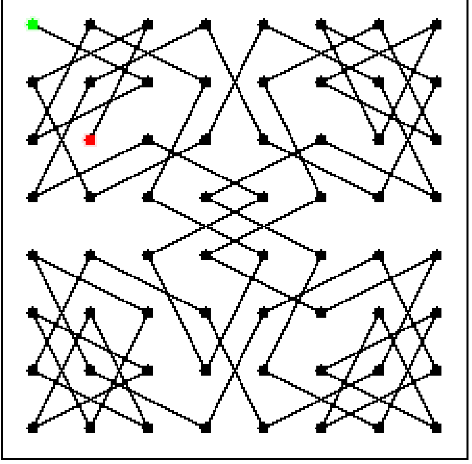
More Terminology

There are a few other terms related to tours that are important in some contexts:

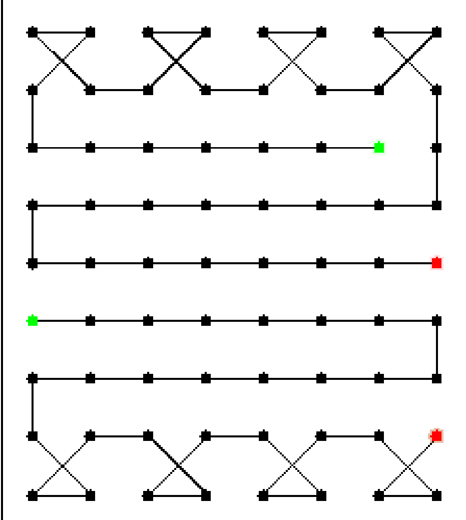
Re-Entrant Tours: A tour whose last location is within a legal move of its first location is called *re-entrant*. An example of a re-entrant von Neumann tour is:



Here is an example of a re-entrant knight's tour:



Piece-Wise Tours: A piece-wise tour is one composed of parts that satisfy some condition, but the whole does not. Here is an example of a piece-wise Moore tour:



Such tours can be made into regular tours by connecting the end of one piece to the beginning of the other, but it often is more useful to view the parts independently.

Incomplete Tours: A path that does not include every location in a grid is called *incomplete* or *open*. Incomplete tours can be used for components of piece-wise tours or alone in some applications.

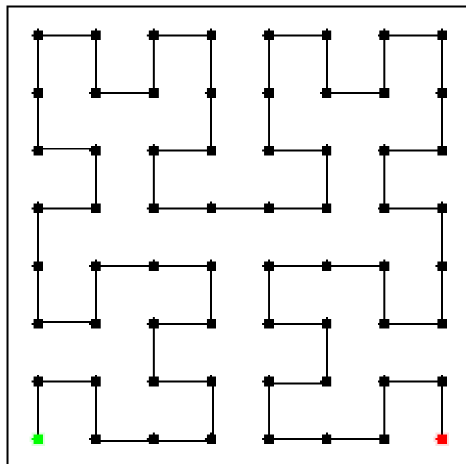
Overtours: A path that includes a location more that once is called an overtour. Overtours also have their applications, which are discussed in another article. [Watch references to non-existent sections, which if not written, should be turned into subjects for further study.]

Graphical Representations

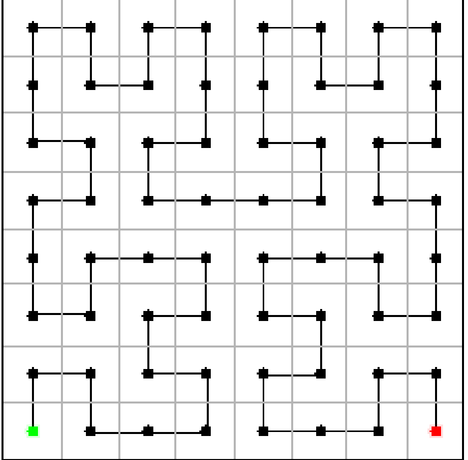
In order to deal with tours, both representations for understanding them and representations for creating and manipulating them are needed.

So far pattern tours have been represented by lines drawn though the cells in the order of traversal. The beginning of a tour was shown in red and the end in green for two reasons: for ease in identifying these important locations and for establishing the direction of transversal.

This method works well for many tours in giving an overall impression of their geometries. Here is an example:



Grid lines may help identify specific locations:



If, however, the tour is non-planar, line crossings may make the tour difficult to follow. And if the tour is chaotic and has many criss-crossing jumps, such a graphic representation may be a useless jumble:

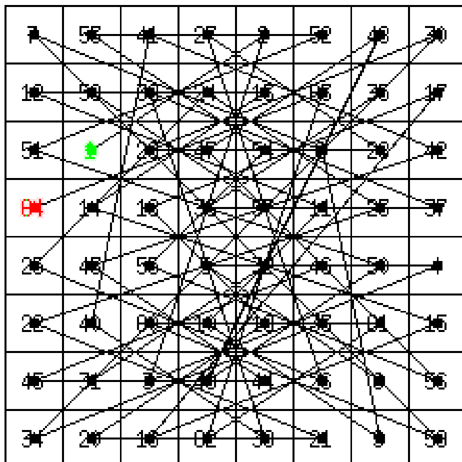
This representation also is ambiguous. There is no way to tell if a line through several cells includes all the cells on the line or jumps over some.

An alternative graphical method is to dispense with the lines and assign numbers to the cells that specify the order of traversal. Again, the ends can be highlighted by colors:

7	53	41	27	2	52	48	30
12	58	38	24	13	63	35	17
51	1	29	47	54	8	28	42
64	14	18	36	57	11	23	37
25	43	55	5	32	46	50	4
22	40	60	10	19	33	61	15
45	31	3	49	44	26	6	56
34	20	16	62	39	21	9	59

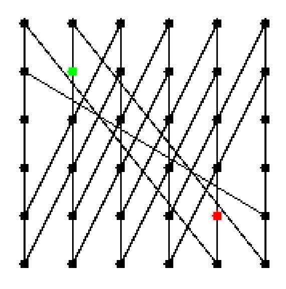
This numerical representation has the virtue of being precise and unambiguous. It may be difficult, however, to locate successive cells on a tour.

An alternative that is sometimes used is to overlay the line and numbered grid representations as in:



Except for small, simple tours, the result may be worse than either of the two forms separately. Both numbered grids and line drawings in situations are needed where a clear understanding of a tour is required. An example is:

28	10	31	13	34	16
19	1	22	4	25	7
29	11	32	14	35	17
20	2	23	5	26	8
30	12	33	15	36	18
21	3	24	6	27	9



Data Representations

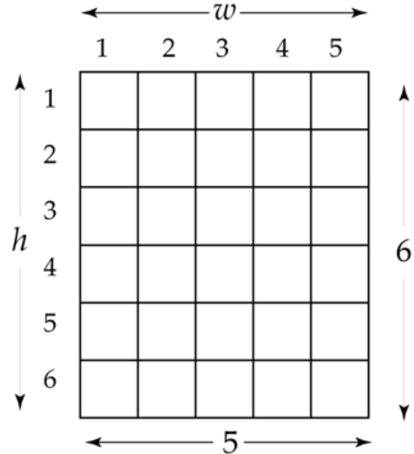
In order to write procedures for manipulating tours, data representations that a computer program can use are necessary. These are very different from the graphical representations that human beings find easy to understand.

Coordinate Lists

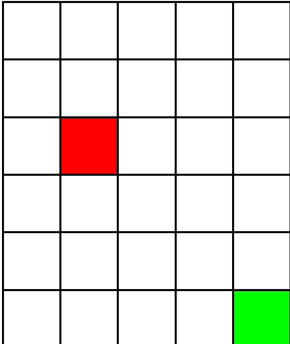
One data representation of a tour as a list of cell locations. For this, A systematic way of identifying locations on a grid is needed.

[Duplicates material given elsewhere.]

The dimensions of a grid in cells are given by two numbers, w for the width in cells and h:

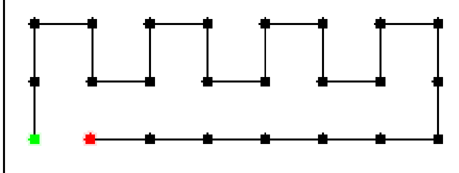


Rows and columns are numbered as indicated. The location of a cell is given by its coordinates, which is a pair (i, j) in which i is the column number of the cell and j is the row number of the cell. For example, in



the coordinates of the red cell are (2, 3) and the coordinates of the green cell are (5, 6).

A tour then can be represented by a list of coordinates in the order of traversal. For example, the tour



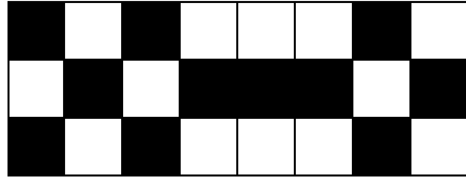
has the coordinate list

(1, 3), (1, 2), (1, 1), (2, 1), (2, 2), (3, 2), (3, 1),
(4, 1), (4, 2), (5, 2), (5, 1), (6, 1), (6, 2), (7, 2),
(7, 1), (8, 1), (8, 2), (8, 3), (7, 3), (6, 3), (5, 3),
(4, 3), (3, 3), (2, 3)

To apply a band to a tour represented by a coordinate list, it is only necessary to pair the coordinate list with the band and place the colors of the band, in order, at the designated succession of locations. For example, reducing the type size of the coordinate list for the tour above and drawing a band below it provide a complete specification for a pattern:

(1,3),(1,2),(1,1),(2,1),(2,2),(3,2),(3,1),(4,1),(4,2),(5,2),(5,1),(6,1),(6,2),(7,2),(7,1),(8,1),(8,2),
(8,3),(7,3),(6,3),(5,3),(4,3),(3,3),(2,3)

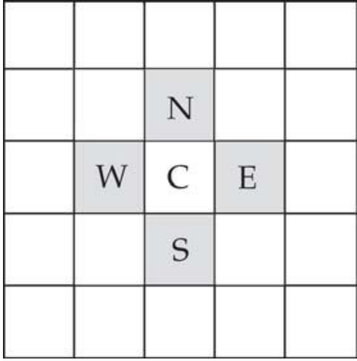
The resulting pattern is:



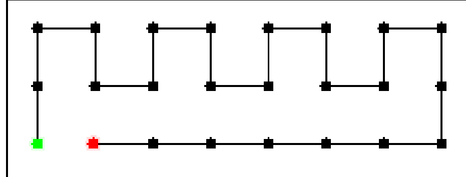
Navigational Representations

Methods for describing tours, which are particularly useful in some constructing some kinds of tours, use navigation rather than specific coordinates.

For von Neumann paths, a list of compass points that specify the direction of one cell to the next is sufficient. Given the neighborhood labels in terms of compass points, as in



the cell following C can be indicated by a letter that corresponds to the direction of movement. For example, for the tour

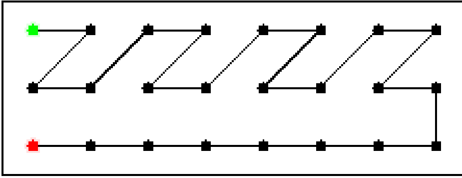


the sequence of directional moves is given by the string of letters

NNESENESENESENESSWWWWWW

Such a string and the location of the starting cell completely characterize von Neumann tours and in a much more compact way than coordinate lists.

As given above, direction strings are limited to von Neumann tours. There is, for example, no way to specify a diagonal move. The concept of direction string could be extended to include Moore tours by adding letters for the diagonal moves. A more general method, that can be used for all tours, is to provide a way for specifying passing over cells without including them on the tour. We'll use lowercase letters for this: nesw in addition to NESW. For example, the Moore tour



is described by the direction string

EsWEeNEEsWEeNEEsWEeNEsWES
WWWWWWW

Of course sW is equivalent to wS, and so on.

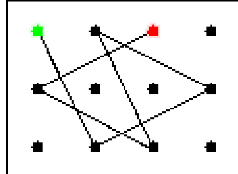
Using this navigational method, it is not necessary to specify the starting cell. Assuming the string starts at the upper-left corner, the first location on the tour can be reached by a string of nesw moves at the beginning.

Although this method is completely general, it is cumbersome for tours in which there are long jumps.

In cases of specialized neighborhoods, such as the knight's, a customized navigational system can be used. Here's the knight' neighborhood with the cells that can be reached lettered clockwise around the knight:

	H		A	
G				B
		K		
F				C
	E		D	

For this fragmentary tour



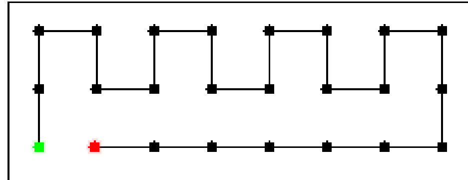
the navigational string is DBGDGB.

There are many possible navigational alternatives to these kinds of navigational strings. An attractive one is to use L-System notation in which the symbols used are interpreted to draw images [3]. Adapted to describing tours, these symbols are:

F	move forward, including cell on tour
f	move forward, not including cell on tour
+	turn right 90°
–	turn left 90°

The initial direction is east and that the starting point is the upper-left corner.

For example, the von Neumann tour shown earlier:

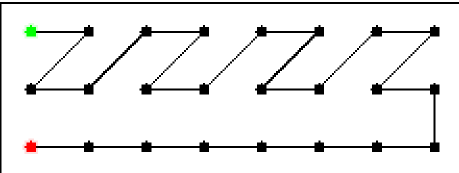


can be represented by the L-System string

+ffFF+F+F–F–F+F–F–F+F–F–F+FF+FFFFFF

The initial +ff gets to the first location on the tour.

The Moore tour shown earlier



can be represented by

F+f+Ff–F+f+Ff–F+f+F+f–F+f+F++F+
fFFFFFF

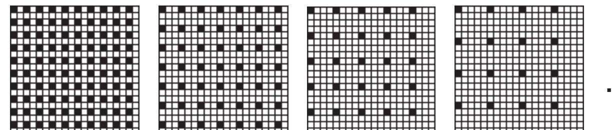
Again, diagonal moves can be represented in different ways. For example, f–F produces the same result as–fF.

An advantage of using the L-System method is that there is a large body of knowledge associated with them. And this material includes some interesting tours.

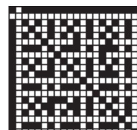
Grid Overlay Patterns

In his book, *A New Kind of Science*, Stephen Wolfram describes a system for constructing patterns by overlaying grid of cells with increasing separation.

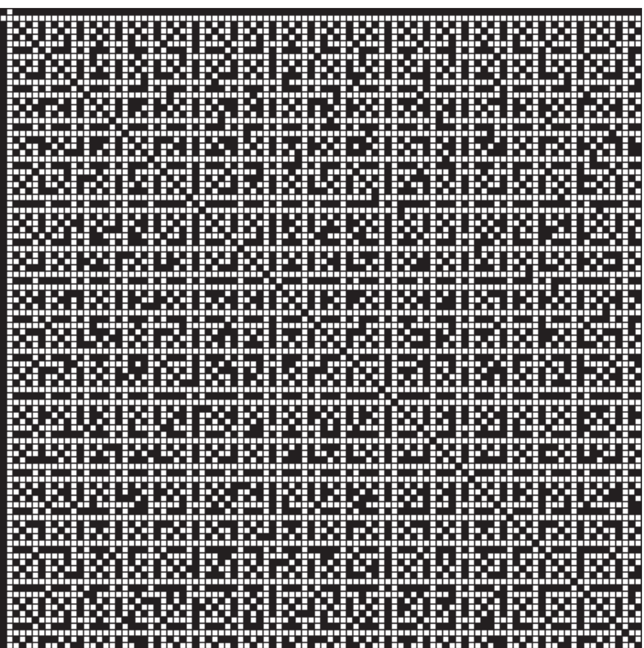
The basic scheme uses grids of black cells separated by rows and columns of white cells. In the first grid, the black cells are separated by single columns and rows of white cells. In the next grid, the separation is by two columns and rows, then three, and so on:



If these grids are overlaid so that black cells show through (logical *or* if black represents *true* and white *false* [2]), the result looks like this:



For larger grids, the results are even more intricate:



While these patterns are not suitable a drawdowns *in toto* because of floats, portions of them are.

There are many possible generalizations to the basic scheme described above. Here are a few:

- using a motif more complicated than a single black cell
- using different motifs for successive grids
- varying the horizontal and vertical separations between the motifs in various ways
- combining successive overlays in different ways, not just with logical *or*

One version of a generalized system is based on sequences that apply to successive grids:

- motifs
- width separations
- height separations
- logical combination operations

The basic system described at the beginning of this article is characterized by the sequences

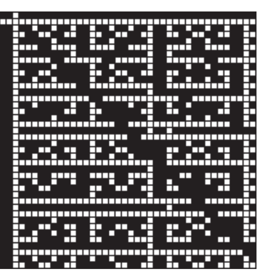
motifs: ■, ■, ■, ■, ■, …
width: 1, 2, 3, 4, 5, …
height: 1, 2, 3, 4, 5, …
operations: +, +, +, +, +, …

As usually happens with such generalizations, there is a vast (in fact, infinite) number of possibilities. With some exploration, it may become clear what kinds of possibilities lead to interesting results.

One way to start is to depart from the basic scheme one way at a time. For example, if a different motif is used, but only one, as in

motifs: ■■, ■■, ■■, ■■, ■■, …

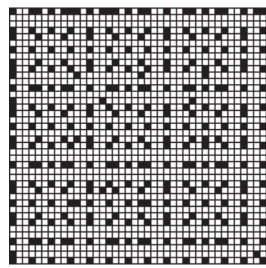
the result is:



Another simple change is to use prime numbers for the separations:

width: 2, 3, 5, 7, 11, …
height: 2, 3, 5, 7, 11, …

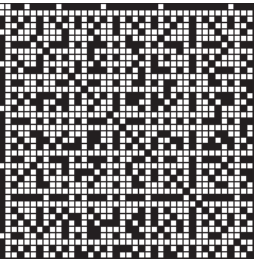
the result is:



Changing the operation to *exclusive or*, ⊕, so that

operations: ⊕, ⊕, ⊕, ⊕, ⊕, …

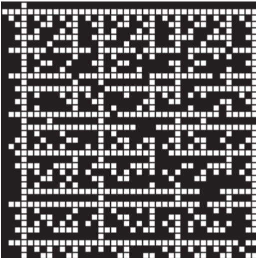
produces



All the changes so far are simple and uniform. Here's one that is a bit more complex:

motifs: ■, ■■, ■■■, ■■, ■, …

where the sequence of motifs shown is repeated. The result is:



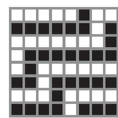
Exercises and Areas for Further Study

What happens if some of these variations are tried in combination? What if they are more varied and complex? Are there schemes that produce results that are more interesting than other schemes?

Line-Based Patterns

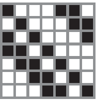
The term line is used to cover both rows and columns of grid-plot patterns. Grid plots are shown in row order, but columns and rows are equivalent for the subject here.

As such, a line can be represented by a binary sequence of 0s and 1s corresponding to white and black grid cells, respectively. For example, the fifth row in the following pattern has the line sequence 01000000.



Many interesting and important weave structures can be made from patterns in which one line serves as a *basis line* and each subsequent line is derived from the previous line by the application of a *transformation rule*. Such patterns are called *line-based patterns*.

For example, in twills, each line after the basis line is produced by a cyclic permutation by one of the preceding line:



Transformation Rules

Transformation rules can be of many kinds. The simplest ones are

cyclic permutation

reversal

complementation (exchange of 0s and 1s)

and combinations of these.

For example, given the line

00111011

its cyclic permutation by 3 is

01100111

Positive values are to the right, negative values to the left. Positive values are sufficient, since there is always a positive value that produces the same results as a negative one. For example, –3 for the line above is equivalent to 5.

The reversal of the given line is

11011100

and its complement is

00100011

Binary Indices

The binary index of a line is the decimal equivalent of the line considered as a binary number.

For example, the binary index for 00111011 is 59.

The combination of a line's length and binary index uniquely characterizes the line. The line length is needed because leading zeros are lost in computing binary indices. For example, 00111011, 111011, and 000000111011 have the same binary index.

The line at the beginning of this section is uniquely identified by 8:59; length 8, binary index 59.

Design-Equivalent Patterns

Various transformations of a pattern are *design equivalent* in the sense that they can be derived from each other by simple transformations [1]: rotation in 90º increments, flips (horizontal, vertical, and around the diagonals), cyclic permutation of rows or columns, complementation, and any combination of these.

Patterns with repeats also are design equivalent to their unit patterns. [definition?]

Design-Equivalent Basis Lines

For line-based patterns, several different basis lines can produce design-equivalent patterns, which are essentially redundant and need to be avoided. A *fundamental basis line* needs to be selected from the alternatives.

The transformation rules listed above are the ones that produce design-equivalent lines.

Given a line, its basis line can be obtained by applying the transformation in all possible ways and selecting the one with the smallest binary index.

Here is an example of how this can be done. Consider this line

11101100

Its cyclic permutations are

01110110
00111011
10011101
11001110
01100111
10110011
11011001

The one with the smallest binary index is the second, 00111011. (This can be determined by inspection without having to produce all the permutations. For very long lines, the permutations can be sorted as numbers to find the one with the smallest binary index.)

The chosen line now is the basis for applying other transformation. It is not necessary to do this for all the permutations: The results would be the same.

The next transformation is reversal, which produces

11011100

Taking the permutations and selecting the one with the smallest binary index produces

00110111

Since its binary index is smaller than the one for the permutations of the original line, it's the one to which the next transformation, complementation, is applied:

11001000

The cyclic permutation of this line that gives the smallest binary index is

00011001

The final transformation is to reverse it to give

10011000

Its cyclic permutation with the smallest binary index is

00010011

so this is the one to use for the basis line. This is the fundamental basis line and has the unique identification 8:19.

For what it's worth, the binary index of the original line is 236.

The Number of Fundamental Basis Lines

Here is a list of the number of fundamental basis lines up to $n = 16$:

n	*lines*
2	1
3	1
4	2
5	3
6	5
7	8
8	14
9	21
10	39
11	62
12	112
13	189
14	352
15	607
16	1144

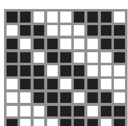
Soundness

There is no guarantee that line-based patterns will produce interlacements that "hang together" if drafted by the conventional draw-up techniques [2].

This issue must be addressed separately.

Shift Patterns

The simplest kind of line-based pattern involves successive cyclic permutations (shifts) by a fixed amount. Twills, with a shift of 1, are the most familiar of these patterns:



For a basis line of length n, all shifts from 1 to n–1 produce patterns (shifts of 0 and n simply repeat the basis line), but it is not necessary to consider all these shifts.

A shift of 1 can be omitted, since the result is a twill and well known. Shifts that are greater than $(n–1)/2$ produce patterns that are design-equivalent [1] to those for shifts that are less than or equal to $(n–1)/2$.

Finally to get square $n \times n$ patterns, a shift must be relatively prime to n; that is no number other than 1 (and n) must evenly divide n and the shift. Otherwise the vertical repeat will be less than n. While such patterns have potential interest, the focus here is on $n \times n$ patterns. Note that for n prime, all shifts are relatively prime to n. [sidebar? reference?]

Here is a list of shifts up to $n = 16$ that meet the requirements above:

n	*shifts*	*number*
2		0
3		0
4		0
5	2	1
6		0
7	2, 3	2
8	3	2
9	2, 4	2
10	3	1
11	2, 3, 4, 5	4
12	5	1
13	2, 3, 4, 5, 6	5
14	3, 5	3
15	2, 4, 7	3
16	3, 5, 7	3

Note that there are no shift patterns for $n = 6$ (as is the case for satins, and for the same reason).

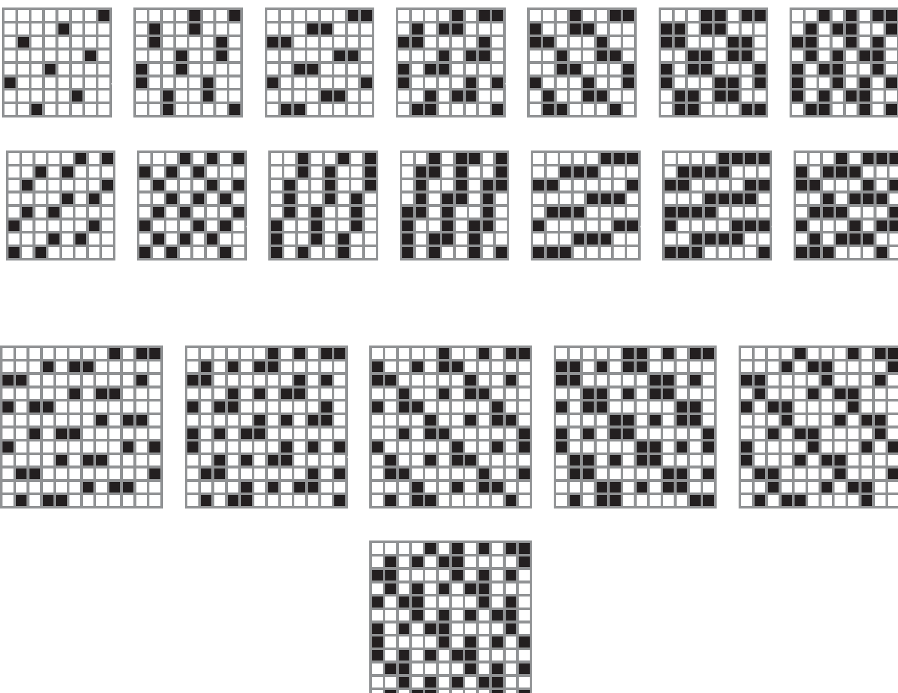
Combining the number of basic lines with the information on shifts gives the number of shift patterns:

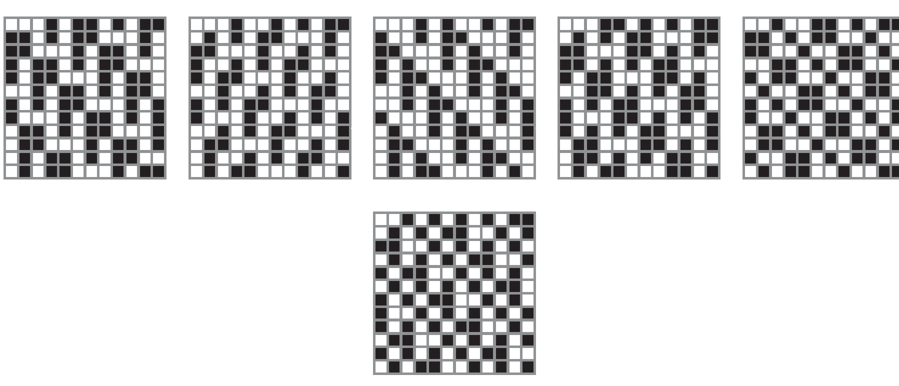
n	*patterns*
2	0
3	0
4	0
5	3

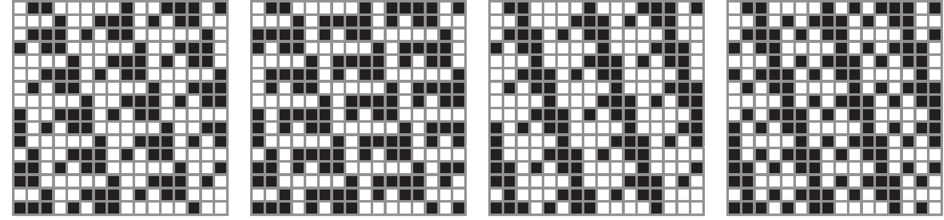
6	0
7	16
8	28
9	42
10	39
11	248
12	112
13	945
14	704
15	1821
16	3432

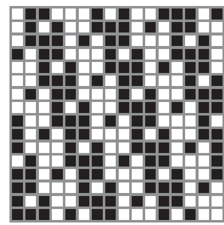
As a final note, all the shift patterns that meet the criteria above produce structurally sound interlacements if drawn up in the conventional manner — they "hang together" [2].

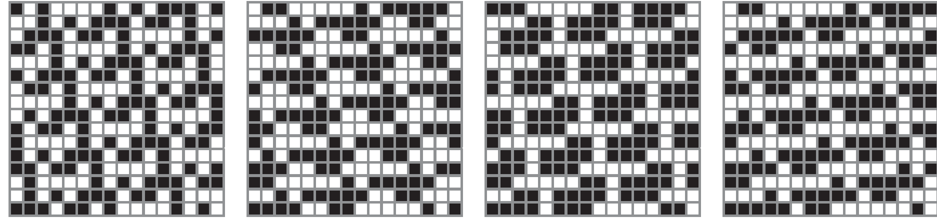
Here are some examples of sift patterns: [The layout needs to be fixed.]

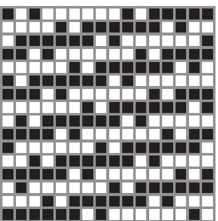










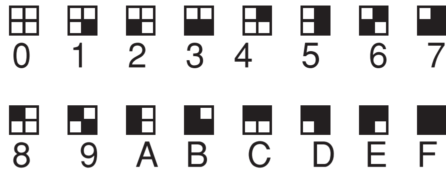


Pantactic Squares

The material that follows in based on a paper in a mathematical journal and a Web page [1-2].

A 5×5 pantactic square is a 5×5 two-color grid pattern in which every 2×2 subpattern is different.

There are $2^4 = 16$ 2×2 patterns:



The identifying labels are obtained by following the rows left to right, taking white cells to be 0 and black cells to be 1. The resulting binary number is converted to hexadecimal.

Here is an example of a 5×5 pantactic square:

The 2×2 subpatterns from left to right, top to bottom are: DB247E81FA05C936. This string uniquely identifies this pantactic square, although there are more concise ways of doing this.

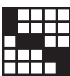
Although there are $2^{25} = 33{,}554{,}432$ 5×5 two-color grid patterns, there are only 800 essentially different 5×5 pantactic squares. These 800 squares can be grouped into 16 categories according to common structural properties.

Properties of 5×5 Pantactic Squares

5×5 pantactic squares have some surprising properties, especially lack of symmetry. While many patterns are the same after some kind of rotation or refection, 5×5 pantactic squares are not: No combination of rotations, reflection, (horizontal, vertical, or diagonal), or color inversion of a 5×5 pantactic square produces the same 5×5 pantactic square.

Another property of 5×5 pantactic squares is a limitation of *connected paths* they can contain. A connected path in a grid pattern is a sequence of cells of the same color, all of which share an edge. For example, in

the black cells form a connected path, while in



they do not, because in two places they connect only at corners.

A 5×5 pantactic square cannot have a connected path that reaches from one edge to the opposite one.

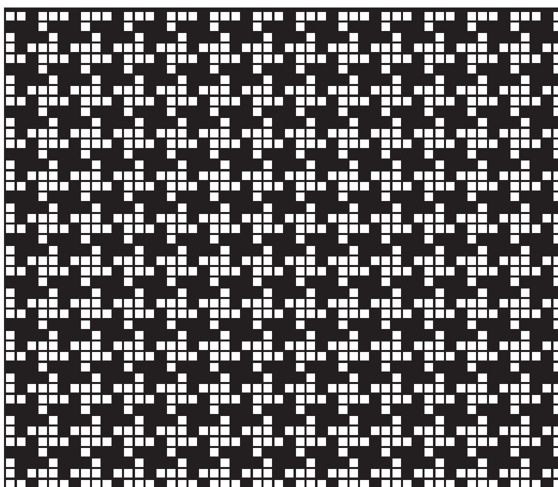
A *connected block* is a collection of cells of the same color in which the cells share edges. An example of a connected block is

The maximum number of connected cells in a connected block in a 5×5 pantactic square is eight.

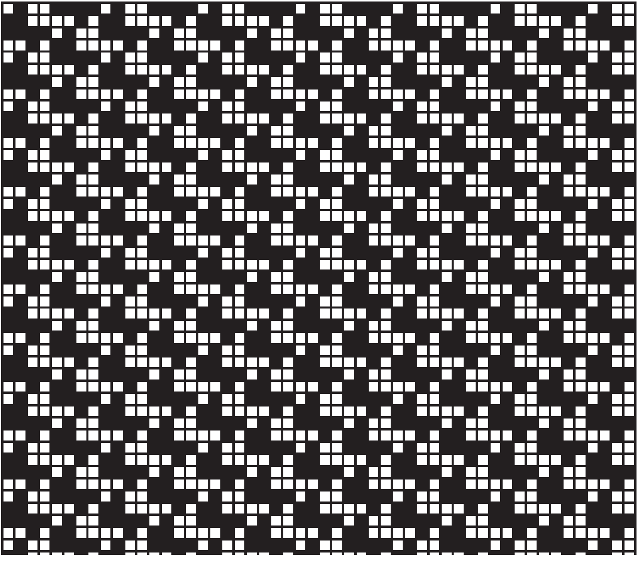
There are 50 essentially different *basic blocks* in 5×5 pantactic squares. Here are four of them:

Pantactic Patterns

Some basic blocks can be arranged in ways that form repeating patterns of arbitrarily large size, all of whose 5×5 subpatterns are pantactic squares. Two examples are:



The arrangement here is a regular tiling.



The arrangement here is a tiling with a three-row offset.

Generalizations

There are other questions to be addressed. The literature on pantactic squares seems to be limited to 5×5 ones. Are there larger ones? Are there non-square pantactic designs? What about more than two colors?

Introduction

Sequences play a central role in drafting: the threading, the treadling, and warp and weft color sequences.

Since shafts and treadles are numbered, integer sequences from mathematics fit in naturally. Numbers can be associated with colors, as in black = 1, white = 2, red = 3, and so on.

Why integer sequences with mathematical origins? Because they abound in patterns — some of the most beautiful and intricate patterns known.

This chapter explores a number of integer sequences and shows some kinds of weave structures that can be found in them. Some of the integers sequences explored here are simple and well known, like squares, cubes, and so on. Some are esoteric and whose origin are too deep to explain here.

Perhaps the most fascinating sequences of all are fractal sequences. Since they are just sequences, their beauty is hidden, unlike the spectacular color fractal images we are used to seeing. But the beauty is there — to be discovered.

This chapter make no attempt at comprehensive coverage — that would, in fact, be impossible. It merely suggests — both on how integer sequences can be used in weave design and where to look for other promising sequences.

Residue Sequences

I don't like nonrepeating decimals. Pi makes me furious.
— Don DeLillo, *Ratner's Star*

Introduction

Most integer sequences that come to mind — such as the integers, the squares, the cubes, the Fibonacci numbers, and the primes — have terms that get larger and larger.

For most design purposes, values need to be limited to a fixed range, such as the number of shafts or treadles available.

The most natural way to bring an integer sequence into a fixed range is to use modular arithmetic — to use the remainders or *residues*[*] of the terms on division by a specified modulus.

For example, the sequence of the cubes

1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331,
1728, 2197, 2744, 3375, 4096, 4913, 5832, 6859, …

has the following residue sequence mod 10:

1, 8, 7, 4, 5, 6, 3, 2, 9, 0, 1, 8, 7, 4, 5, 6, 3, 2, …

Periodicity

In addition to bringing sequences into fixed ranges, residues often reveal underlying patterns. The most distinctive pattern is periodicity, in which a fixed number of terms repeats indefinitely. For example, the residue sequence for the cubes mod 10 has period 10 with the repeat:

1, 8, 7, 4, 5, 6, 3, 2, 9, 0

(The fact that the modulus and the period are the same in this case is not a coincidence, but this relationship does not hold in general.)

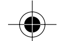
Some residue sequences have *pre-periodic* parts before the period begins. An example is the sequence of digits in the decimal expansion of

1/77760

which has the pre-periodic part

0, 0, 0, 0, 1, 2

before settling down to the repeat



24 Jun 2006 H2: ResidueSequences

8, 6, 0, 0, 8, 2, 3, 0, 4, 5, 2, 6, 7, 4, 8, 9, 7, 1, 1, 9,
3, 4, 1, 5, 6, 3, 7

Periodic sequences without pre-periodic parts are called *purely periodic*.

Although many residue sequences are periodic, some are not. For example, residue sequences for the primes are not periodic for any modulus greater than 2. Other residue sequences are periodic only for certain moduli. Furthermore, some residue sequences appear to the eye to be periodic but are not. Examples are the residues of the two Wythoff sequences [1]:

$a_n = \lfloor n \times \phi \rfloor$ lower

$a_n = \lfloor n \times \phi^2 \rfloor$ upper

where ϕ is the golden ratio, 1.6180339887 ... and $\lfloor x \rfloor$ is the floor of x, the largest integer less than or equal to x.

For example, the residue sequence for the lower Wythoff sequence mod 8 is:

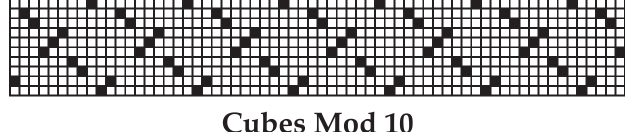
1, 3, 4, 6, 0, 1, 3, 4, 6, 0, 1, 3, 5, 6, 0, 1, 3, 5, 6, 0, 1,
3, 5, 6, 0, 2, 3, 5, 6, 0, 2, 3, 5, 7, 0, 2, 3, 5, 7, 0, …

I'm inclined to leave the "we" approach below intact. What do you think?

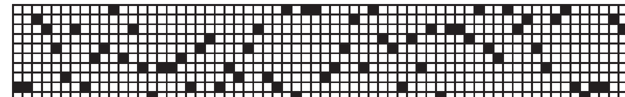
Repeating Patterns

The concept of pattern is familiar to everyone, but a precise meaning is elusive and often depends on context. Nonetheless, our intuitive concept of pattern serves fairly well in practice. We see repetition, as in a periodic sequence, as a pattern. We recognize various symmetries as patterns. And, in general, we perceive order as different from chaos. Of course, not all patterns are attractive.

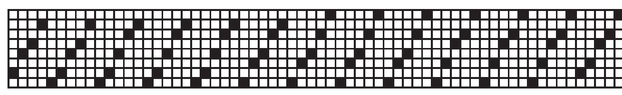
With effort, we can detect patterns in relatively short sequences of integers, but patterns are much easier to detect if the integers are shown by magnitude, as in grid plots. Here are some grid plots of residue sequences in which the bottom row is 0 and the top row is the modulus minus 1.



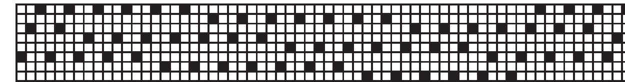
Cubes Mod 10



Fibonacci Numbers Mod 10



Lower Wythoff Sequence Mod 8



Upper Wythoff Sequence Mod 8

Periodic Residue Sequences

Although some non-periodic residue sequences have interesting patterns and hence offer design possibilities, most of the interesting questions involve periodic residue sequences:

- What kinds of sequences yield periodic residue sequences?
- What are their periods for different moduli?
- What residues are present?
- How are the residues distributed?

Although there is no comprehensive answer to the first question, there are some classes of sequences that have periodic residue sequences for all moduli:

- Linear recurrences with constant coefficients, in which each term is expressed as a linear combination of previous terms:

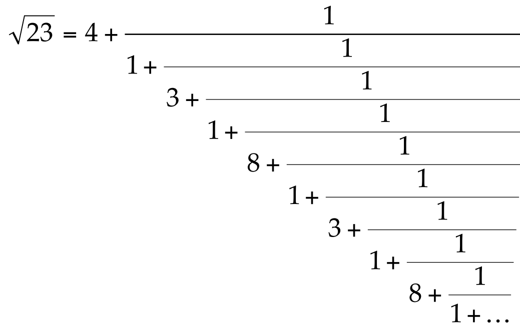
  $$a_n = c_1 \times a_{n-1} + c_2 \times a_{n-2} + \ldots + c_k \times a_{n-k}$$

  The Fibonacci sequence is an example:

  $$a_1 = a_2 = 1$$

  $$a_n = a_{n-1} + a_{n-2} \qquad n > 2$$

- The denominators of continued fraction expansions of quadratic irrationals, such as



$$\sqrt{23} = 4 + \cfrac{1}{1 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{8 + \cfrac{1}{1 + \cfrac{1}{3 + \cfrac{1}{1 + \cfrac{1}{8 + \cfrac{1}{1 + \ldots}}}}}}}}}$$

whose denominator sequence has the repeat
1, 3, 1, 8.

- The digits of the fractional parts of decimal (and other base) expansions of rational numbers, such as

  fract(2/7) = .285714285714285714 …

  which has the repeat 2, 8, 5, 7, 1, 4.

There are, of course, many others, but the ones above have been extensively studied and offer endless possibilities.

Even if a residue sequence is known to be periodic, predicting (as opposed to discovering) its period may be difficult and there are many open questions in this area.

The period often depends on the factors of the modulus. For example, the maximum period for Fibonacci residue sequences is $6 \times m$, where m is the modulus. This maximum is achieved only for moduli of the form

$$m = 2 \times 5^n \qquad n > 0$$

That is, m = 10, 50, 250, …

A quick glance at a few residue sequences shows that in many cases not all residues are present and that some residues occur more often than others. This has design implications, but this matter has not been studied in any detail.

Computing Residue Sequences

Computing residue sequences by hand is tedious and prone to error. One problem with computing residues of a sequence is that the values in many sequences get large very quickly. For example, the Fibonacci sequence, which starts out innocuously as 1, 1, 2, 3, 5, 8, … quickly gets out of hand — the forty-seventh term is 2,971,215,073.

Most hand-held calculators work with floating point numbers, not integers. Some more sophisticated calculators can perform integer operations, including computing residues, but such calculators have so many other “more important” features that it may be hard, short of buying one, to find out if a particular calculator computes residues. And, while large integers can be done by hand with enough care and patience, calculators have limits on the sizes of integers they can handle.

If you are a programmer, it’s easy to write a program to compute residues — most programming languages include a remainder operation in their repertoire of integer arithmetic. However, the size of integers is again a problem, and worse, a program may give the wrong answer for an integer larger than the

programming language handles properly. For example, the forty-seventh term in the Fibonacci sequence given above exceeds the word size for a 32-bit computer and some programming languages, including C, don't check for arithmetic overflow. Some programming languages, on the other hand, can handle arbitrarily large integers; if you're conversant with one, computing residue sequences is a snap.

All this aside, if a sequence can be formulated as a recurrence (and many can, even if how to do it is not obvious), large integers can be avoided altogether by not first computing the sequence and then getting the residues, but rather by computing the residue sequence directly.

This is possible because

$$(a + b) \bmod m = ((a \bmod m) + (b \bmod m)) \bmod m$$
$$(a - b) \bmod m = ((a \bmod m) - (b \bmod m)) \bmod m$$
$$(a \times b) \bmod m = ((a \bmod m) \times (b \bmod m)) \bmod m$$

This kind of relationship does not hold, in general, for cancellation of common factors (division). That's an interesting subject but not important here.

What all this means is that residues often can be computed on the fly. For example, Fibonacci residue sequences can be computed by

$$a_n = (a_{n-1} + a_{n-2}) \bmod m$$

The residue is taken at each step. Consequently intermediate values never exceed $2 \times (m - 1)$.

Residue Sequences in Weave Design

Residues can be used in several ways in weave design. The emphasis here is in their use in designing threading and treadling sequences.

Since residue sequences work on a 0-based system, while shafts and treadles are numbered from 1, it is necessary to convert residue sequences to a 1-based system. All that's needed is to change all 0 values to *m*, where *m* is the modulus. To convert 0-based *remainders* to 1-based *residues*, add *m* to values less than 1 and leave the rest unchanged. Since residues are not negative, the two rules amount to the same thing. See Reference 2 for an explanation.

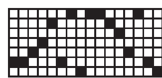
If a residue sequence is periodic, the repeat can be used as a threading or treadling unit. If a residue sequence has a pre-periodic part, that part can be discarded — or used in a variety of ways. If a residue sequence is not periodic, a portion of it can be used.

Where not all residues are present, it may be useful to "fill in" the gaps by moving others into vacant spaces. While this is not necessary if there are enough shafts or treadles without it, it makes clear the resources required and may make

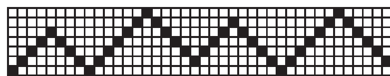
patterns easier to see.

The attached appendix [Note referenced "appendix".] shows the periods of the Fibonacci residue sequences for moduli from 4 through 16, both in their natural form and with duplicates removed and higher values moved down to fill in for missing residues.

Another approach to design using residue sequences is to use the values as "pivots" connected by alternating ascending and descending straight draws. Thus, the Fibonacci sequence mod 7, which has period 16,



shifted to a 1-based system with duplicates removed, produces the pivot sequence



which has period 40.

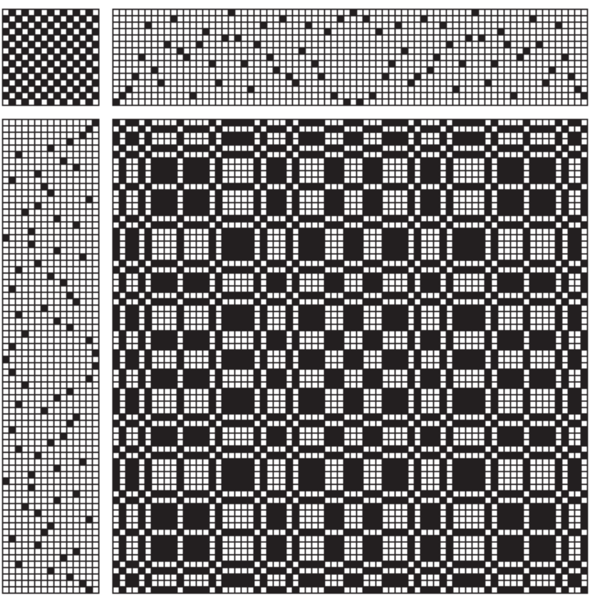
Note that when pivot values increase or decrease in succession, the intermediate values are passed over — only alternating high and low values count.

Examples

A Fibonacci Threading

The first example used a repeat of the Fibonacci residue sequence for modulus 15, reflected to create a palindrome, as a threading unit. The weave is treadled as drawn in using a tabby tie-up.

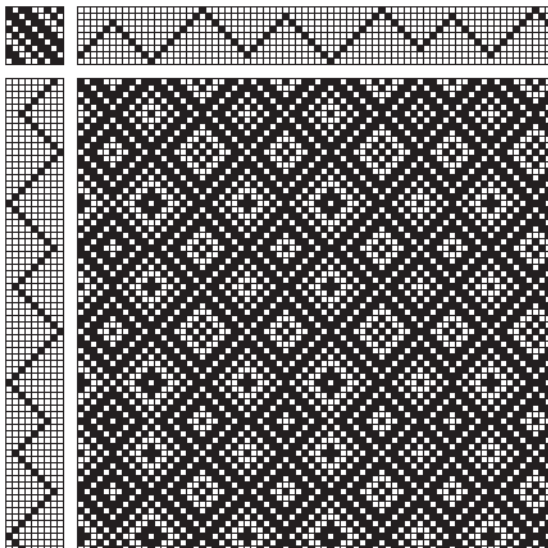
The repeat is 74. Here's the draft:



A Wythoff Point Twill

The second example uses 37 terms of the upper Wythoff sequence mod 9 to provide pivots for the threading and then reflected it to form a palindrome. Again, the treadling is as drawn in, with a /2/1/1/2/2/1 twill tie-up.

The repeat is 264. Here's part of the draft:



A reduced drawdown repeat for the entire draft is shown on the last page of this section.

Exercises and Further Explorations

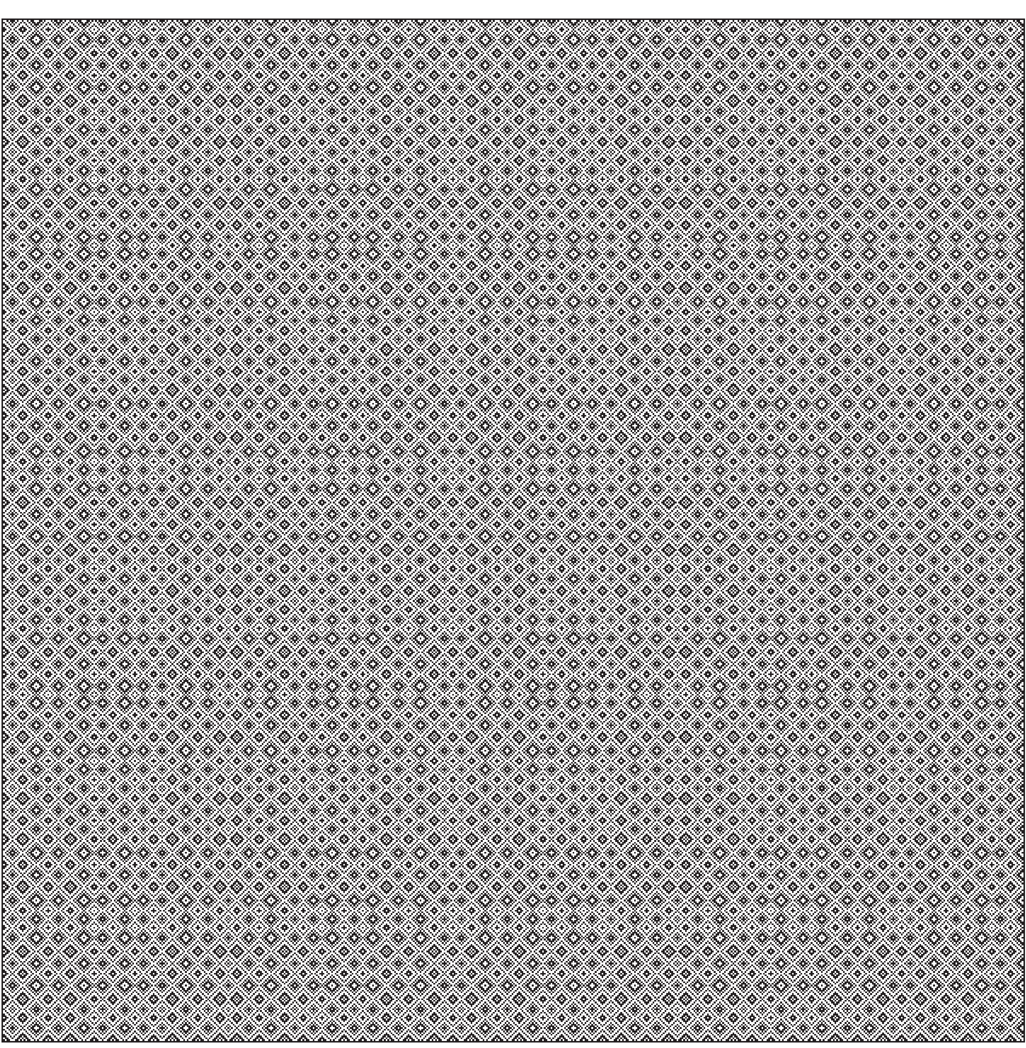
There are so many areas to explore that it's hard to even know how to list them.

There is, of course, no end to interesting residue sequences. In addition, there are many ways to combine and transform residue sequences to produce new ones.

More interesting, perhaps, are the various ways residue sequences can be used in weave design. So far, only their direct use for threading and treadling sequences and a derivative use in pivot draws have been presented. Other possibilities are:

- other derivative uses for designing threading and treadling sequences
- various uses related to profile drafting [3, 4]
- binary (mod 2) residue sequences for designing tie-ups
- binary residue sequences for designing drawdowns
- color selection

What else?



Reduced Drawdown Repeat for the Wythoff Point Twill

Recurrence Relations

A recurrence relation gives the terms of a sequence as a function of previous terms. For example, the Fibonacci sequence is given by the recurrence

$$a_n = a_{n-1} + a_{n-2}$$

with the initial terms $a_1 = a_2 = 1$ to get the sequence started. Different initial terms produce different but related sequences.

The number of initial terms required is determined by how far back in the sequence terms are specified — called the *order* of the recurrence relation. For example,

$$a_n = a_{n-1} + 2a_{n-3}$$

is a recurrence relation of order 3 and requires three initial terms, a_1, a_2, and a_3, to specify the sequence it produces.

The examples given above are linear recurrence relations with constant coefficients — LRRCs for short — and are instances of the general form

$$a_n = c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_k a_{n-k} \qquad (1)$$

where only the first powers of previous terms are used and the coefficients are constant.

There are other kinds of recurrence relations. For example,

$$a_n = a^2_{n-1} + a^2_{n-2} + a_{n-4}$$

is a quadratic recurrence of order 4, while

$$a_n = a_{n-1} + na_{n-2}$$

is a linear recurrence of order 2 but with a non-constant coefficient.

LRRCs are important in subjects including pseudo-random number generation, circuit design, and cryptography, and they have been studied extensively. LRRCs also have periodic residue sequences [1]. Despite the importance of LRRCs and the work done on them, much about them remains unknown. Very little of a general nature is known about nonlinear recurrence relations..

LRRCs

LRRC Canonical Form

Equation 1 above shows the canonical form for LRRCs. This form does not provide for a constant term, as in

$$a_n = a_{n-1} + 1$$

The reason for not having a constant term in the canonical form has to do with manipulations of LRRCs in which a constant term would require special handling.

A linear recurrence of order k with a constant term can be converted to a linear recurrence of order $k + 1$ in canonical form. Consider the example above:

$$a_n = a_{n-1} + 1 \qquad (2)$$

From this it follows that

$$a_{n-1} = a_{n-2} + 1 \qquad (3)$$

Subtracting Equation 3 from Equation 2,

$$a_n - a_{n-1} = a_{n-1} + 1 - a_{n-2} - 1$$

and hence

$$a_n = 2a_{n-1} - a_{n-2}$$

which is in the required canonical form.

Problems Related to LRRCs

There are many interesting problems related to LRRCs. The article on residue sequences touched on the properties of their residue sequences. Other problems of interest are:

- computing the sequence for an LRRC
- determining if a sequence can be represented by an LRRC and, if so, finding it
- solving an LRRC to produce an explicit formula for its nth term

An LRRC Generator

An LRRC can be completely characterized by two lists: one containing its coefficients and another containing its initial terms. For an LRRC of order k, both lists are of length k. For example, the recurrence relation

$$a_n = a_{n-1} + 2a_{n-3}$$

has the coefficient list [1, 0, 2]; the initials list, as always, determines the actual sequence. For example, the initials list [1,1,0] produces the sequence

1, 1, 0, 2, 4, 4, 8, 16, 24, 40, 72, 120, …

Finding LRRCs

Many sequences can be represented by LRRCs, even if the recurrences are not obvious.

The *difference method* often works and it can be done by hand or with a simple program [2]. This method starts with a row containing the terms of the original sequence. The second row consists of the differences of successive terms in the first row, and so on. The rows are labeled Δ^0, Δ^1, Δ^2, … . Here's an example:

Δ^0	1		7		18		34		55		81		112		148		189		…
Δ^1		6		11		16		21		26		31		36		41		…	
Δ^2			5		5		5		5		5		5		5		…		
Δ^3				0		0		0		0		0		0		…			

If a constant row appears, as it does in this example, the process is complete, there is an LRRC, and it can be obtained by using Equation 4 below, which is a consequence of the way the differences are computed:

$$\Delta^k a_n = \sum_{i=0}^{k} (-1)^i \binom{k}{i} a_{n+k-i} \qquad (4)$$

where i is the binomial coefficient

$$\binom{k}{i} = \frac{k!}{(k-i)!\,i!}$$

To get an LRRC in canonical form, it is necessary to go to a row of zeroes; Δ^3 in this case. Therefore, by Equation 4

$$\Delta^3 a_n = \sum_{i=0}^{3} (-1)^i \binom{3}{i} a_{n+3-i} = 0$$

Expanding this,

$$\binom{3}{0} a_{n+3} - \binom{3}{1} a_{n+2} + \binom{3}{2} a_{n+1} - \binom{3}{3} a_n = 0$$

and hence

$$a_{n+3} - 3a_{n+2} + 3a_{n+1} - a_n = 0$$

from which the LRRC follows

$$a_n = 3a_{n-1} - 3a_{n-2} + a_{n-3}$$

The initial terms are, of course, the first three in Δ^0.

27 Mar 2006 H3: RecurrenceRelations

Needs explanation.

Any recurrence derived from a finite number of terms is, of course, conjectural.

Explicit Formulas for LRRC Terms

Any sequence that leads to a 0 Δ sequence can be represented by a polynomial in n. Conversely, all polynomials in n can be represented by a single LRRC; the coefficients of the polynomial only affect the initial terms for the LRRC.

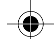
This follows from another equation that results from the method of differences:

$$a_{n+m} = \sum_{k=0}^{n} \binom{n}{k} \Delta^k a_m \qquad (5)$$

From this, an explicit formula for the nth term of the corresponding LRRC can be obtained Setting m to 1 in Equation 5 gives

$$a_{n+1} = 1\binom{n}{0} + 6\binom{n}{1} + 5\binom{n}{2} + 0\binom{n}{3}$$

(1, 6, 5, and 0 are the leading terms in Δ^0, Δ^1, Δ^2, and Δ^3.) This evaluates to



$$a_{n+1} = 1 + \frac{7}{2}n + \frac{5}{2}n^2$$

Fractal Sequences

This article reminds me that there is a "gallery" proposed for the book. It would be good to keep a list of candidates. Any volunteers?

Infinity is where things happen that don't.

— S. Knight

Ever since Benoit Mandelbrot published his book on fractals [1], we've become accustomed to seeing fantastic and beautiful fractal images such as the ones at the bottom of the pages of this article.

A fractal is a mathematical object that exhibits *self similarity* — it looks the same at any scale. If you zoom in on an image of a fractal, you see the same structure no matter how far you go, at least to the resolution of the image. In an actual fractal, there is no limit.

Fractal Sequences

In the case of sequences, a fractal sequence contains an infinite number of copies of itself, embedded within itself, as strange as this may seem.

The idea can be shown by the sequence

1, 2, 1, 3, 2, 1, 4, 3, 2, 1, 5, 4, 3, 2, 1, 6, 5, 4, …

Striking out the first instance of every value,

~~1~~, ~~2~~, 1, ~~3~~, 2, 1, ~~4~~, 3, 2, 1, ~~5~~, 4, 3, 2, 1, ~~6~~, 5, 4, …

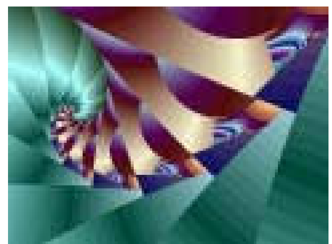
↓

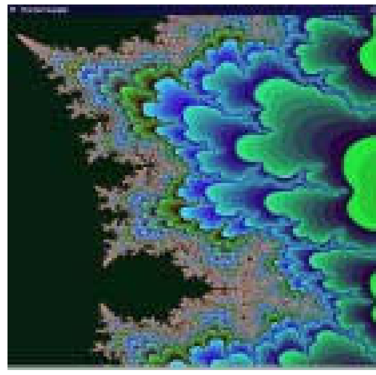
1, 2, 1, 3, 2, 1, 4, 3, 2, 1, 5, 4, …

↓

1, 2, 1, 3, 2, 1, 4, 3, 2, 1, 5, 4, …

which is the same as the original sequence, as far as it goes. Of course, we can't show the complete sequence — it is infinite — for this you have to have faith.





Rules that remove the values in fractal sequences to show their self similarity are called *fractal decimation* rules.

Image placement is a problem for this article. I've not tried to address it yet.

Bounded Fractal Sequences

The Morse-Thue sequence [2] is a binary fractal sequence, consisting only of 0s and 1s:

0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, …

For this sequence, striking out even-numbered values leaves the original sequence:

0, ~~1~~, 1, ~~0~~, 1, ~~0~~, 0, ~~1~~, 1, ~~0~~, 0, ~~1~~, 0, ~~1~~, 1, ~~0~~, …

↓

0, 1, 1, 0, 1, 0, 0, 1, …

↓

0, 1, 1, 0, 1, 0, 0, 1, …

The Morse-Thue sequence can be generalized to include more different values. The three-valued Morse-Thue sequence is

0, 1, 2, 1, 2, 0, 2, 0, 1, 1, 2, 0, 2, 0, 1, 0, 1, …

and the four-valued Morse-Thue sequence is

0, 1, 2, 3, 1 ,2, 3, 0, 2, 3, 0, 1, 3, 0, 1, 2, 1, …

Generalized Morse-Thue sequences also are fractal sequences. Can you find decimation rules that show their self similarity?

Another binary fractal sequence is the rabbit sequence [3]:

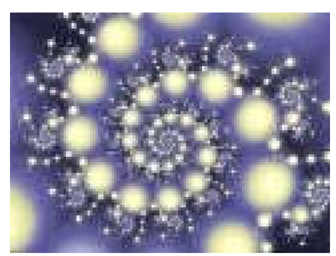
1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, …

To show that this sequence is self-similar, underline 1, 0 pairs, replace them by 1s, and replace non-underlined 1s by 0s:

$\underline{1, 0}$, 1, $\underline{1, 0}$, $\underline{1, 0}$, 1, $\underline{1, 0}$, 1, $\underline{1, 0}$, $\underline{1, 0}$, 1, …

↓

1, 0, 1, 1, 0, 1, 0, 1, 1, 0, …



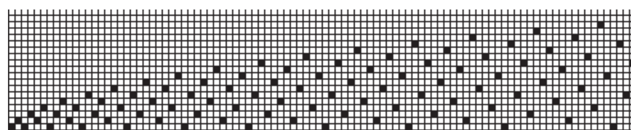
↓

1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, …

Again, this is the original sequence.

Unbounded Fractal Sequences

Many fractal sequences increase without bounds. Examples are signature sequences [2], which provide characterizations of irrational numbers (numbers like $\sqrt{7}$ that cannot be represented by fractions). Here is a grid plot of the signature sequence for ϕ, the golden ratio:



Signature sequences are one category of Kimberling fractal sequences [3].

Two operations that, when applied to Kimberling fractal sequences, produce fractal sequences are *upper trimming* and *lower trimming*. Upper trimming strikes out the first instance of every value, as illustrated by the first example in this article. Lower trimming subtracts 1 from each value and discards 0s. For the first example in this article, it goes like this:

1, 2, 1, 3, 2, 1, 4, 3, 2, 1, 5, 4, 3, 2, 1, 6, 5, 4, …

↓

0, 1, 0, 2, 1, 0, 3, 2, 1, 0, 4, 3, 2, 1, 0, 5, 4, 3, …

↓

1, 2, 1, 3, 2, 1, 4, 3, 2, 1, 5, 4, 3, …

↓

1, 2, 1, 3, 2, 1, 4, 3, 2, 1, 5, 4, 3, …

This is the same as the original sequence. This is not always true of lower trimming, but the result always is some fractal sequence.

Periodic Sequences

A periodic sequence is a sequence in which a subsequence repeats [4]. An example is the sequence of Fibonacci numbers, mod 6. The repeat length is 24:

1, 1, 2, 3, 5, 2, 1, 3, 4, 1, 5, 0, 5, 5, 4, 3,
1, 4, 5, 3, 2, 5, 1, 0

Infinite periodic sequences are, technically speaking, self similar. Decimation rules for periodic sequences are simple: Remove the initial repeat. In fact removing any repeat or any combination of repeats does the same thing: There is an infinite number of decimation rules for an infinite periodic sequence.

Periodic sequences have an important role in weave design, but they lack the intriguing aspects of other kinds of fractal sequences. Periodic sequences are best left to other contexts.

Adapting Fractal Sequences to Weave Design

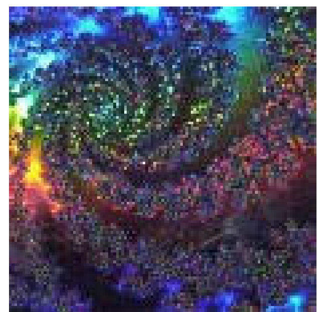
Threading and Treadling Sequences

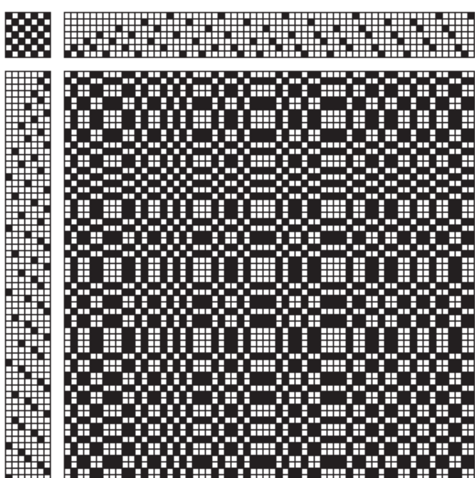
The obvious use of fractal sequences in weave design is as threading and treadling sequences. For this purpose, fractal sequences can be divided into two classes: Those whose values fit within the constraints of a loom and those that do not.

The former can be used directly — at least portions of them. For fractal sequences, such as the Morse-Thue sequences, that have 0s, simply adding 1 to each value produces a sequence that works for the 1-based numbering of shafts and treadles.

Fractal sequences that have values exceeding the constraints of a loom can be converted to the desired range using modular reduction [5].

For example, the signature sequence for the golden ratio, reduced (shaft) modulo 7, treadles as drawn in with a tabby tie-up produces this draft:

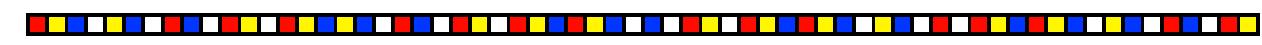




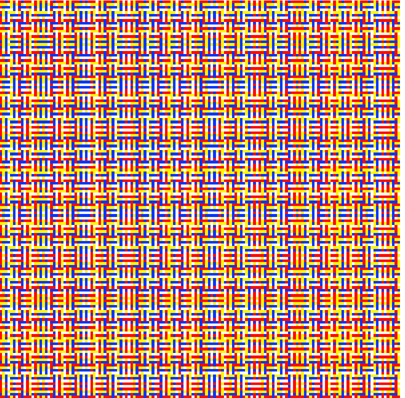
Color Sequences

Fractal sequences also can be used to derive warp and weft color sequences by assigning a color to each different value in the sequence.

Here is a color sequence based on the 4-valued Morse-Thue sequence:



A plain weave with this color sequence used for the warp and weft is



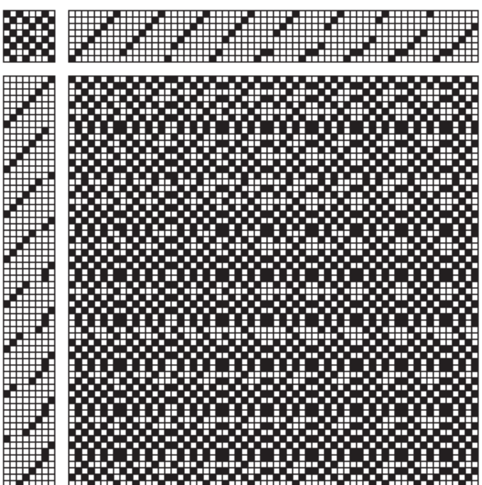
Selecting Fractal Sequences for Weave Design

You can't expect to get weave designs from fractal sequences that rival the fractal images at the bottoms of the pages at this article: These images are too complex to be woven in a loom-controlled fashion. Tapestry weaving is an intriguing possibility, however.

The fractal sequences that are the most interesting from a design viewpoint

are those that develop successive variations and extensions of an initial "theme". The Morse-Thue and rabbit sequences are excellent examples of this type of fractal sequence.

Such sequences often give designs that appear at first glance to be periodic but on closer examination show continual variations. An example is this draft based on the 8-valued Morse-Thue sequence:



Prospecting for Fractal Sequences

The few examples of fractal sequences given here can get you started on designing. But if you find the results interesting, you may want to try other fractal sequences.

The best source for integer sequences of all kinds is the *On-Line Encyclopedia of Integer Sequences* [2]. There you can search for sequences by entering a few initial terms or using keywords. The keywords fractal and self-similar produce long lists of relevant sequences. You can also look for Morse-Thue and the synonym Thue-Morse.

Sequences in the *Encyclopedia* also can be looked up by their identifying numbers, which consist of an A followed by 6 digits. Some numbers for examples given here and related ones are:

A010060	Morse-Thue Sequence
A005614	Rabbit Sequence
A053838	2-Valued Morse-Thue Sequence
A053839	4-Valued Morse-Thue Sequence

When you look up a sequence by number, the description may not be what

you expect: A sequence may have many origins. Similarly, there are different forms of some sequences.

The site also has a page listing many self-similar sequences with simple decimation rules [6].

The Morse-Thue Sequence

[Check cross references.] The Morse-Thue sequence is a binary fractal sequence with many interesting properties. It begins as

0, 1, 1, 0, 1, 0, 0, 1, …

This sequence was introduced in 1906 by the Norwegian mathematician Axel Thue (pronounced TOO) as an example of an aperiodic recursively computable string of symbols.

Axel Thue Sidebar

Later Marvin Morse (on the principle of quoting when you can't do better)

> … proved that the trajectories of dynamic systems whose phase spaces have a negative curvature everywhere can be completely characterized by a *discrete* sequence of 0s and 1s — a stunning discovery [1].

Because of the importance of Morse's discovery, his name usually is listed first, although the sequence sometimes also is called the Thue-Morse sequence.

Constructing the Morse-Thue Sequence

There are many ways of constructing this sequence. The one shown most is the siple L-System [check cross references]

seed: 0

0 ➛ 01

1 ➛ 10

At each step every 0 and 1 is replaced by the specified pair, simultaneous (at least conceptually), Thus, the development proceeds like this:

0 ➛ 0, 1 ➛ 0, 1, 1, 0 ➛ 0, 1, 1, 0, 1, 0, 0, 1 ➛ …

Another way to produce the Morse-Thue sequence is to start with 0 and iterate the following process: Take the present sequence and append its complement (replacing 0 by 1 and 1 by 0) to it. It goes like this:

0
0, 1
0, 1, 1, 0
0, 1, 1, 0, 1, 0, 0, 1
0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0
…

The advantage of this method is that it is simple and the number of terms produced increases rapidly.

A third method for producing the Morse-Thue sequence is to write the nonnegative integers in binary form:

0, 1, 10, 11, 100, 101, 110, 111, …

Then replace every value by its *digit reduction* mod 2.

> Digit reduction sums the digits of a number and repeats the process if necessary until only one digit remains. Thus the digit reduction of 111 is 3, whose residue mod 2 is 1.

Properties of the Morse-Thue Sequence

The Morse-Thue sequence is self similar (fractal) [reference], as can be seen by striking out every even-numbered value, which produces the original sequence:

0, ~~1~~, 1, ~~0~~, 1, ~~0~~, 0, ~~1~~, 1, ~~0~~, 0, ~~1~~, 0, ~~1~~, 1, ~~0~~, …

Among the fascinating properties of the Morse-Thue sequence is that it is *cube-free*. This means that it does not contain the subsequences 0, 0, 0 or 1, 1, 1. But cube-free is a more general concept. In the jargon of combinatorics on words [2], a word is any sequence of characters from the alphabet being used (here, 0 and 1). Cube-free applies to all words. For example, if

W = 1, 0, 1, 1, 0

(which is a word in the Morse-Thue sequence), then W, W, W, which is

1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0

does not occur in the Morse-Thue sequence.

Generalizing the Morse-Thue Sequence

The Morse-Thue sequence generalizes to bases other than 2. For example, the base-5 generalization of the Morse-Thue sequence is

0, 1, 2, 3, 4, 1, 2, 3, 4, 0, 2, 3, 4, 0, 1, 3, 4, 0,
1, 2, 4, 0, 1, 2, 3, 1, 2, 3, 4, 0, …

All three methods used for computing the regular, base-2 Morse-Thue sequence generalize for larger bases.

Geometrical Interpretations of the Morse-Thue Sequence

Visualizing 0 as a black square and 1 as a white square, the Morse-Thue sequence appears graphically as shown in Figure Ω.1:

…

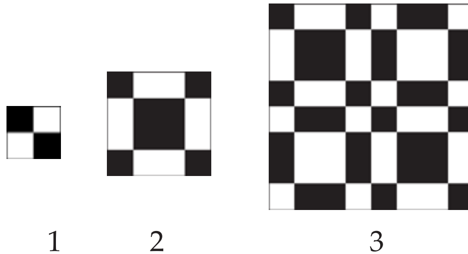
Figure Ω.1. The Morse-Thue Sequence

The steps for the append-complement method of construction are shown in Figure Ω.2.

…

Figure Ω.2. Morse-Thue Sequence Construction

This can be extended to two dimensions by at each step appending the complement both horizontally and vertically [3]. Figure Ω.3 shows the first four iterations:



24 Jun 2006 H5: MorseThue

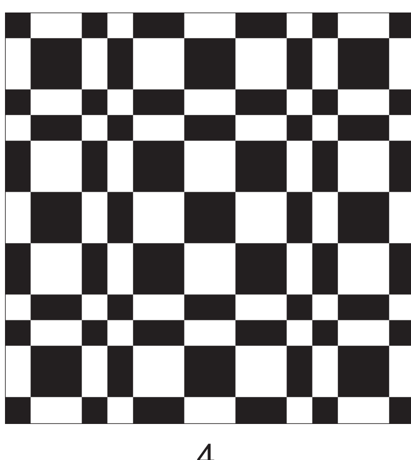


Figure Ω.3. Constructing The Morse-Thue Plane

Like the Morse-Thue sequence itself, the Morse-Thue plane is fractal. And, despite the appearance of symmetry and regularity, there are no repetitions. That is, no finite portion of the plane can be tiled regularly to produce the whole plane.

Applications of the Morse-Thue Sequence

The Morse-Thue sequence has applications in many areas. In addition to the one mentioned at the beginning of this article, the Morse-Thue sequence has been used in graphic design and in music composition [4-6].

It should not be surprising to discover that the Morse-Thue sequence can be used as the basis for a variety of interesting weaves. Figure Ω.4 shows a weaving draft that was "drawn up" from the sixth iteration of the plane-construction process shown in Figure 3. Notice that the Morse-Thue sequence appears in the threading and treadling and that it takes only two shafts and two treadles to produce this weave.

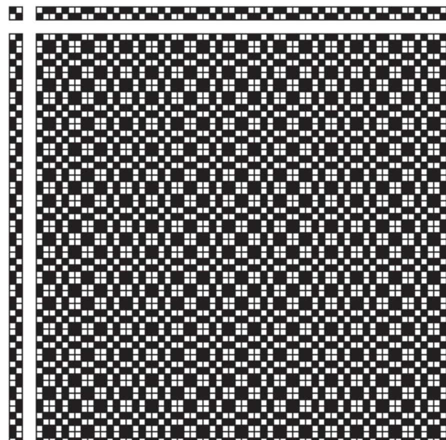


Figure Ω.4. A Morse-Thue Weave

There are other ways the Morse-Thue sequence can be used in weave design. We'll explore some of these in other sections.

Signature Sequences

An interesting class of fractal sequences consists of *signature sequences* for irrational numbers. The signature sequence of the irrational number x is obtained by putting the numbers

$$i + j \times x \qquad i, j = 1, 2, 3, \ldots$$

in increasing order. Then the values of i for these numbers form the signature sequence for x, which is denoted by $\mathbb{S}(x)$.

Here's the signature sequence for ϕ, the golden mean:

1, 2, 1, 3, 2, 4, 1, 3, 5, 2, 4, 1, 6, 3, 5, 2, 7, 4, 1, 6, 3, 8,
5, 2, 7, 4, 9, 1, 6, 3, 8, 5, 10, 2, 7, 4, 9, 1, 6, 11, …

Both upper trimming and lower trimming of a signature sequence leave the sequence unchanged.

Signature sequences have a characteristic appearance, but they vary considerably in detail depending of the value of x.

Signature sequences start with a run 1, 2, …, n+1, where $n = \lfloor x \rfloor$, the integer part of x. The larger the value of x, the more quickly terms in the sequence get larger. Most signature sequences display runs, either upward or downward or both — which one is usually a matter of visual interpretation. At some point, most signature sequences become interleaved runs. This sometimes gives the illusion of curves.

Although signature sequences are defined only for irrational numbers, the algorithm works just as well for rational numbers. Although signature sequences for rational numbers are not fractal sequences, they are as close as you could determine manually. The structure of a signature sequence depends on the magnitude of x. Furthermore, there are irrational numbers arbitrarily close to any rational number. There is no difference in the initial terms of signature sequences for numbers that are close together. For example, $\mathbb{S}(3.0)$ and $\mathbb{S}(\pi)$ do not differ until their 117th terms.

It's also worth noting that there really is no way, in general, to perform exact computations for irrational numbers. Computers approximate real numbers, and hence irrational numbers, using floating-point arithmetic. A floating-point number representing an irrational number is just a (very good) rational approximation to the irrational number. For example, the standard 64-bit floating-point encoding for π is

07 Jul 2006 H6: SignatureSequences

7074237752028440 / 2^{51}

Figure Ω.1 shows grid plots for some signature sequences. Signature sequences for large numbers are not included because they are unwieldy.

Using Signature Sequences in Weaving Drafts

Signature sequences can be used as the basis for threading and treadling sequences. To use signature sequences for this purpose, it is necessary to bring the values of terms within the bounds of the number of shafts and treadles used. The mathematically reasonable way is to take their residues, modulo the number of shafts or treadles, using 1-based arithmetic. Figure Ω.2 shows residue sequences derived from signature sequences. In most cases, taking residues preserves the essential characteristics of signature sequences.

Sequences like these, if used directly, produce drawdown patterns that lack repeats or symmetry. More attractive patterns can be obtained by taking a small portion of a signature sequence and then reflecting it to get symmetric repeats.

Figure Ω.3 shows a draft for such a sequence with 16 shafts and treadles and a $\frac{2}{\quad 2}$ twill tie-up.

It seems natural to use initial terms of a signature sequence. The structure of signature sequences, however, changes as the sequence goes on. Figure Ω.4 shows magnified portions of the drawdown pattern for a signature sequence. This suggests that it might be worth trying subsequences of signature sequences in various locations.

Figure Ω.5 shows some drawdown patterns for various combinations of signature sequences. All have 16 shafts and treadles and $\frac{2}{\quad 2}$ twill tie-ups.

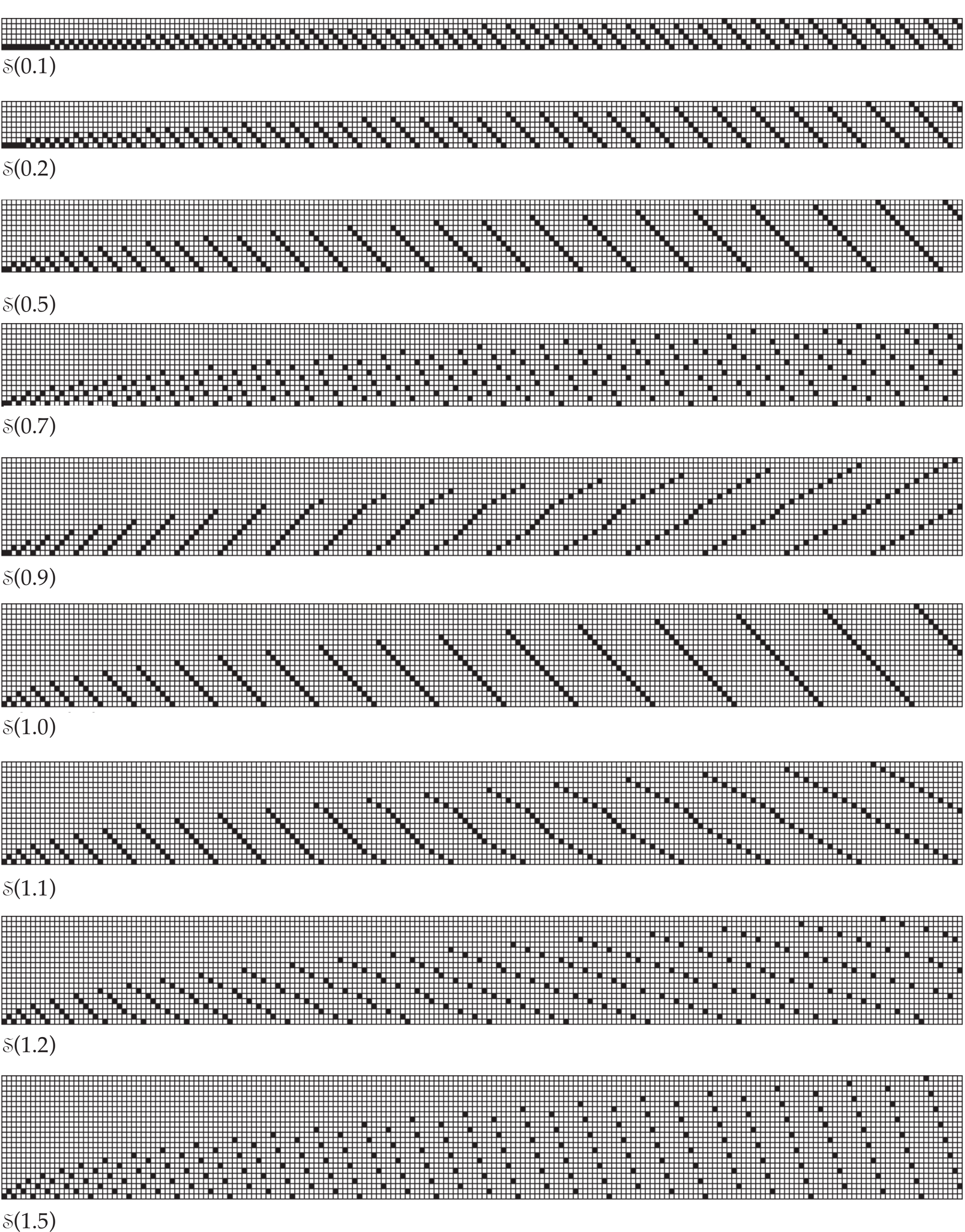


Figure 1

. Grid Plots for Signature Sequences

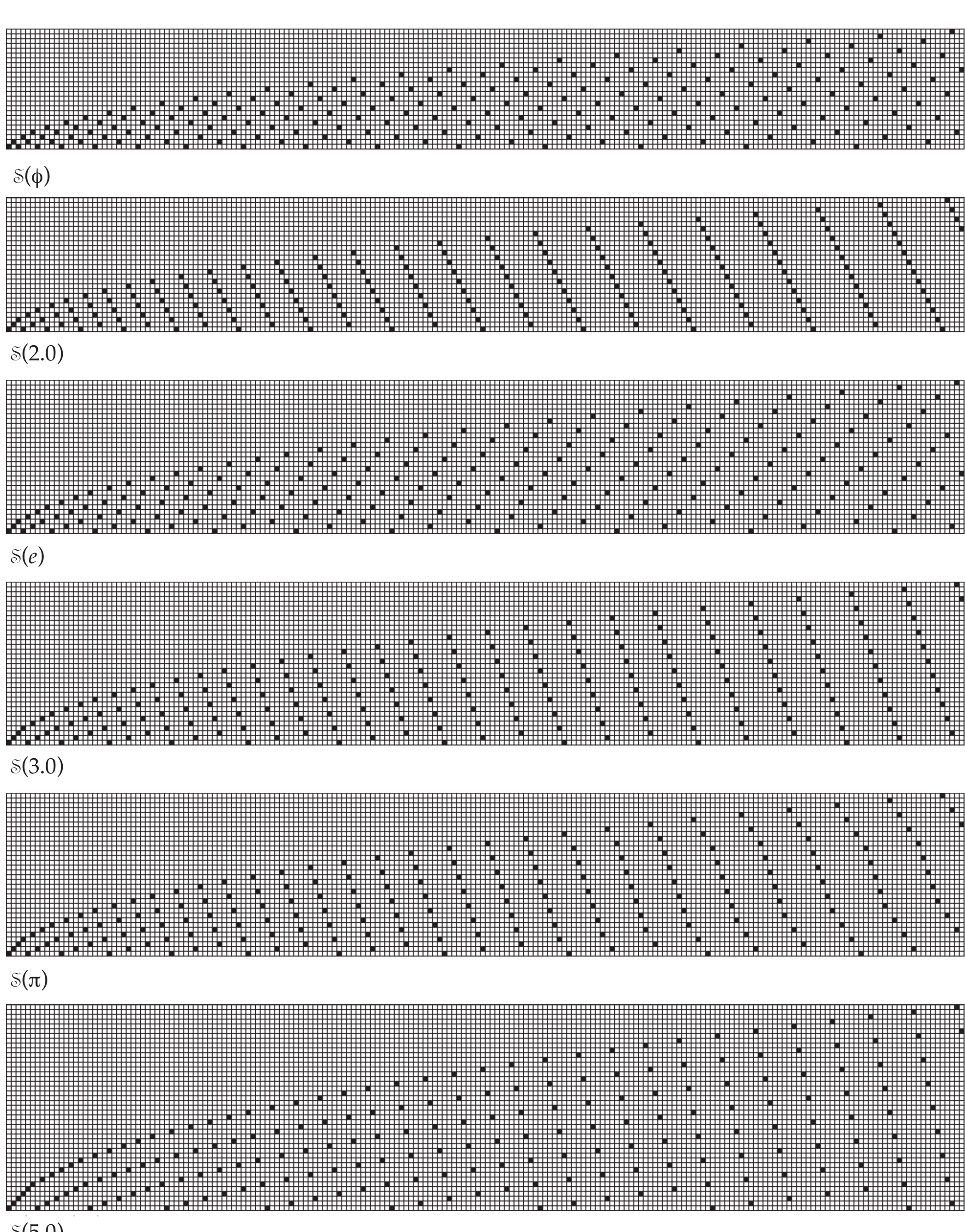


Figure Ω.1, continued. Grid Plots for Signature Sequences

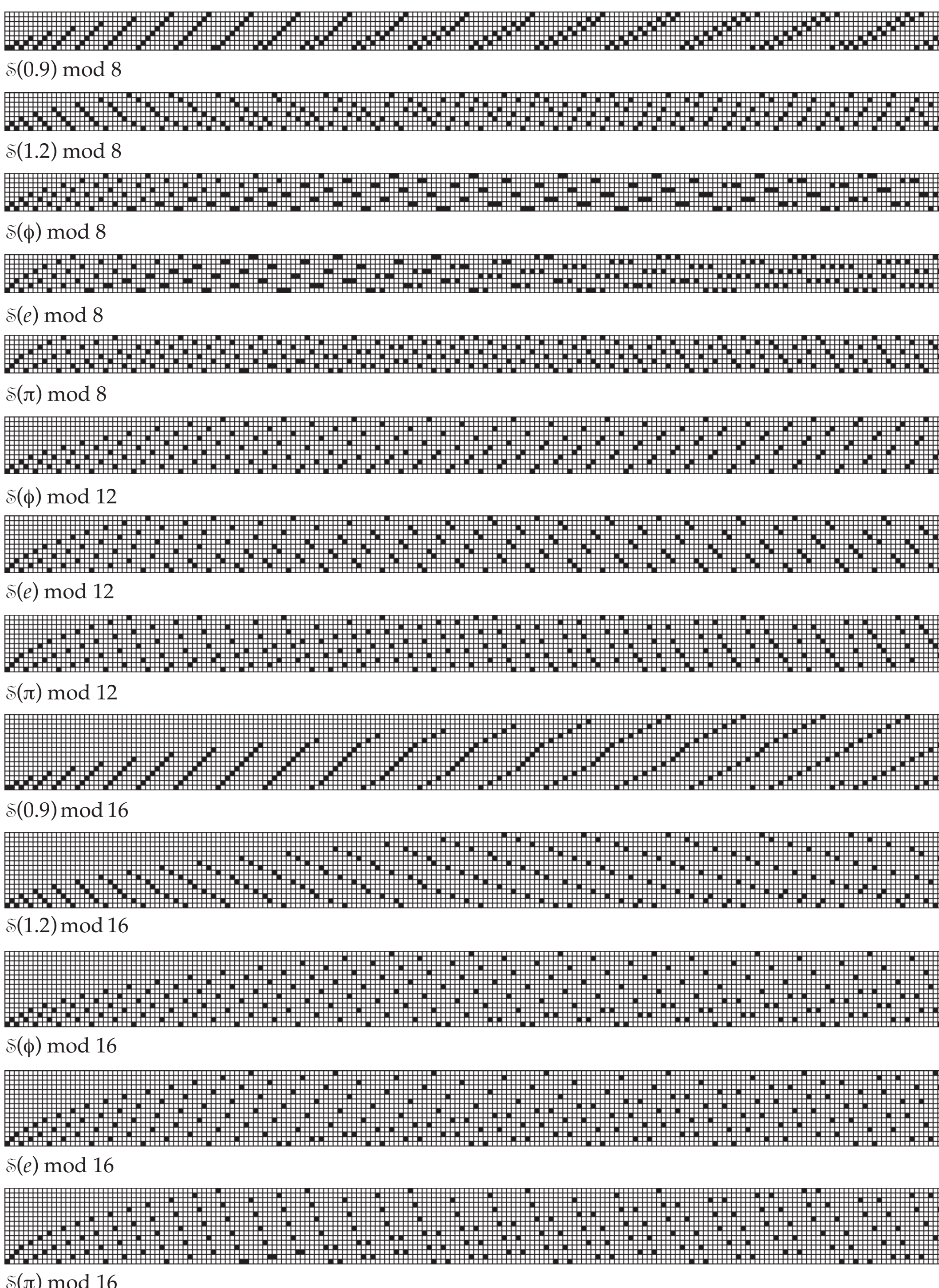


Figure Ω.2. Signature Sequence Residues

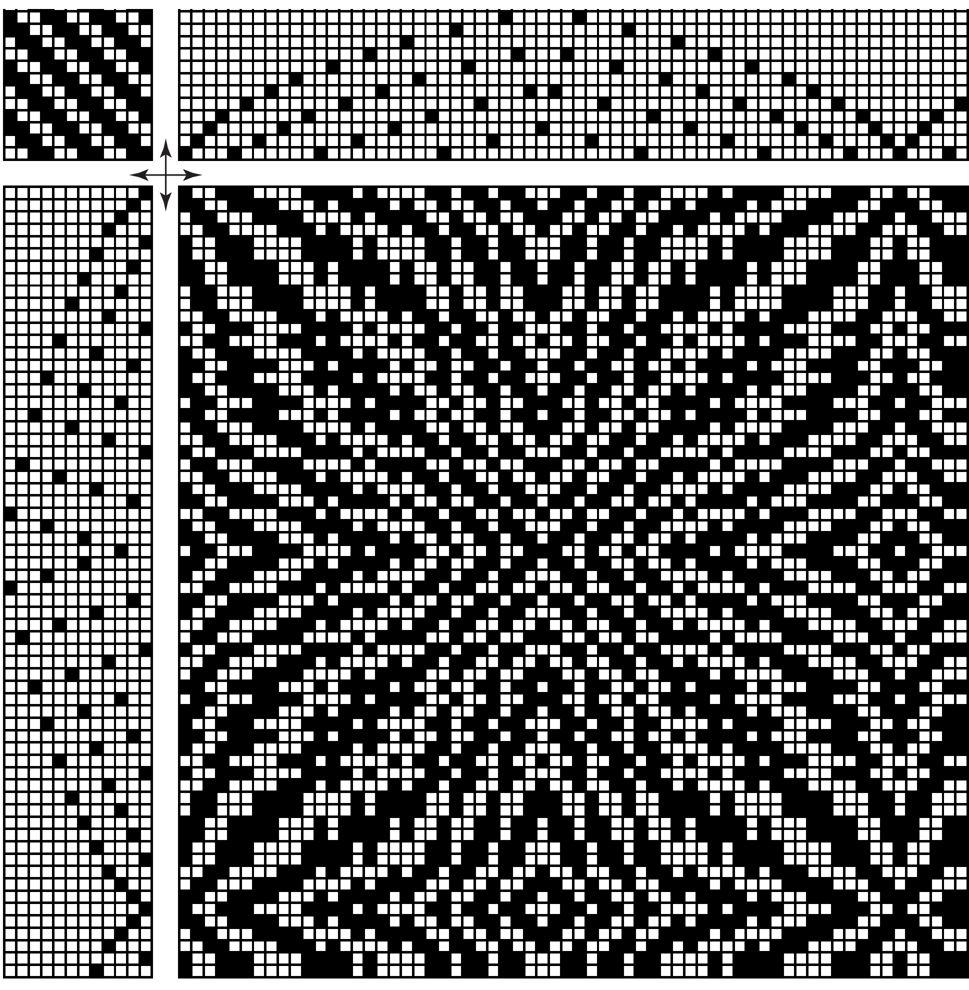


Figure Ω.3. Drawdown for A Reflected Portion of $\mathbb{S}(\pi)$

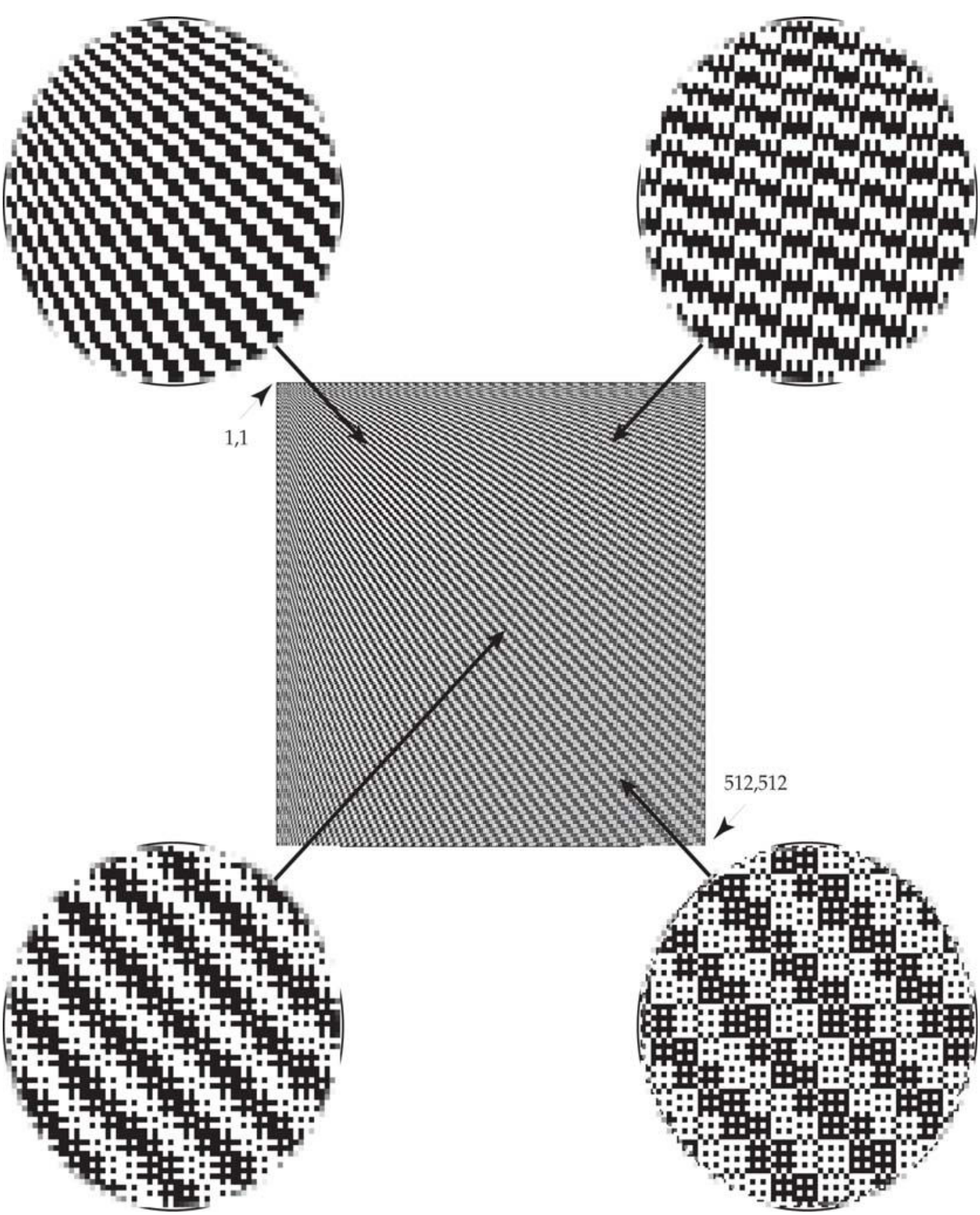
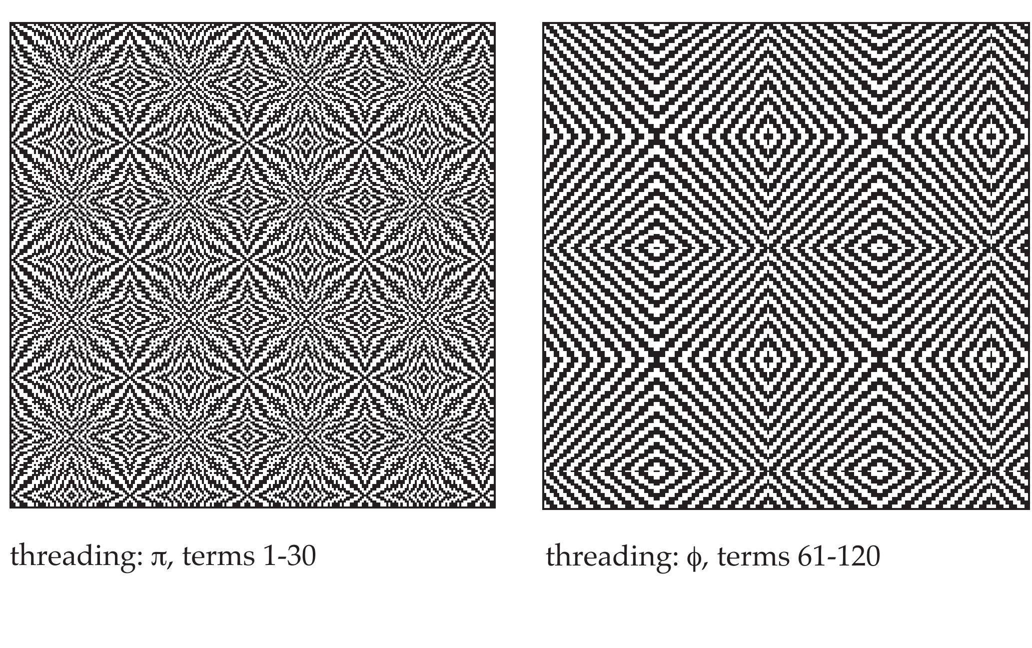
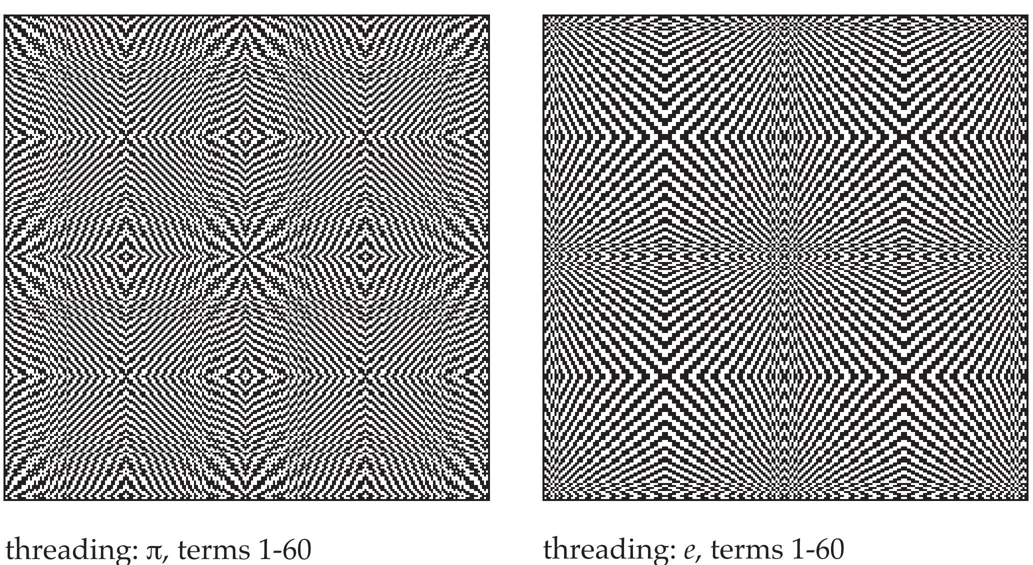


Figure Ω.4. Magnified Portions of the $\phi \times \phi$ Signature Drawdown Plane



threading: π, terms 1-30

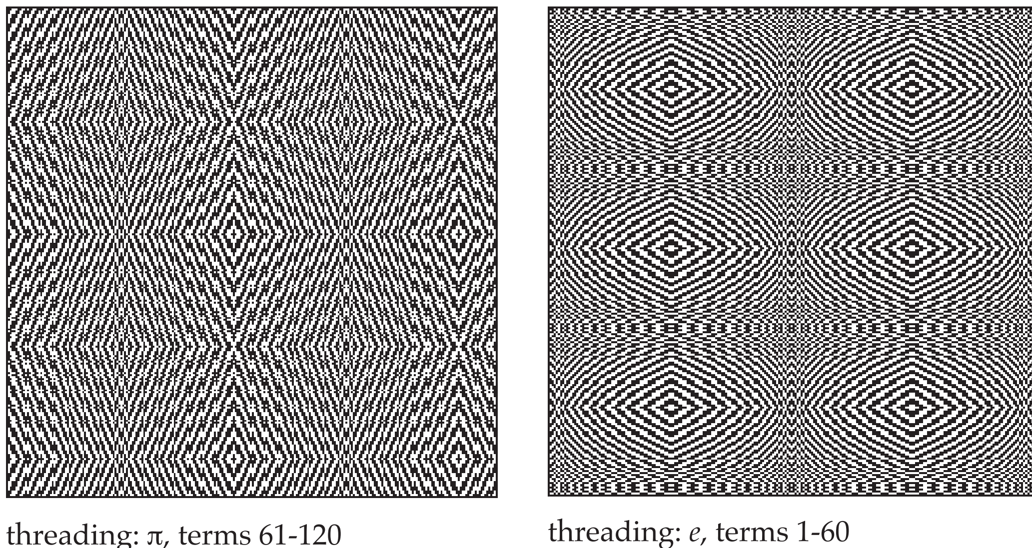
threading: ϕ, terms 61-120



threading: π, terms 1-60

threading: *e*, terms 1-60

Figure Ω.5. Drawdown Patterns for Signature Sequences



threading: π, terms 61-120

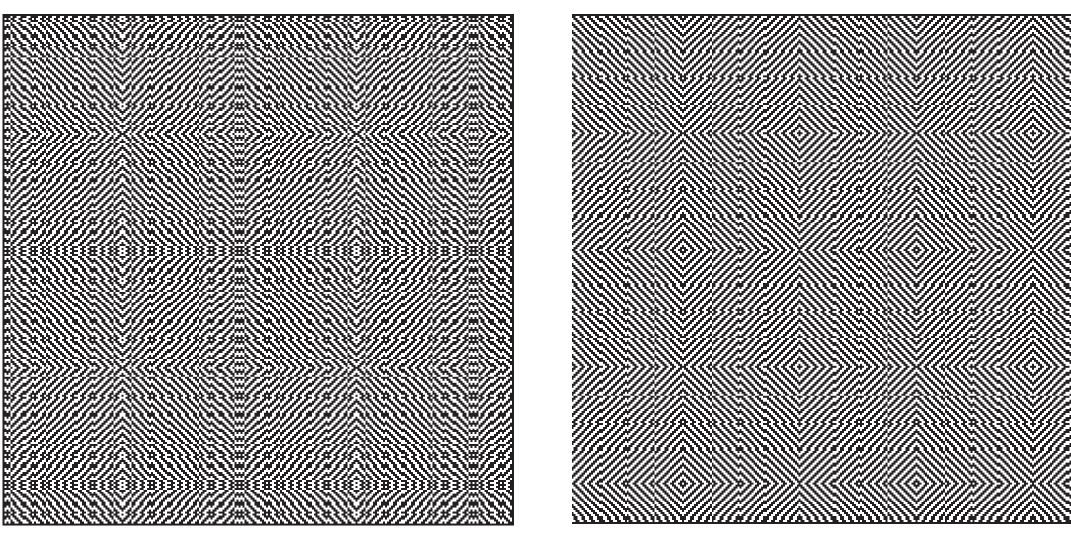
threading: e, terms 1-60

threading: e, terms 1-60

threading: 0.9, terms 61-120

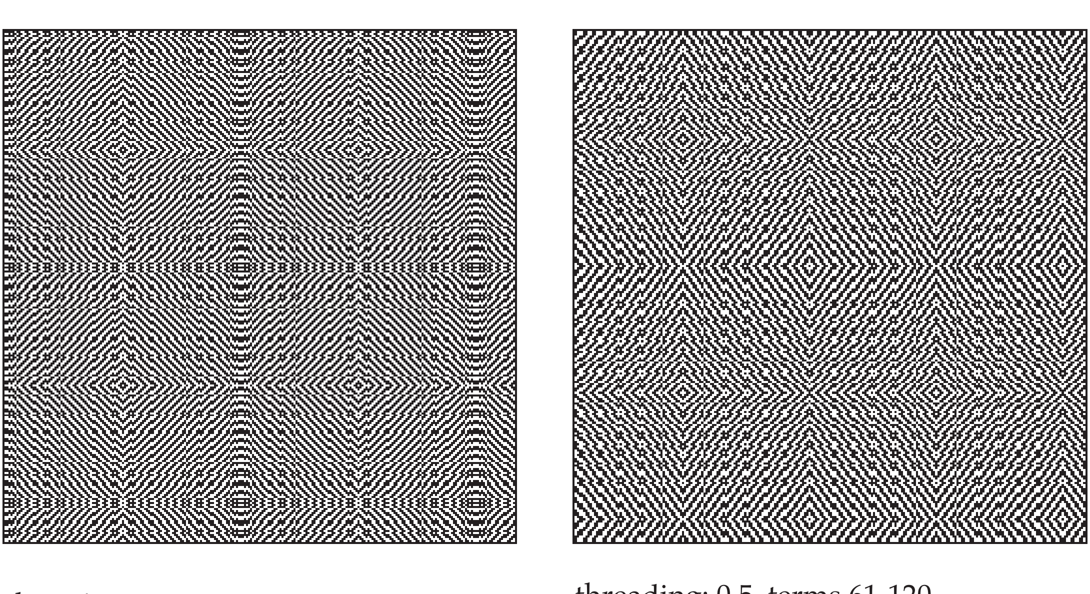
Figure Ω.5, continued. Drawdown Patterns for Signature Sequences

07 Jul 2006 H6: SignatureSequences



threading: 1.0, terms 1-60

threading: 1.0, terms 61-120



threading: 0.9, terms 1-60

threading: 0.5, terms 61-120

Figure Ω.5, continued. Drawdown Patterns for Signature Sequences

Note: floor() should be replaced by a presently missing font combination.

Spectra Sequences

Given an irrational number α, the integer sequence *floor*(α), *floor*(2α), *floor*(3α), … is called the *spectrum sequence* of α. For example, the spectrum sequence of π is 3, 6, 9, 12, 15, 18, 21, 25, 28, 31, 34 … and the spectrum sequence of e is 2, 5, 8, 10, 13, 16, 19, 21, 24, 27, 29, … .

The spectrum sequence of x is denoted by $\mathcal{S}(x)$.

Beatty Sequences

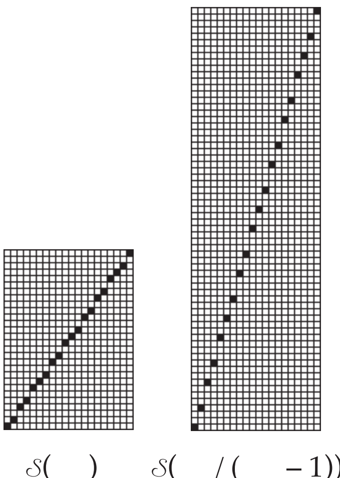
A very interesting case occurs for two positive irrational numbers α and β such that

$$1/\alpha + 1/\beta = 1$$

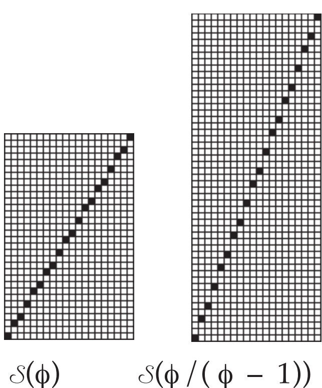
Then $\mathcal{S}(\alpha)$ and $\mathcal{S}(\beta)$ together contain all the positive integers without repetition. These are called *Beatty sequences* after Samuel Beatty, who discovered their remarkable property.

Note: This formulation is by Weisstein [1]. Superficially this gives the impression that α and β are independent. However, given α, $\beta = \alpha/(\alpha - 1)$. Similarly, given β, $\alpha = \beta/(\beta - 1)$. It would seem more straightforward to say that, given a positive irrational number α, $\mathcal{S}(\alpha)$ and $\mathcal{S}(\alpha/(\alpha - 1))$ are Beatty sequences that together contain all the positive integers without repetition. The catch is that α must be greater than 1; otherwise β is negative.

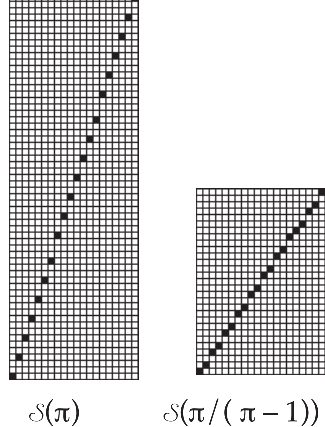
Here are grid plots for some Beatty sequence pairs.



$\mathcal{S}(\quad)$ $\quad$ $\mathcal{S}(\quad /(\quad - 1))$



$\mathcal{S}(\phi)$ $\mathcal{S}(\phi/(\phi - 1))$



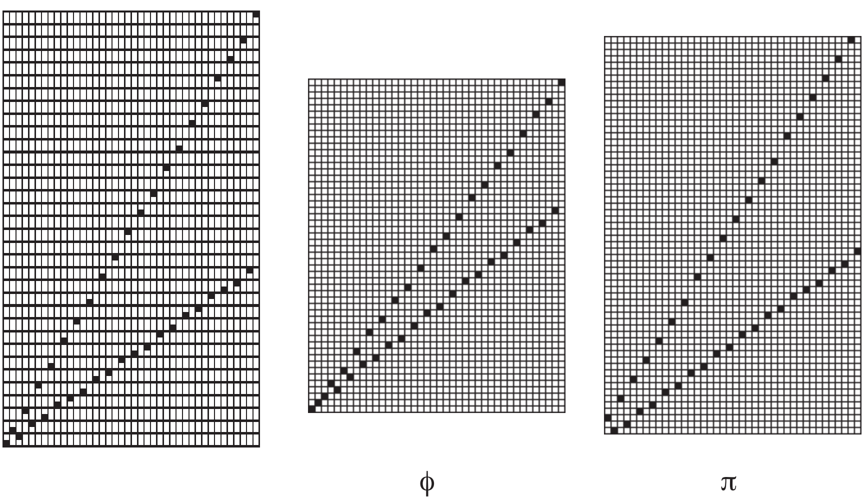
$\mathcal{S}(\pi)$ $\mathcal{S}(\pi/(\pi - 1))$

Here are two observations about spectra sequences:

(1) Because /(− 1) simplifies to 2 + , the complementary sequence $\mathcal{S}($) is just the same sequence, spread out over bigger gaps. Clusters of two and three follow the same pattern in both sequences.

(2) The complement of $\mathcal{S}(\phi)$ is $\mathcal{S}(\phi^2)$ after simplification, due to the special properties of ϕ.

Collating the pairs of Beatty sequences in shown above gives these grid plots:

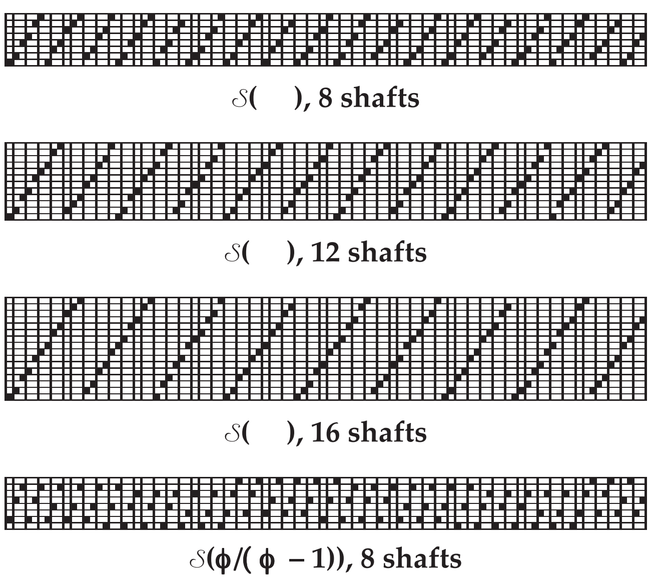


Spectra T-Sequences

As mentioned several times in earlier sections, almost all integer sequences with any structure can be used as the basis for interesting weave patterns. Spectra sequences are no exception.

As usual, it's necessary to bring such sequences within the bounds of the number of shafts or treadles used by taking residues in shaft arithmetic [2].

The following grid plots show some Beatty T-sequences and different numbers of shafts.

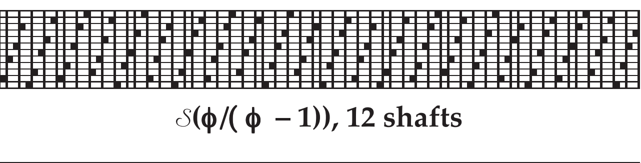


$\mathcal{S}$( ), 8 shafts

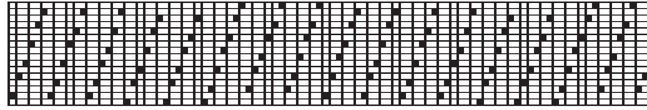
$\mathcal{S}$( ), 12 shafts

$\mathcal{S}$( ), 16 shafts

$\mathcal{S}(\phi/(\phi - 1))$, 8 shafts



$\mathcal{S}(\phi/(\phi - 1))$, 12 shafts

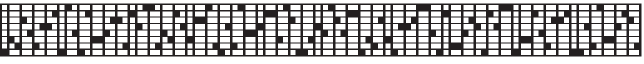


$\mathcal{S}(\phi/(\phi - 1))$, 16 shafts

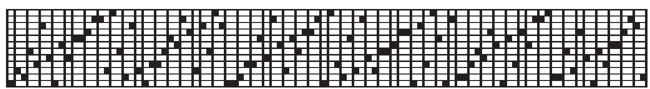
$\mathcal{S}(\pi)$, 4 shafts

$\mathcal{S}(\pi)$, 6 shafts

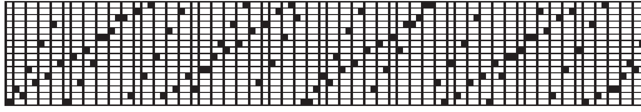
The following grid plots show collated Beatty t-sequences corresponding to the previous sequences.



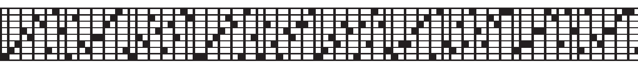
, 8 shafts



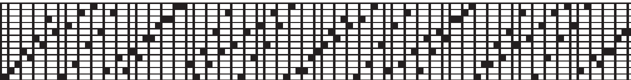
, 12 shafts



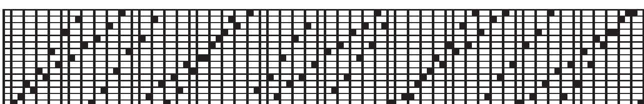
, 16 shafts



ϕ, 8 shafts



ϕ, 12 shafts

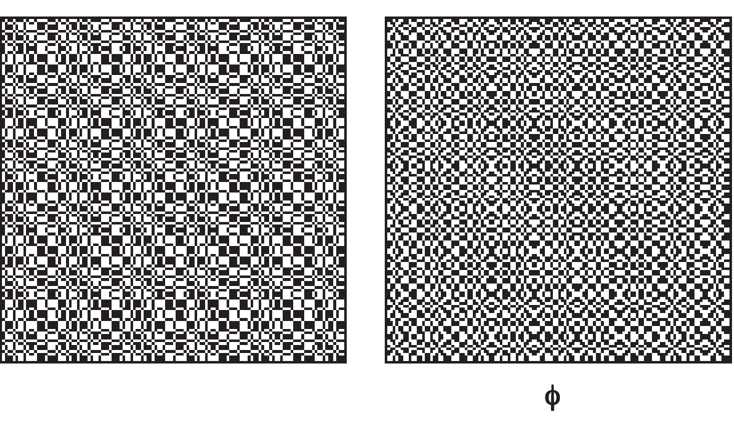


ϕ, 16 shafts

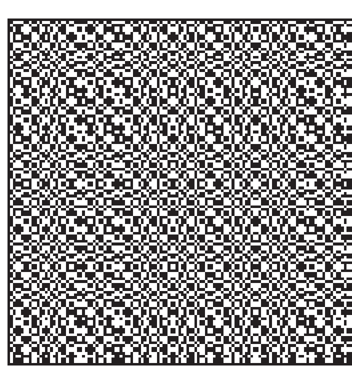
π, 4 shafts

π, 6 shafts

The following drawdowns are for collated Beatty t-sequences with 2/2 twill tie-ups and 8 shafts and treadles, treadled as drawn in.



ϕ



π

Continued Fractions

Continued fractions are part of the "lost mathematics," the mathematics now considered too advanced for high school and too elementary for college.

— Petr Beckmann, *A History of Pi*

Most persons taking courses in mathematics do not encounter continue fractions. When first encountered, they have a forbidding appearance. Yet continued fractions have an elegant theory and are important in several branches of mathematics.

A continued fraction is a fraction in which the numerators and denominators may contain (continued) fractions. Displayed in their full laddered form, they look like this:

$$\pi = 3 + \cfrac{1}{7 + \cfrac{1}{15 + \cfrac{1}{1 + \cfrac{1}{292 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{1 + \dots}}}}}}}$$

See Figure Ω.1 on the next page for other examples.

The numerators and denominators in a continued fraction can themselves be complicated, as evidenced by Figure Ω.1i. Most work on continued fractions deals with *ordinary* continued fractions, in which the numerators and denominators are numbers:

$$a_1 + \cfrac{b_1}{a_2 + \cfrac{b_2}{a_3 + \cfrac{b_3}{a_4 + \cfrac{b_4}{a_5 + \cfrac{b_5}{a_6 + \cfrac{b_6}{a_7 + \cfrac{b_7}{a_8 + \dots}}}}}}}$$

Two sequences completely characterize an ordinary continued fraction: a_1, a_2, a_3, a_4, … and b_1, b_2, b_3, b_4 … .

A *simple* continued fraction is an ordinary continued fraction in which all the numerators are 1 and all the denominators are integers and positive except possibly a_1:

a

$$e-1=1+\cfrac{1}{1+\cfrac{1}{2+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{4+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{6+\ldots}}}}}}}}$$

b

$$\frac{1}{e-2}=1+\cfrac{1}{2+\cfrac{2}{3+\cfrac{3}{4+\cfrac{4}{5+\cfrac{5}{6+\cfrac{6}{7+\cfrac{7}{8+\ldots}}}}}}}$$

c

$$\frac{\pi}{2}=1-\cfrac{1}{3-\cfrac{2\cdot 3}{1-\cfrac{1\cdot 2}{3-\cfrac{4\cdot 5}{1-\cfrac{3\cdot 4}{3-\cfrac{6\cdot 7}{1-\cfrac{5\cdot 6}{3-\ldots}}}}}}}$$

d

$$\frac{4}{\pi}=1+\cfrac{1^2}{2+\cfrac{3^2}{2+\cfrac{5^2}{2+\cfrac{7^2}{2+\cfrac{9^2}{2+\cfrac{11^2}{2+\cfrac{13^2}{2+\ldots}}}}}}}$$

e

$$\sin(x)=\cfrac{x}{1+\cfrac{x^2}{(2\cdot 3-x^2)+\cfrac{2\cdot 3x^2}{(4\cdot 5-x^2)+\cfrac{4\cdot 5x^2}{(6\cdot 7-x^2)+\ldots}}}}$$

f

$$\tan(1)=1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{3+\cfrac{1}{1+\cfrac{1}{5+\cfrac{1}{1+\cfrac{1}{7+\cfrac{1}{1+\ldots}}}}}}}}$$

g

$$\frac{1+\sqrt{5}}{2}=1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\ldots}}}}}}}$$

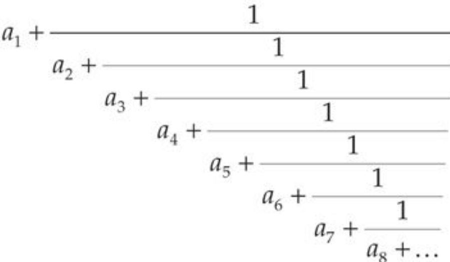
h

$$\log(1+x)=\cfrac{x}{1+\cfrac{1^2x}{2+\cfrac{1^2x}{3+\cfrac{2^2x}{4+\cfrac{2^2x}{5+\cfrac{3^2x}{6+\cfrac{3^2x}{7+\ldots}}}}}}}$$

i

$$\sqrt[3]{2}+2=\cfrac{1}{3+\cfrac{3+\cfrac{1}{3+\cfrac{3+\ldots}{3+\ldots}}}{3+\cfrac{3+\cfrac{1}{3+\cfrac{3+\ldots}{3+\ldots}}}{3+\cfrac{3+\cfrac{1}{3+\cfrac{3+\ldots}{3+\ldots}}}{3+\cfrac{3+\cfrac{1}{3+\cfrac{3+\ldots}{3+\ldots}}}{3+\cfrac{3+\cfrac{1}{3+\cfrac{3+\ldots}{3+\ldots}}}{3+\cfrac{3+\cfrac{1}{3+\cfrac{3+\ldots}{3+\ldots}}}{3+\ldots}}}}}}}$$

Figure Ω.1. A Gallery of Continued Fractions



Only one sequence is needed to characterize a simple continued fraction. For example, the continued-fraction sequence for π is

3, 7, 15, 1, 292, 1, 1, 1, … .

As you’d expect, this sequence is infinite.

There are several important facts about simple continued-fraction sequences:

1. Rational numbers (fractions) have finite sequences. An example is 11/13, which has the sequence 0, 1, 5, 2.
2. Irrational numbers have infinite sequences.
3. Quadratic irrationals have periodic sequences. An example is $\sqrt{7}$, which has the sequence $2, \overline{1, 1, 1, 4}$.
4. All other irrational numbers have non-periodic sequences. The sequence for π, shown above, is an example.
5. There is a one-to-one correspondence between an irrational number and its simple continued-fraction sequence. Furthermore, any periodic sequence of positive integers represents a unique irrational number. (For rational numbers, there are two equivalent sequences: one that ends … a_m, 1 and one that ends … $a_m - 1$.)

Computing Continued Fractions

Continued fractions are closely related to the familiar Euclidean algorithm for computing the greatest common divisor of two integers, *i* and *j*. Euclid’s algorithm might look like this in pseudo-code:

```
until j = 0 do {
   r := remdr(i, j)
   i := j
   j := r
   }
print(i)          # previous value of j
```

The terms in the simple continued faction for i / j consist of values of $i \div j$ (integer division, remainder discarded) in the loop above:

```
until j = 0 do {
 print(i ÷ j)
 r := remdr(i, j)
 i := j
 j := r
 }
```

The problem with trying to compute continued fractions for irrational numbers is that floating-point numbers used by computers to represent real numbers are finite approximations to real numbers, and hence they really are rational numbers whose values are “close” to the corresponding real numbers. For example, the standard 64-bit floating-point encoding for π is

$$7074237752028440 / 2^{51}$$

The corresponding continued-fraction sequence is, of course, finite:

3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 3, 3, 2,
 1, 3, 3, 7, 2, 1, 1, 3, 2, 42, 2

and only the first 13 terms are the same as for the sequence for the actual irrational number:

3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 2, 1, 1,
 2, 2, 2, 2, 1, 84, 2, 1, …

Patterns

Simple continued-fraction sequences for rational numbers usually are short and any patterns are accidental and mostly uninteresting.

Since quadratic irrationals have periodic simple continued-fraction sequences, they have patterns that may be of interest in designing weaves.

Simple continued-fraction sequences for other irrationals are not periodic and most have no evident patterns.

Some, however, do. An example is tan(1) (see Figure Ω.1f), whose simple continued-fraction sequence is

$$1, \overline{1, 2n+1} \qquad n = 1, 2, 3, \ldots$$

Explain overbar notation.

Another example is $e - 1$ (see Figure Ω.1a), whose simple continued-fraction sequence is

$1, \overline{1, 2n, 1}$ $\qquad$ $n = 1, 2, 3, \ldots$

Such sequences have periodic *forms*. The simple continued-fraction sequence for π has no such structure, but there is an ordinary continued-fraction for π/4 (see Figure Ω.1d) that has numerator and denominator sequences with periodic forms:

numerators: $\overline{(2n-1)^2}$ $\quad n = 1, 2, 3, \ldots$

denominators: $1, \overline{2}$

Figures Ω.2 through Ω.4 indicate some possibilities for weaves based on continued fractions..

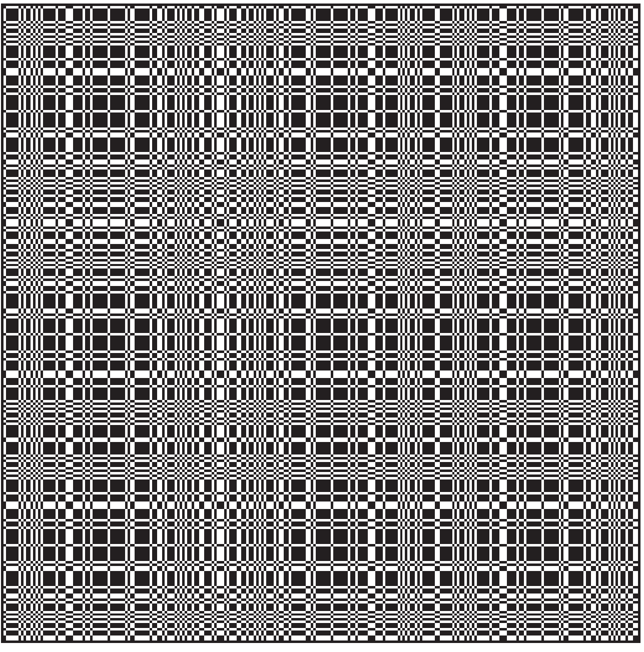


Figure Ω2. $\sqrt{10089}$**, Tabby Tie-Up**

Need ideas for better examples.

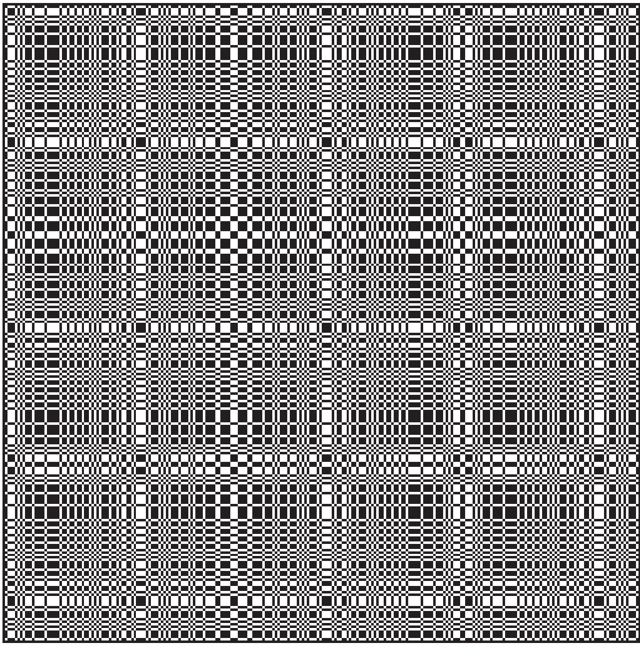


Figure Ω.3. $\sqrt{9949}$**, Tabby Tie-Up**

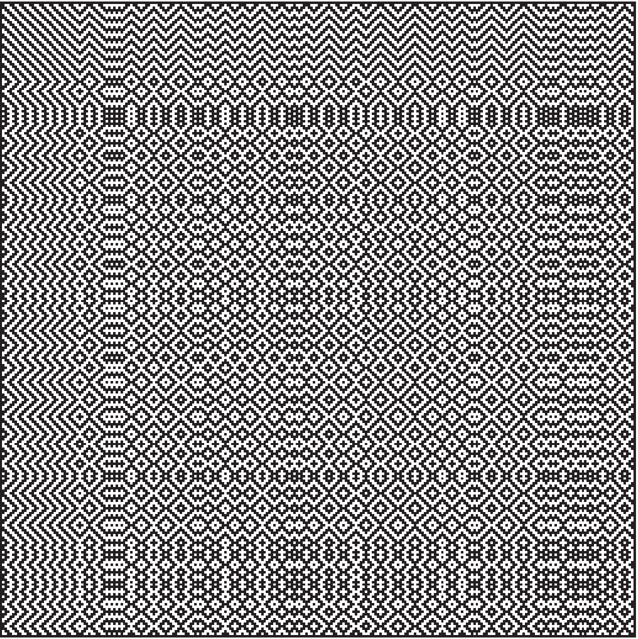


Figure Ω4. $\sqrt{9949}$**, Twill Tie-Up**

Need more discussion of designing with continued fractions.

Learning More About Continued Fractions

Much of the literature about continued fractions is highly technical and specialized. There are, however, a few books that are accessible [1-3]. There also are Web resources [4-5].

Farey Fractions

In this day of hand-held calculators and computers, for most of us fractions are only dim, unpleasant memories of early rote schooling and seemingly pointless, tedious exercises.

Despite the fact that we can get along without all but the simplest fractions for most everyday business, fractions are important in mathematics, the physical sciences, and computer science.

Fractions may seem to be unlikely candidates for design inspiration, but patterns and beauty can be found in the most unexpected places in mathematics.

Farey Fractions

The Farey fractions, named after the British geologist John Farey (1766-1826), provide an example.

The Farey fraction sequence of order i, $\mathfrak{F}(i)$, consists of all fractions with values between 0 and 1 whose denominators do not exceed i, expressed in lowest terms and arranged in order of increasing magnitude. For example, $\mathfrak{F}(6)$ is

$$\frac{0}{1}, \frac{1}{6}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{5}{6}, \frac{1}{1}$$

Farey observed that the fractions in such sequences are the *mediants* of their adjacent fractions. The mediant of n_1/d_1 and n_2/d_2 is

$$(n_1 + n_2)/(d_1 + d_2)$$

which looks like a naive attempt to add fractions.

Farey sequences have a number of other interesting and useful properties [1, 2]. Our interest here, however, is with their use in weave design.

A sequence of fractions can be interpreted as integer sequences in a number of ways. Since the numerators and denominators show distinctive patterns, a natural method is to separate a sequence of fractions into two sequences, one of the numerators and one of the denominators as in:

$$\mathfrak{F}_n(6) = 0, 1, 1, 1, 1, 2, 1, 3, 2, 3, 4, 5, 1$$

$$\mathfrak{F}_d(6) = 1, 6, 5, 4, 3, 5, 2, 5, 3, 4, 5, 6, 1$$

The patterns in Farey sequences can be seen in grid plots, as shown in Figures Ω.1 and Ω.2. The bottom line of a plot corresponds to the smallest value in the sequence.

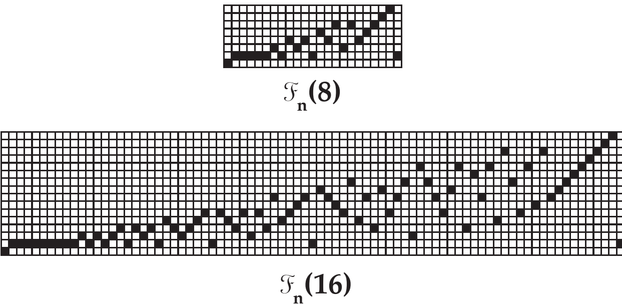


Figure Ω.1. Farey Numerator Sequences

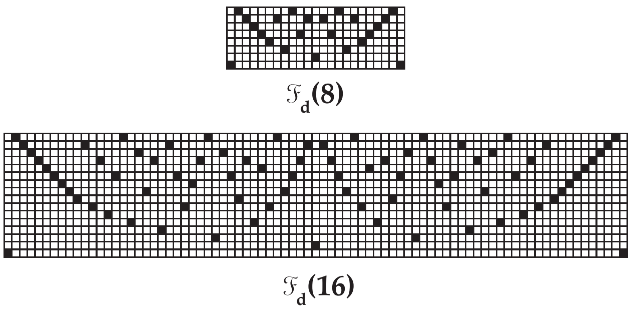


Figure Ω.2. Farey Denominator Sequences

The patterns for other values of *i* are similar in structure. As *i* gets larger, the sequences are longer and the patterns more articulated.

The patterns in the numerator sequences are clear and interesting, although not particularly attractive. The patterns in the denominator sequences, however, are very attractive. Part of this is because these sequences are palindromic, adding the visual appeal of symmetry. A palindrome can be constructed from any sequence, but this one occurs naturally.

Properties of Farey Sequences

Farey sequences have several properties that relate to their appropriateness for weave design.

Both $\mathcal{F}_n(i)$ and $\mathcal{F}_d(i)$ contain *i* different values. The 0 in $\mathcal{F}_n(i)$ can be handled in various ways. One way is to add 1 to all values in the sequence. Another way is to use modular shaft arithmetic [3] with modulus *i*, in which case the 0 is changed to *i* and all other values remain unchanged. In any event, all shafts and treadles in their ranges are utilized.

The distribution of values in the sequences is not balanced, however. The value 0 appears only once in $\mathcal{F}_n(i)$ and the value 2 appears only once in $\mathcal{F}_d(i)$ for

07 Apr 2006 H9: FareyFractions

$i > 1$ (for $i = 1$, 2 does not appear at all, but this is an uninteresting case for weave design). The distributions of other values follow interesting patterns, but that is a deeper topic that we won't consider here.

In $\mathcal{F}_n(i)$ there is a string of 1s of length $\lfloor i/2 \rfloor + 1$ starting with the second value of the sequence, where $\lfloor x \rfloor$ is the integer part of x. No other successive values are the same. For $i > 1$, no successive values in $\mathcal{F}_d(i)$ are the same.

The lengths of Farey sequences increase only modestly with i. There is no simple formula, but the length is about

$$3(i/\pi)^2 \approx 0.304 \times i^2$$

which gives increasingly better approximations as i gets larger [2].

Here are the lengths of $\mathcal{F}(i)$ for $4 \le i \le 32$:

i	*length*	*i*	*length*
4	7	19	121
5	11	20	129
6	13	21	141
7	19	22	151
8	23	23	173
9	29	24	181
10	33	25	201
11	43	26	213
12	47	27	231
13	59	28	243
14	65	29	271
15	73	30	279
16	81	31	309
17	97	32	325
18	103		

Drafting with Farey Sequences

One way to use Farey sequences is directly as threading and treadling sequences. Figures 3 and 4 show drawdown patterns for $\mathcal{F}(8)$ with 8 shafts and 8 treadles, treadled as drawn in. Direct tie-ups were used to make the patterns clear.

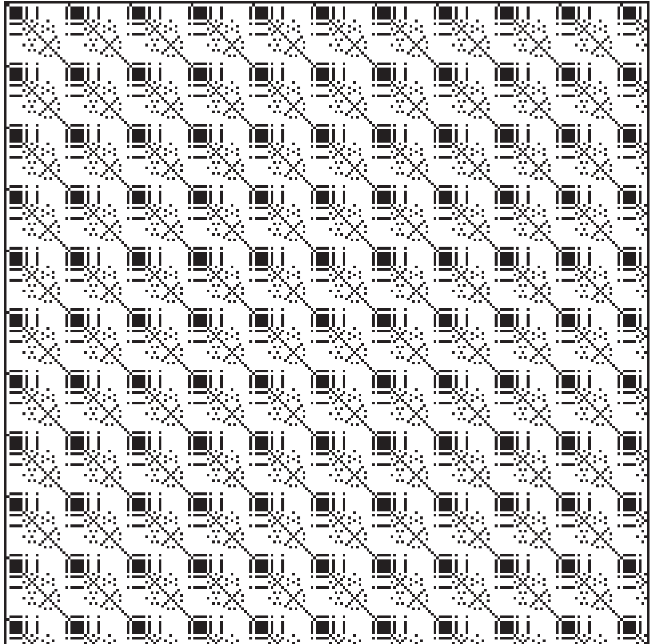


Figure Ω.3. $\mathcal{F}_n(8)$

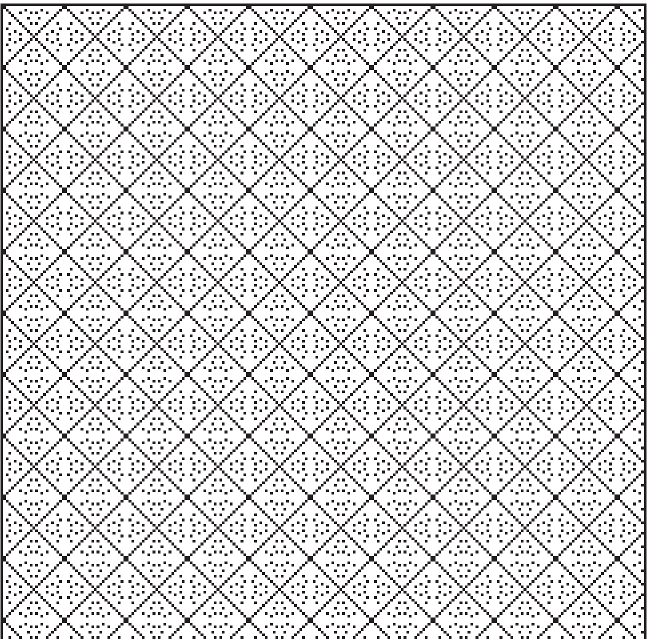


Figure Ω.4. $\mathcal{F}_d(8)$

Figure 5Ω. shows the drawdown pattern for $\mathcal{F}_n(8)$ threading and $\mathcal{F}_d(8)$ treadling.

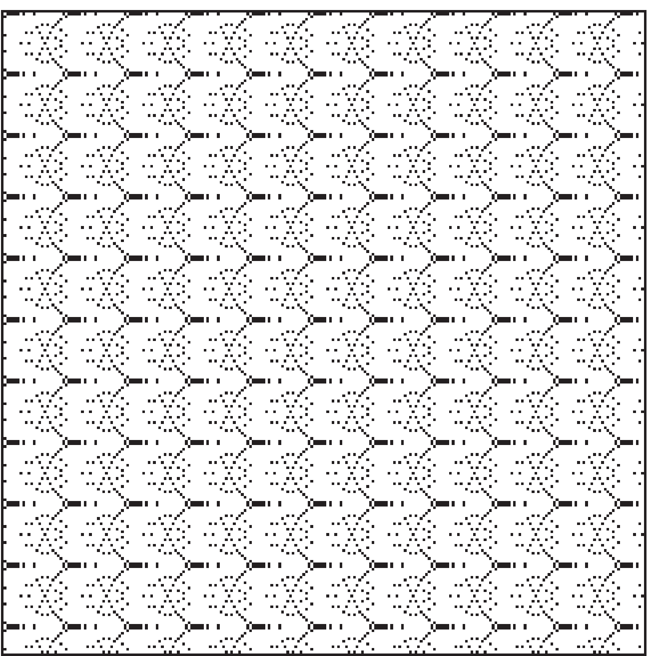


Figure Ω.5. $\mathcal{F}_n(8)$ versus $\mathcal{F}_d(8)$

Direct tie-ups are not suitable for weaving with these sequences for structural reasons. Figures Ω.6-8 show the corresponding drawdown patterns for $\frac{2}{2}$ twill tie-ups.

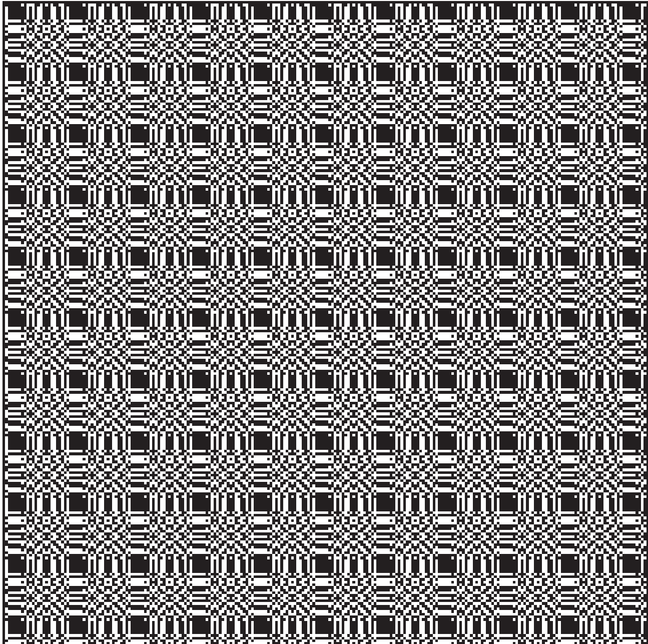


Figure Ω.6. $\mathcal{F}_n(8)$ Twill

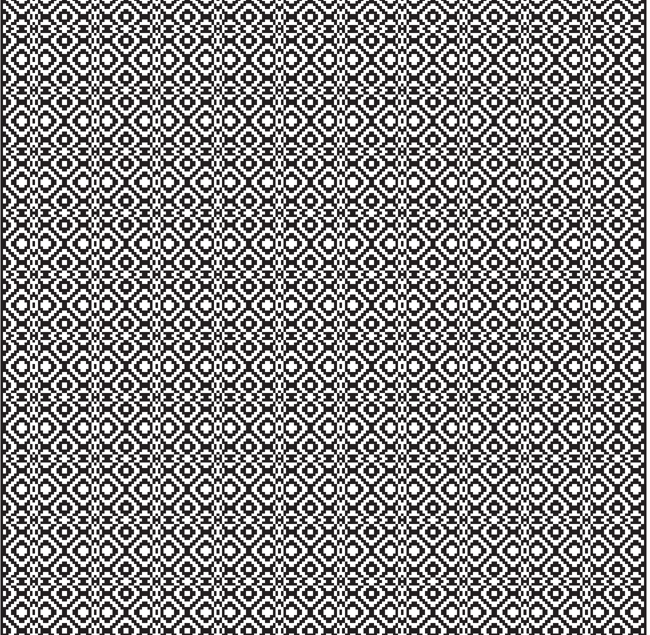


Figure Ω.7. $\mathcal{F}_d(8)$ Twill

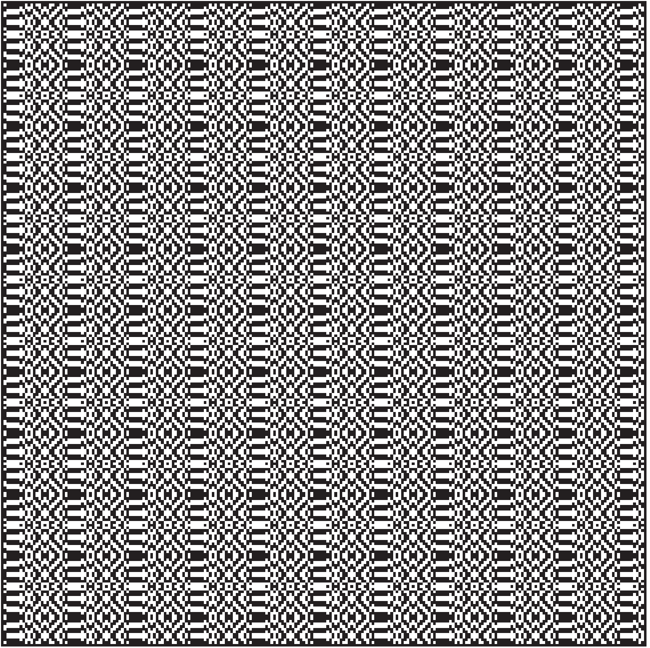


Figure Ω.8. $\mathcal{F}_n(8)$ versus $\mathcal{F}_d(8)$ Twill

Adapting Farey Sequences for Thread-by-Thread Drafts

Modifications often are needed to make sequences from mathematical sources suitable for weaving or to improve the appearance of weaves derived from them.

This usually means doing some violence to the mathematical properties of the sequences, but weave design is, after all, an artistic enterprise — mathematics can only provide inspiration.

Numerator sequences are more troublesome than denominator sequences because numerator sequences have a string of 1s starting at term 2.

One thing to do is to simply remove successive duplicates. This is an easy method that can be applied to all sequences that have successive duplicate values.

Another method is to add incidentals between successive duplicates, analogous to the use of incidentals in name drafting to produce alternating odd/even values for overshot [4].

Yet another method is to change alternative values to break the sequence of duplicates. This has the virtue of maintaining the length of the sequence. For numerator sequences, an attractive method is to change every other 1 into a 0.

See Figures Ω.9 and Ω.10.

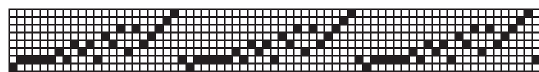


Figure Ω.9. $\mathcal{F}_n(8)$ Repeated

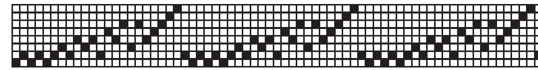


Figure Ω.10. $\mathcal{F}_n(8)$ With Changes Repeated

The drawdown pattern for the changed sequence is shown in Figure Ω.11. Compare this with Figure Ω.3, which shows the pattern without changes.

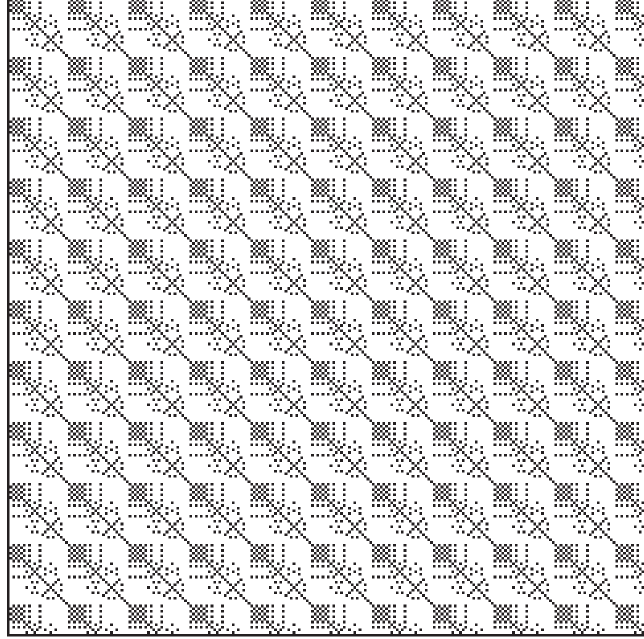


Figure Ω.11. $\mathcal{F}_n(8)$ with Changes

When considering successive duplicates, it is important to look at the first and last values of a sequence, since these become adjacent when the sequence is repeated. Farey numerator sequences have first and last values of 0 and 1, respectively. Note that this meshes with the 0, 1 change method just discussed.

Farey denominator sequences are true palindromes with the same first and last value. For repeats, *pattern palindromes* obtained by removing the last value of a pure palindrome usually are used. Then, of course, a true palindrome for the entire pattern is obtained by appending the first value of the pattern palindrome to the end of the last repeat. The difference this makes in the drawdown pattern

is minor. Compare Figure Ω.12 with Figure Ω.4.

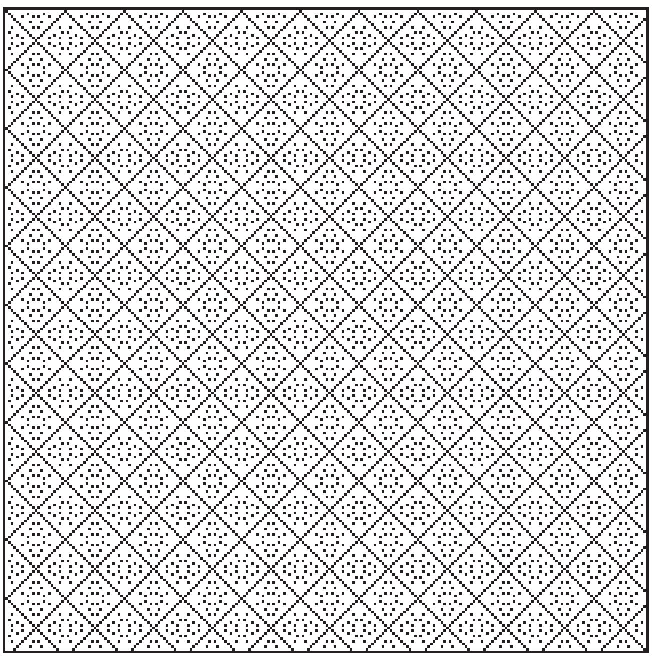


Figure Ω.12. $\mathcal{F}_d(8)$ **with Change**

Drafting Variations

Parameters

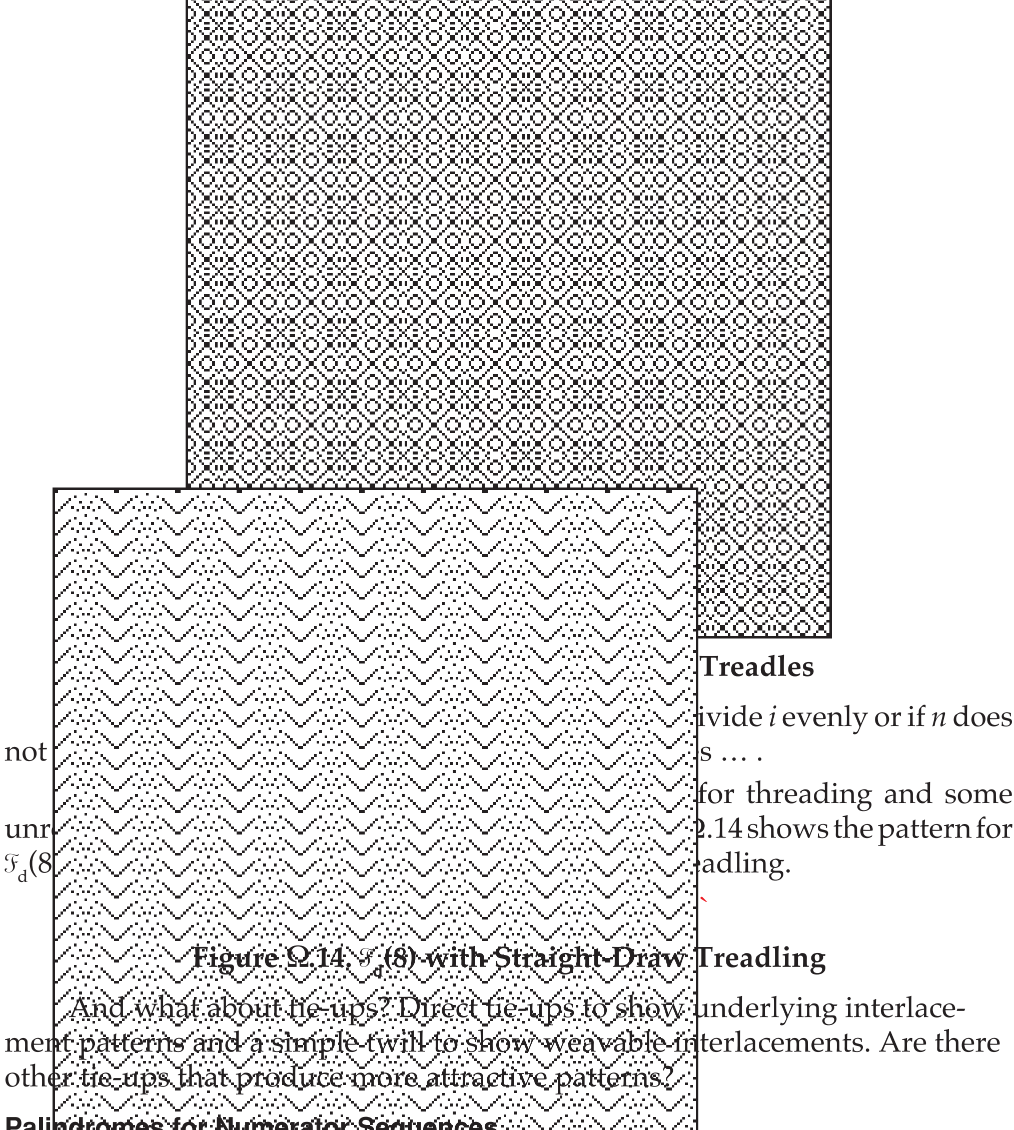
Even with just two sequences used for thread-by-thread drafts, there are endless variations for drafting. The parameters are

i	Farey order for threading
j	Farey order for treadling
m	number of shafts, $\leq i$
n	number of treadles, $\leq j$
t	threading sequence type (numerator or denominator)
u	treadling sequence type (numerator or denominator)

One general question is what happens if $m < i$ and/or $n < j$, assuming modular arithmetic is used to reduce the sequences appropriately.

Figure 13 shows the pattern for $\mathcal{F}_d(8)$ threading and treadling with 4 shafts and 4 treadles.

How about some feedback on tie-ups to use here an the rest of the book?



Treadles

ivide i evenly or if n does not … s … .

for threading and some unr… 2.14 shows the pattern for $\mathcal{F}_d$(8… adling.

Figure Ω.14. $\mathcal{F}_d(8)$ with Straight-Draw Treadling

And what about tie-ups? Direct tie-ups to show underlying interlacement patterns and a simple twill to show weavable interlacements. Are there other tie-ups that produce more attractive patterns?

Palindromes for Numerator Sequences

Farey denominator sequences are palindromic but Farey numerator sequences are not. Another design possibility is to form palindromes by reflecting numerator sequences and use them alone or in combination with denominator sequences.

Figure 15 shows the pattern for $\mathcal{F}_n(8)$, reflected and treadled as drawn in. Figure 16 shows the pattern for $\mathcal{F}_n(8)$, reflected for threading and $\mathcal{F}_d(8)$ for treadling.

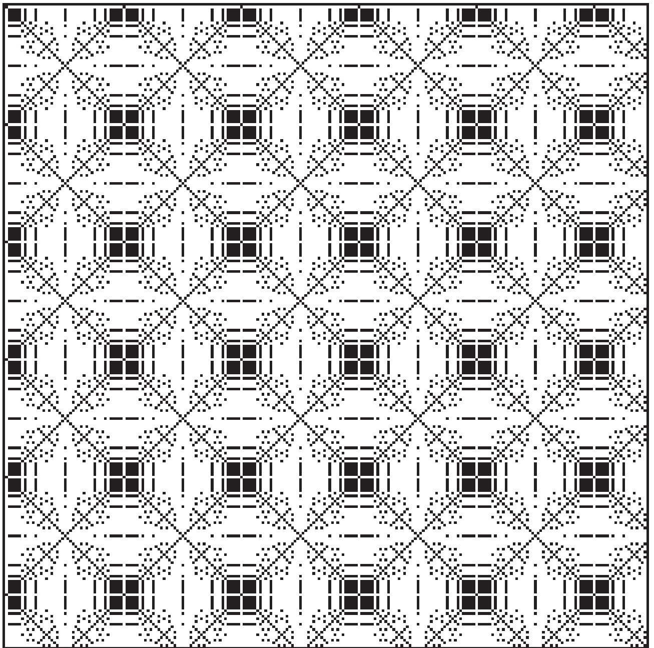


Figure Ω.15. $\mathfrak{F}_n(8)$ Reflected

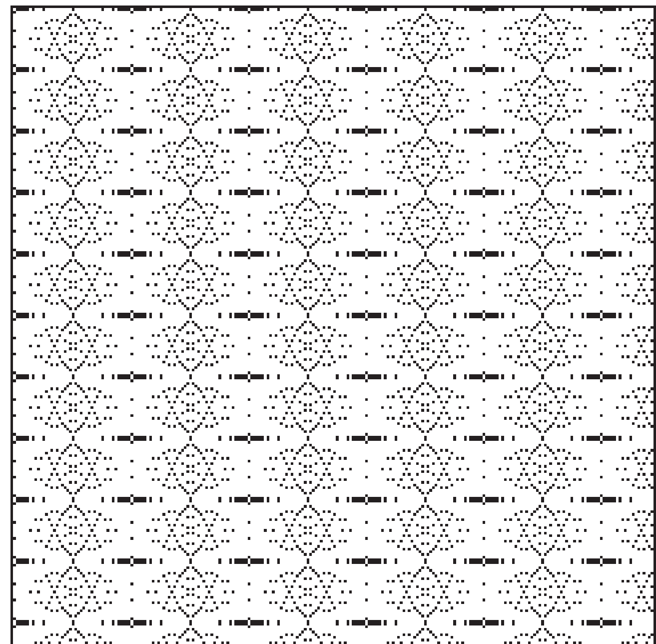


Figure Ω.16. $\mathfrak{F}_n(8)$ Reflected Threading Versus $\mathfrak{F}_d(8)$

Interleaved Sequences

Another possibility is to interleave the numerator and denominator sequences. For $\mathfrak{F}_n(6)$ the result is

0, 1, 1, 6, 1, 5, 1, 4, 1, 3, 2, 5, 1, 2, 3, 5, 2, 3, 3,
4, 4, 5, 5, 6, 1, 1

Figure Ω.17 shows the pattern for $\mathfrak{F}_n(8)$ and $\mathfrak{F}_n(8)$ interleaved, treadled as drawn in with a direct tie-up. Figure 18 shows the pattern for the corresponding $\frac{2}{2}$ twill.

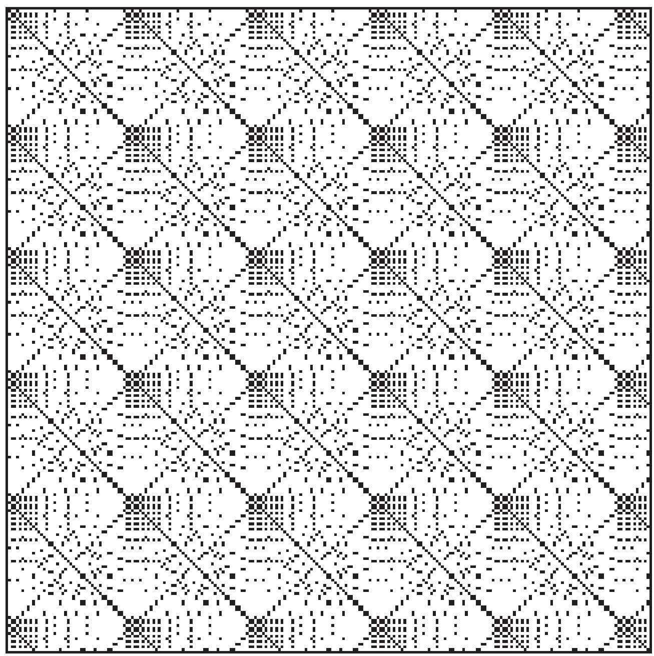


Figure Ω.17. $\mathfrak{F}_n(8)$ and $\mathfrak{F}_d(8)$ Interleaved

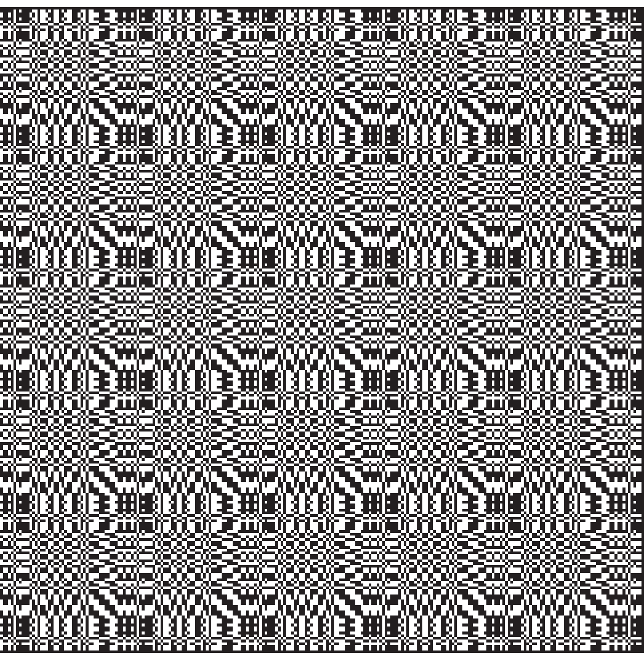


Figure Ω.18. $\mathcal{F}_n(8)$ and $\mathcal{F}_d(8)$ Interleaved Twill

Combining Farey Sequences of Different Orders

So far we've only shown patterns based on Farey Sequences of the one order. Figure 19 shows the pattern that results of concatenating Farey denominator sequences of orders 1 through 8, treadled as drawn in with a direct tie-up.

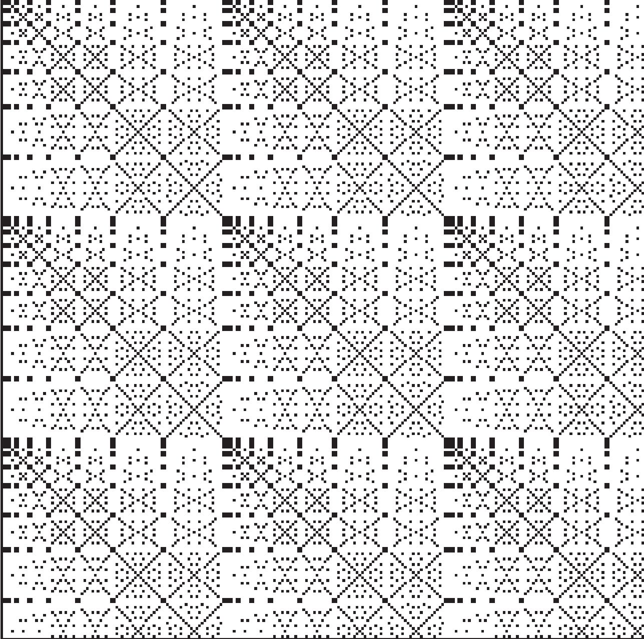


Figure 19. $\mathcal{F}_d(1)$, $\mathcal{F}_d(2)$, … , $\mathcal{F}_d(8)$

You can figure out the pattern if this sequence is reflected to form a pattern palindrome.

Point Twills

While the direct use of Farey sequences for threading and treadling produces interesting results, interpreting the values in the sequences as inflection points for point draws is more promising. In such an interpretation, only high and low values in runs are considered. For example,

1, 1, 2, 5, 3, 1

has the inflection points 1, 5, and 1.

Figure Ω.20 shows grid plots for $\mathcal{F}_d(8)$ point draws.

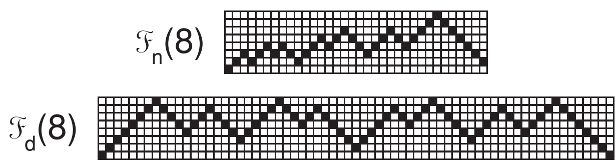


Figure Ω.20. $\mathcal{F}(8)$ **Point Draws**

Figures Ω.21 and Ω.22 show the patterns for the corresponding $\frac{2}{2}$ point twills, treadled as drawn in.

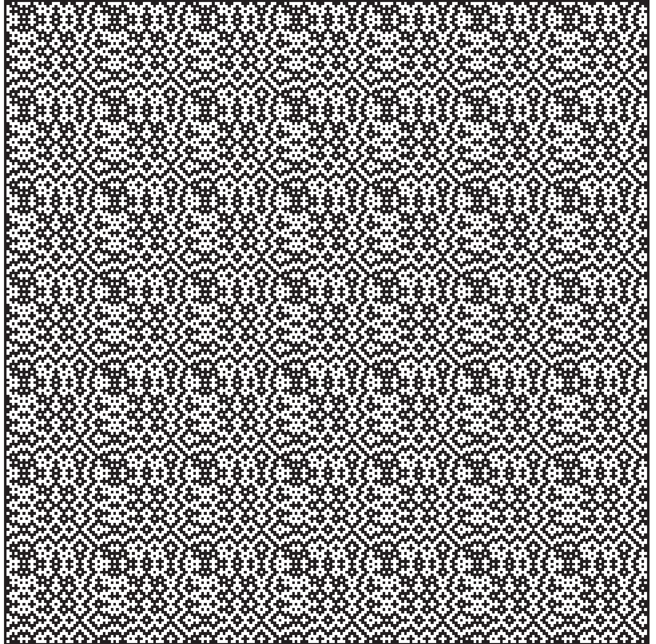


Figure Ω.21. $\mathcal{F}_n(8)$ **Point Twill**

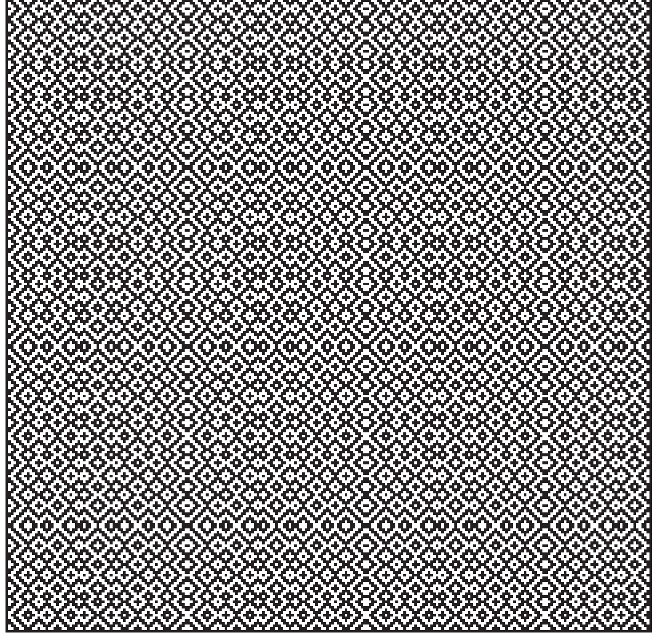


Figure Ω.22. $\mathcal{F}_d(8)$ **Point Twill**

The lengths of point draws are, of course, considerably longer than the lengths of the sequences from which they are derived:

i	$\mathcal{F}_n(i)$ *point length*	$\mathcal{F}_d(i)$ *point length*
4	6	11
5	12	23
6	14	27
7	26	51
8	34	67
9	48	95
10	56	111
11	86	171
12	98	195
13	140	279
14	158	315
15	186	371
16	218	435
17	290	579
18	316	631
19	406	811
20	446	891
21	506	1011
22	556	1111
23	688	1375
24	736	1471
25	862	1723
26	934	1867
27	1056	2111
28	1140	2279
29	1350	2699
30	1414	2827
31	1654	3307
32	1782	3563

Other Possibilities

The wide range of possibilities touched on here does not begin to exhaust the potential of Farey fraction design.

Color specification, for example, always is a design possibility for sequences.

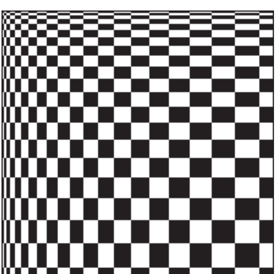
And Farey fraction design is not limited to weaving [5].

Resources

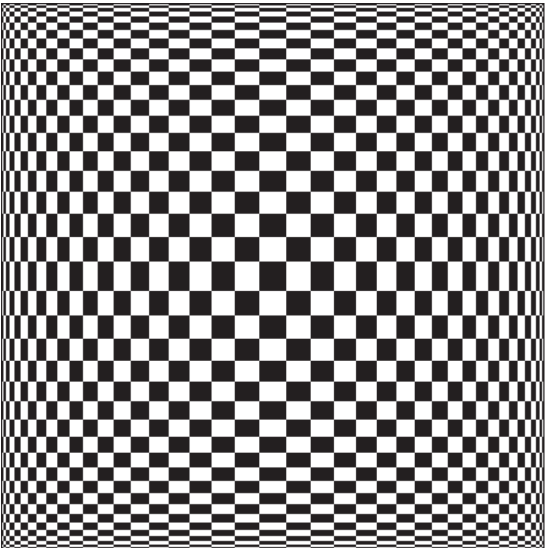
If you want to experiment with Farey sequences, you'll find sequence data at Reference 6 and 7.

Term Replication Sequences

Patterns like this one have a visual fascination:



This fascination is enhanced by mirroring:



These patterns come from a very simple sequence:

1 2 2 3 3 3 4 4 4 4 5 5 5 5 5 …

For example, if this sequence is used for the threading and treadling sequences in a weaving draft with a tabby tie-up, the resulting drawdown is as shown in image above.

In the sequence above, each term is replicated according to its value. There is no accepted good name for this sequence. It was called the multi sequence in previous articles [1] and the On-Line Encyclopedia of Integer Sequences [2] refers to it as "*n* appears *n* times", which is descriptive but far from elegant.

The sequence above is one of a class of sequences obtained by applying *term replication functions* to *bases sequences*.

For the example above, the base sequence is the positive integers, $I^{+} = 1\ 2\ 3\ 4\ 5\ \ldots$ and the replication function is $\mathrm{r}(v) = v$, where v is the value of the term.

If the base sequence is the Fibonacci numbers, $F = 1\ 1\ 2\ 3\ 5\ 8\ \ldots$, then this rule yields

1 1 2 2 3 3 3 5 5 5 5 5 8 8 8 8 8 8 8 8 …

Compact Representations of Term Replication

Sequences in which terms are replicated may be difficult to understand if terms are written out in the usual fashion.

One way to reduce visual clutter is to list replicated terms only once along with their replication factor. The notation

$\underline{i}_j$

indicates that there are j copies of i. Thus, the result of applying $\mathrm{r}(v) = v$ to I^{+} and the primes, $P = 2\ 3\ 5\ 7\ \ldots$, can be written as

$\underline{1}_1\ \underline{2}_2\ \underline{3}_3\ \underline{4}_4\ \underline{5}_5\ \ldots$
$\underline{2}_2\ \underline{3}_3\ \underline{5}_5\ \underline{7}_7\ \ldots$

Another way to represent the results of applying a replication function to a base sequence is to write the base sequence above the replication sequence, with a bar separating the two. For the examples above, the representations are

$$\frac{1\ 2\ 3\ 4\ 5\ \ldots}{1\ 2\ 3\ 4\ 5\ \ldots}$$

$$\frac{2\ 3\ 5\ 7\ \ldots}{2\ 3\ 5\ 7\ \ldots}$$

For named sequences, a simpler, linear typographical form can be used, as in P/F.

Value-Based Replication Functions

Replication functions whose values are determined solely by term values are called value-based.

Many kinds of value-based replication functions are possible, such as the following:

$$r(v) = 1 \qquad [1]$$

$$r(v) = v \qquad [2]$$

$$r(v) = v + 2 \qquad [3]$$

$$r(v) = v \text{ smod } 5 \qquad [4]$$

$$r(v) = \begin{cases} 1 & v \text{ even} \\ 2 & v \text{ odd} \end{cases} \qquad [5]$$

$$r(v) = \begin{cases} 1 & v \text{ even} \\ 0 & v \text{ odd} \end{cases} \qquad [6]$$

Eqn. 1 leaves the base sequence unchanged. Eqn. 2 produces the results described previously. Eqn. 3 is like Eqn. 2 except that 2 replications are added. In Eqn. 4, the replication factor is reduced shaft-modulo 5 [1], so that values whose residues are 1 smod 5 are not replicated, values whose residues are 2 smod 5 are replicated two times, and so on.

In Eqns. 5 and 6, the result depends on the parity of the value. In Eqn. 5 even values are not replicated, while odd ones are duplicated. In Eqn. 6, even values are not replicated and odd values are discarded (being replicated 0 times).

Note that in Eqns. 2 and 3, replication factors increase without limit as v does. In the other equations, the replication factors are bounded regardless of how large v is.

Position-Based Replication Functions

Replication factors can be based on the positions of terms instead of their values, position being the number of the term in the sequence. For example, in *P*, 2 is term 1, 3 is term 2, 5 is term 3, 7 is term 4, and so on.

For example, if p is the position of a term in a sequence, the replication function

$$r(p) = \begin{cases} 1 & p \text{ odd} \\ 2 & p \text{ even} \end{cases} \qquad [7]$$

25 Jun 2006 H10: TermReplication

doubles even-numbered terms but not the odd-numbered terms.

The replication function

$$\mathrm{r}(p) = p \qquad [8]$$

replicates by the position of the term. For I^+, Eqn. 8 produces the same results as Eqn. 2. For P, it produces

$$2\ \underline{3}_2\ \underline{5}_3\ \underline{7}_5 \ldots$$

Value- and Position-Based Replication Functions

Replication functions can depend both on value and position. An example is

$$\mathrm{r}(v, p) = \begin{cases} v & p \text{ odd} \\ p & p \text{ even} \end{cases} \qquad [9]$$

For F, Eqn. 9 produces

$$\underline{1}_3\ \underline{2}_2\ \underline{3}_3\ \underline{5}_5\ \underline{8}_6 \ldots$$

Replication Sequences

Replication factors can be determined independently of the base sequence. For example, for the base sequence I^+ and the replication sequence P,

$$I^+ / P =$$

$$\frac{1\ 2\ 3\ 4 \ldots}{2\ 3\ 5\ 7 \ldots} =$$

$$\underline{1}_2\ \underline{2}_3\ \underline{3}_5\ \underline{4}_7 \ldots =$$

$$1\ 1\ 2\ 2\ 2\ 3\ 3\ 3\ 3\ 3\ 4\ 4\ 4\ 4\ 4\ 4\ 4 \ldots$$

As another example, consider the 1-based Morse-Thue sequence [3], M = 1 2 2 1 2 1 1 2 … as the base sequence and I^+ as the replication sequence:

$$M / I^+ =$$

$$\frac{1\ \ 2\ \ 2\ \ 1\ \ 2\ \ 1\ \ 1\ \ 2 \ldots}{1\ \ 2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8} =$$

$1\ \underline{2}_5\ \underline{1}_4\ \underline{2}_5\ \underline{1}_{13}\ \underline{2}_8 \ldots =$

1 2 2 2 2 2 1 1 1 1 2 2 2 2 2 1 1 1 1 1 1 1 1
1 1 1 1 1 2 2 2 2 2 2 2 2 …

Term-Replication Sequence Patterns

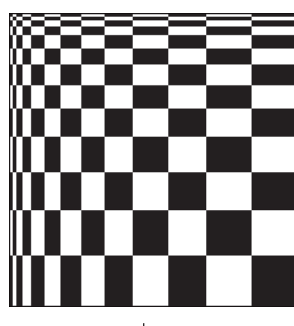
Patterns derived from term-replication sequences may not be suitable, as-is, for interlacement patterns in weaving for structural reasons. Such patterns, however, may make good block patterns for profile drafting.

As in all such things, designing good patterns based on term replication requires a combination of experience, skill, and creativity.

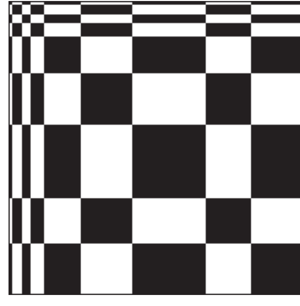
The next two pages show some examples that can be used as a basis for experimentation. Following two pages show some examples of mirrored patterns based on term replication sequences.

All the examples in the appendices are produced using tabby tie-ups with treadling as drawn in. Hint, hint … .

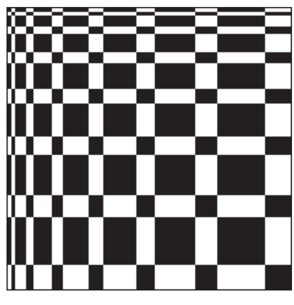
Basic Patterns Derived from Term-Replication Sequences



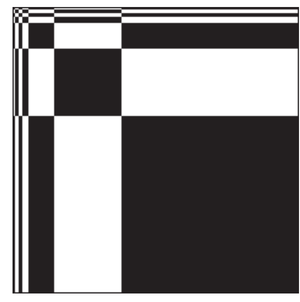
I^+ / P



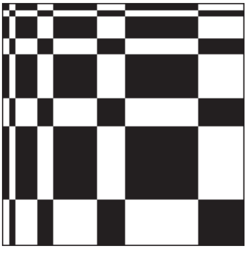
M / P



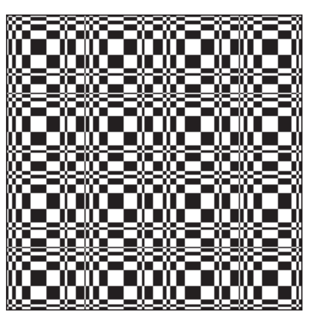
F / I^+



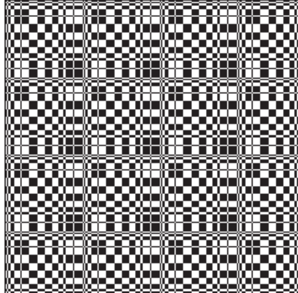
M / F



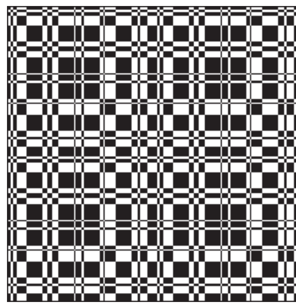
F / P



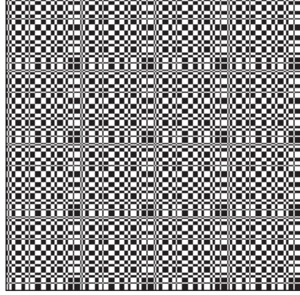
M / (F smod 7)



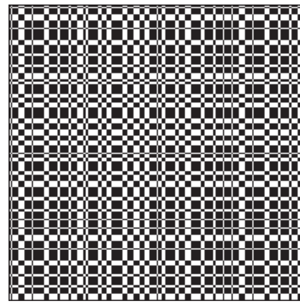
I / (*F* smod 7)



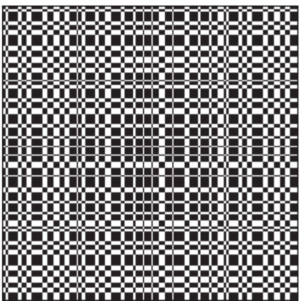
(*F* smod 3) / (*F* smod 5)



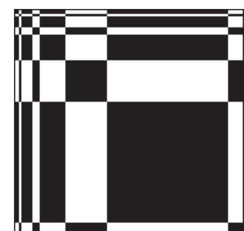
I / (*F* smod 5)



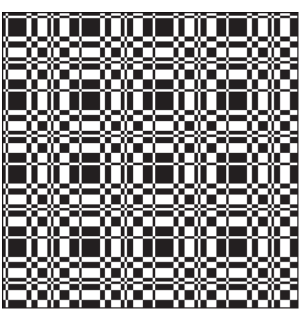
(I^+ smod 3) / (*F* smod 5)



F / (*F* smod 5)

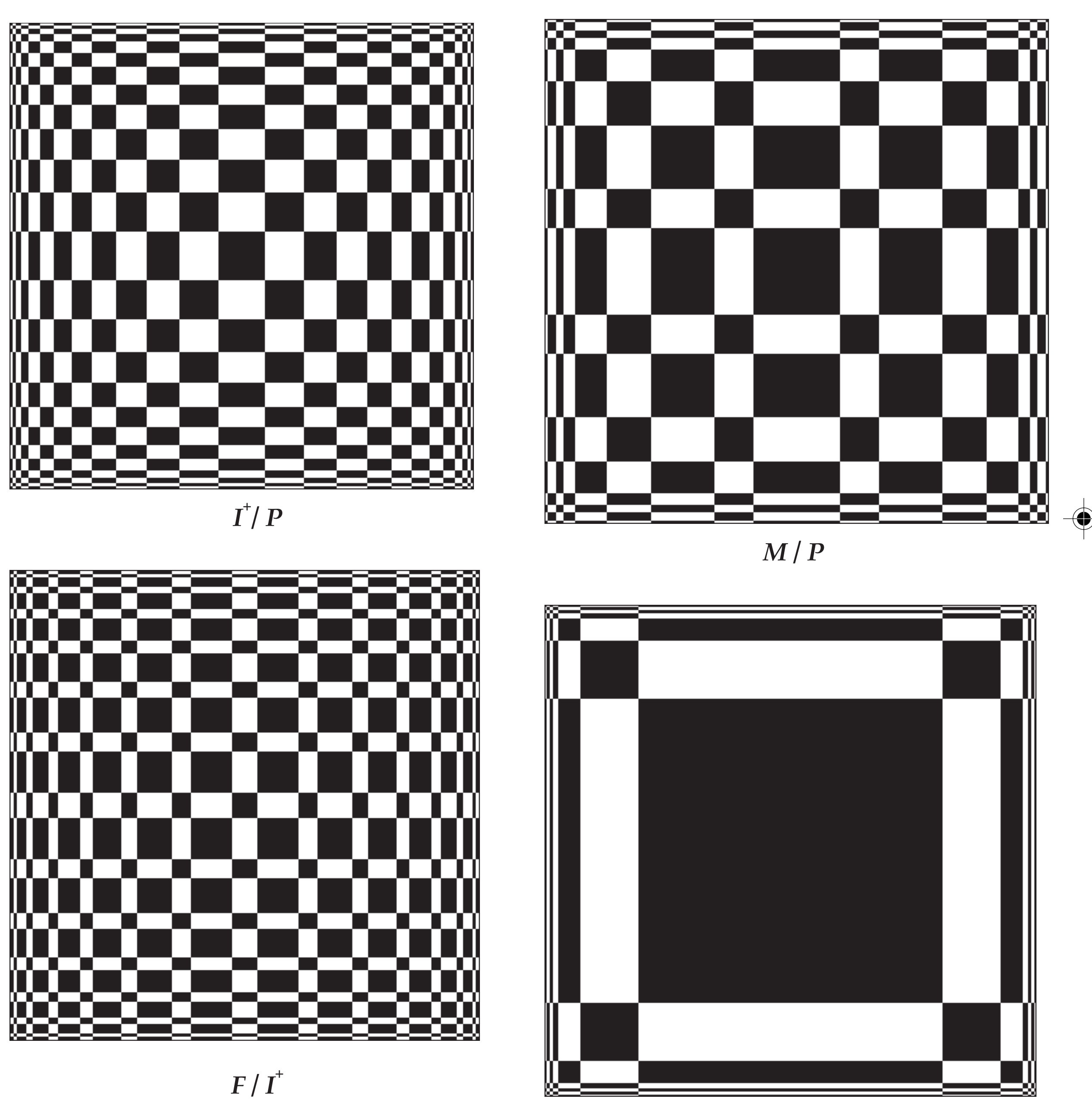


Eqn. 8 Applied to *F*



(*F* smod 5) / (*F* smod 5)

Mirrored Patterns Derived from Term-Replication Sequences



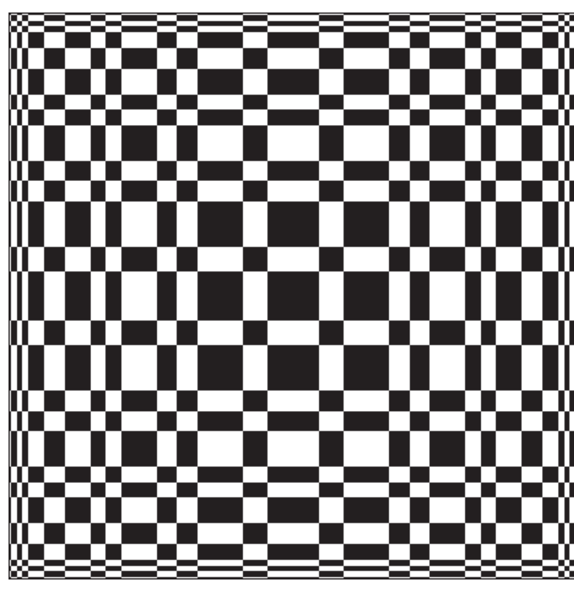
I^{+}/P

M/P

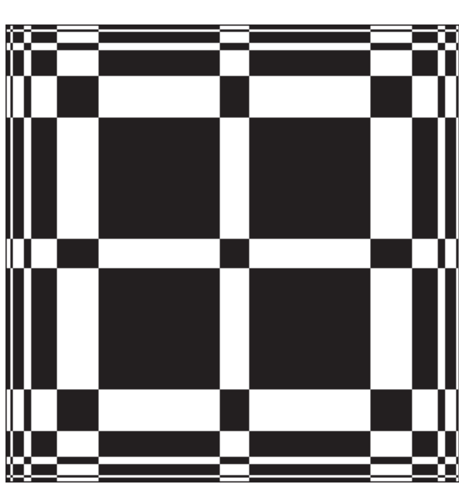
F/I^{+}

M/F

nces



M / I



Eqn. 8 Applied to *F*

Algebraic Expressions

Mathematics is often defined as the science of space and number. …it was not until the recent resonance of computers and mathematics that a more apt definition became fully evident: mathematics is the science of patterns.

— Lynn Arthur Steen

Ada Dietz introduced a novel method of weave design in her seminal monograph *Algebraic Expressions in Handweaving* [1]. Her idea was to use multivariate polynomials (polynomials in several variables) raised to different powers to produce sequences that could be used as the basis for design. Such design sequences can be used as profile sequences, color sequences, and so on [2-7].

Dietz Polynomials

The polynomials Ada Dietz used consist of the sum of variables with unit coefficients raised to a power. An example is $(a + b + c)^3$. **Note:** Standard mathematical notation uses italic lowercase letters at the end of the alphabet, such as x, y, and z, for variables, and roman lowercase letters at the beginning of the alphabet, such as a, b, and c for constants. The use of letters here is deliberately different, since in many uses, variables correspond to blocks, for which the first letters of the alphabet usually are used.

The number of variables used corresponds to the number of blocks desired, while the power to which the polynomial is raised corresponds to the "degree of interaction" among the blocks.

For example, in $(a+b+c+d)^2$ there are four blocks, a, b, c, and d, with a small amount of interaction, while in $(a+b)^5$, there are two blocks, a and b, with a large amount of interaction.

Design sequences are constructed from such expressions in the following way. First, the polynomial is multiplied out, combining like terms, to give the individual terms:

1: $(a + b + c + d)^2 = a^2 + 2ab + 2ac + 2ad + b^2 + 2bc + 2bd + c^2 + 2cd + d^2$

2: $(a + b)^5 = a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + b^5$

Next, powers are replaced by products of variables:

1: $a^2 + 2ab + 2ac + 2ad + b^2 + 2bc + 2bd + c^2 + 2cd + d^2 =$
$aa + 2ab + 2ac + 2ad + bb + 2bc + 2bd + c + 2cd + dd$

2: $a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + b^5 =$

03 Jul 2006 H11: AlgebraicExpressions

aaaaa + 5aaab + 10aaabb + 10aabbb + 5abbbb + bbbbb

Next, numerical coefficients are interpreted as repetitions of the variables that follow them:

1: *aa + 2ab + 2ac + 2ad + bb + 2bc + 2bd + cc +2cd + dd* →
aa + abab + acac + adad + bb + bcbc + bdbd + cc + cdcd + dd

2: *aaaaa + 5aaab + 10aaabb + 10aabbb + 5abbbb + bbbbb* →
aaaaa + aaabaaabaaab + aaabbaaabbaaabbaaabbaaabbaaabbaaabbaaabbaaab_
baaabb + aabbbaabbbaabbbaabbbaabbbaabbbaabbbaabbbaabbbaabbb +
abbbbabbbbabbbbabbbbabbbb + bbbbb

An underscore indicates a term that is too long to fit on the current line and is continued onto the next.

Note that this transformation produces a result that is not mathematically equivalent to the previous expression, since, for example, $abab = a^2b^2$, not $2ab$. The use of → above instead of = indicates the result is not mathematically equivalent.

Finally, the terms are concatenated to produce a profile sequence:

1: *aa + abab + acac + adad + bb + bcbc + bdbd + cc +cdcd + dd* →
aaababacacadadbbbcbcbdbdcccdcddd

2: aaaaa + aaabaaabaaab + aaabbaaabbaaabbaaabbaaabbaaabbaaabbaaabbaaabbaaabb +
aabbbaabbbaabbbaabbbaabbbaabbbaabbbaabbbaabbbaabbb +
abbbbabbbbabbbbabbbbabbbb + bbbbb →
aaaaaaaaabaaaabaaaabaaaabaaaabaaabbaaabbaaabbaaabbaaabbaaabbaaabba_
aabbaaabbaaabbaabbbaabbbaabbbabbbaabbbaabbbaabbbaabbbaabbbaabbbabbbb_
abbbbabbbbabbbbabbbbbbbbb

Although the procedure described above is not mathematically sound, it is unambiguous if somewhat arbitrary.

The actual variables used are just names and are used to stand for things like blocks and colors.

The pattern of variables in the result depends on the ordering of the variables and terms. Conventional mathematical practice is followed in the development above. Variables in the polynomials are in alphabetical order from left to right and products in terms are written in the order of the variables. Furthermore, the variables in the terms are ordered lexically (in dictionary order). For example, *aa* appears before *aab*, and *aab* appears before *aabb*. Other orderings could be used, but for uniformity, strict lexical ordering is used in the examples here.

Computing Dietz Polynomials

What is going on in deriving design sequences from polynomials is easier to see if the simplifications that usually are performed in multiplying out products of polynomials are bypassed and do not use powers or combine like terms.

A simple example is $(a + b)^2$, which conventionally is multiplied out to give $a^2 + 2ab + b^2$. Instead, the multiplication process, without the use of powers and combining like terms, looks like this

$$\begin{array}{r} a + b \\ \underline{a + b} \\ ab + bb \\ \underline{aa + ab \quad\quad} \\ aa + ab + ab + bb \end{array}$$

which directly yields *aaababbb*.

So the steps in the Dietz process amount to removing simplifications usually made in polynomial arithmetic. When computing polynomial design sequences by hand, the easiest method is to avoid the simplifications usually made, going more directly to the end result (being careful to keep terms separated and in the correct order).

Design Sequence Lengths

Dietz design sequences become quite long, especially when the power ("degree of interaction") is large. Here is a table showing lengths for various numbers of variables and powers:

variables	*power*	*length*
1	1	1
1	2	2
1	3	3
1	4	4
1	5	5
1	6	6
	…	
2	1	2
2	2	8
2	3	24
2	4	64
2	5	160
2	6	384
	…	

3	1	3
3	2	18
3	3	81
3	4	324
3	5	1215
3	6	4374
	…	
4	1	4
4	2	32
4	3	192
4	4	1024
4	5	5120
4	6	24576
	…	
5	1	5
5	2	50
5	3	375
5	4	2500
5	5	15625
5	6	93750
	…	
6	1	6
6	2	72
6	3	648
6	4	5184
6	5	38880
6	6	279936
	…	
7	1	7
7	2	98
7	3	1029
7	4	9604
7	5	84035
7	6	705894
	…	
8	1	8
8	2	128
8	3	1536
8	4	16384
8	5	163840
8	6	1572864
	;	

9	1	9
9	2	162
9	3	2187
9	4	26244
9	5	295245
9	6	3188646
	…	

Sequences whose lengths are greater than several hundred are not good candidates for weave design, although parts of them may be.

Interlacement Patterns

There are many ways these sequences can be used in design, a subject we'll take up in a subsequent article.

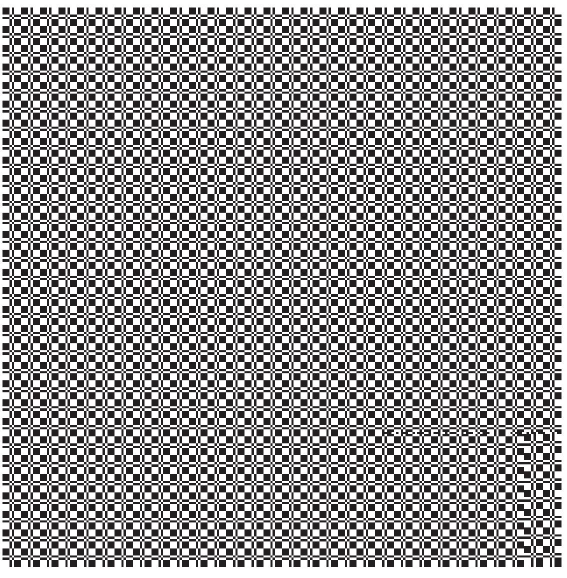
An understanding of the nature of these sequences can be obtained by using them as threading and treadling sequences.

Interlacement patterns for patterns (drawdown images) for various Dietz polynomials are shown on the following pages. In these patterns, the variables a, b, c, … are assigned the shafts 1, 2, 3, … . Direct tie-ups are used and the treadling is as drawn in.

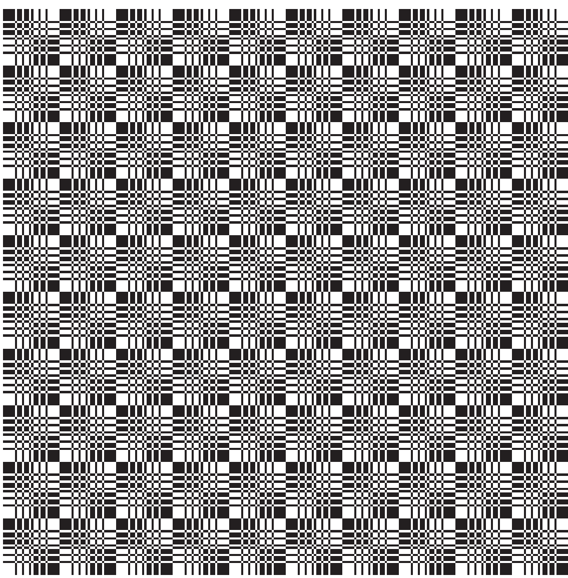
Note how the patterns change down the columns as the powers increase and across the rows of successive pages as the number of variables (and hence shafts and treadles) increases.

The patterns show 240 ends and picks. As the power and number of variables increase, some patterns do not show a full repeat. See the table of sequence lengths given on the previous page. [More to come.]

Examples

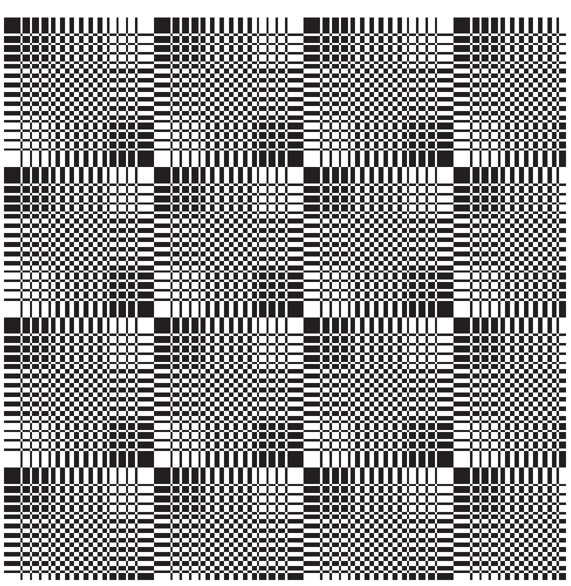


$(a + b)^2$

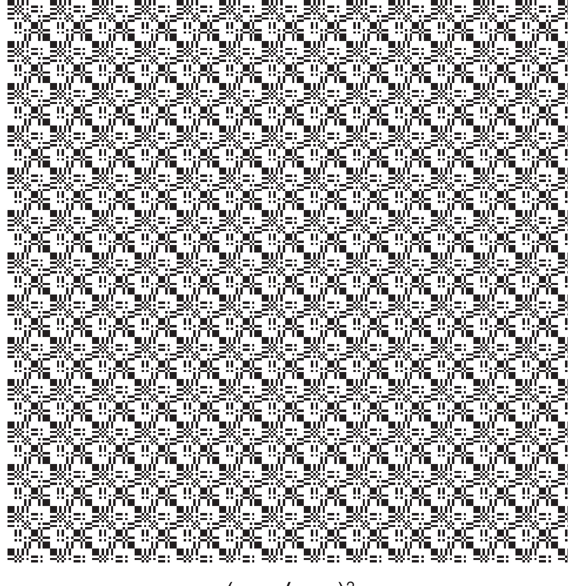


$(a + b)^3$

Examples

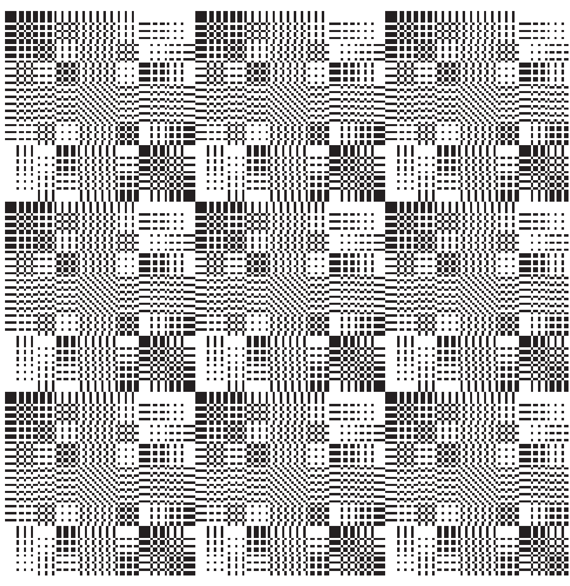


$(a + b)^4$

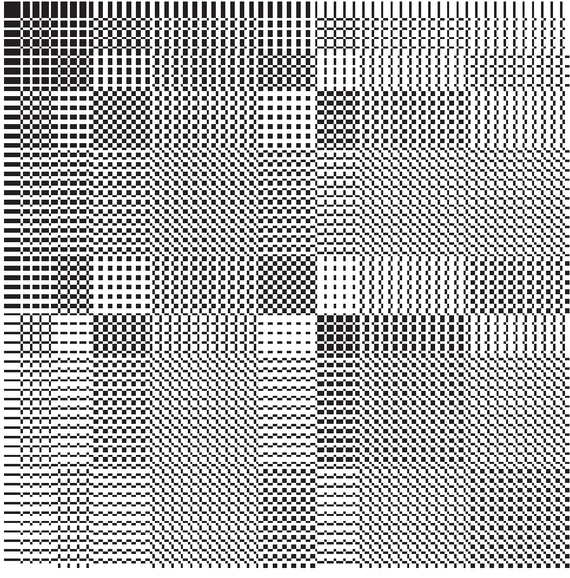


$(a + b + c)^2$

Examples

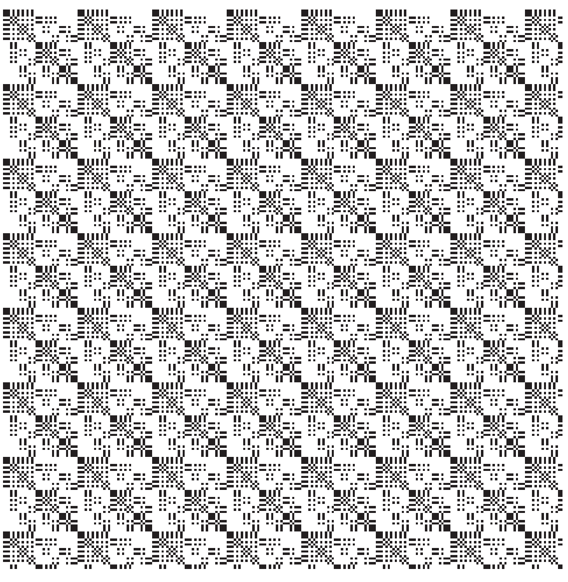


$(a + b + c)^3$

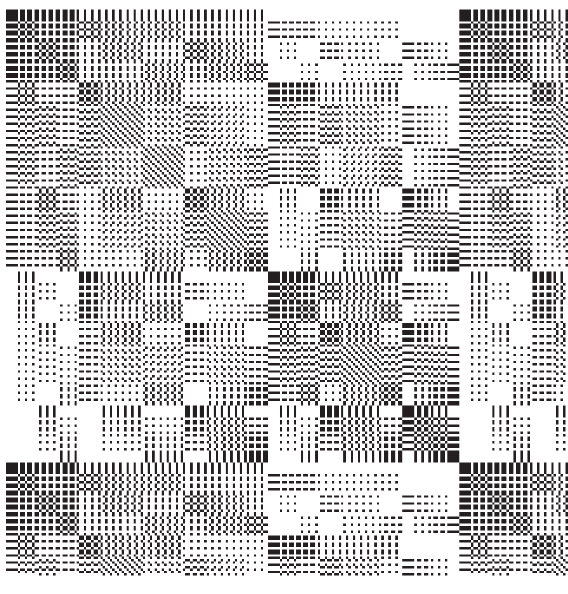


$(a + b + c)^4$

Examples

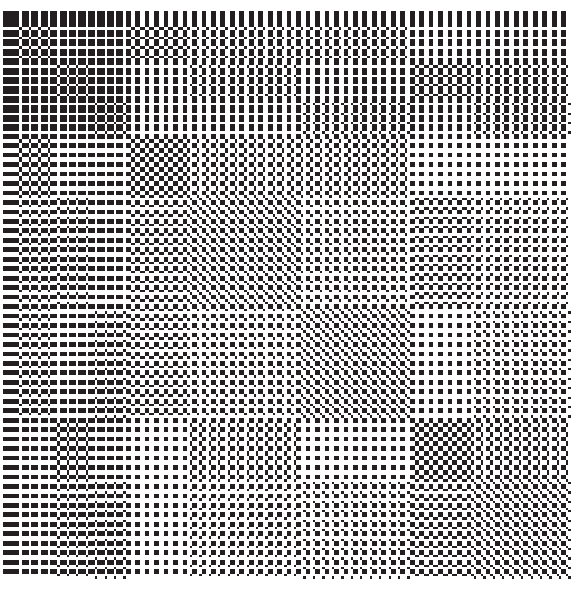


$(a + b + c + d)^2$

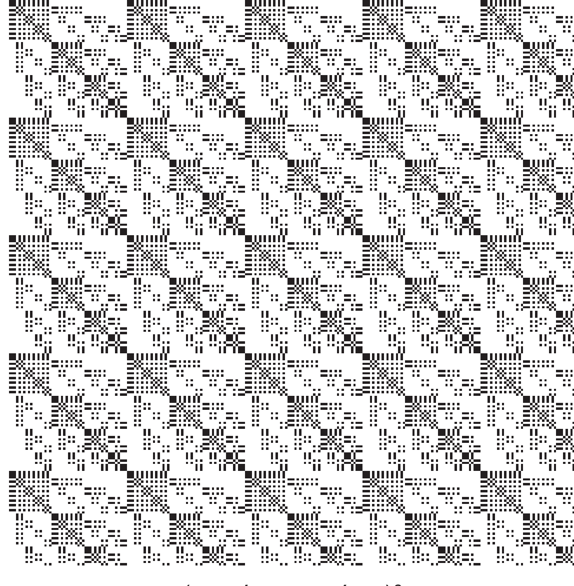


$(a + b + c + d)^3$

Examples

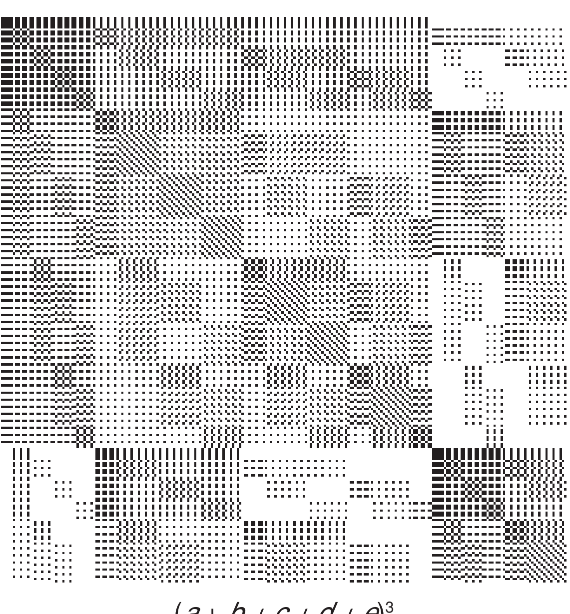


$(a + b + c + d)^4$

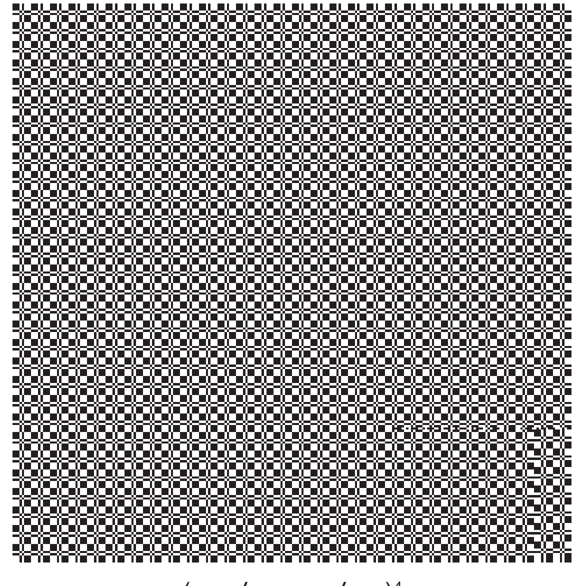


$(a + b + c + d + e)^2$

Examples

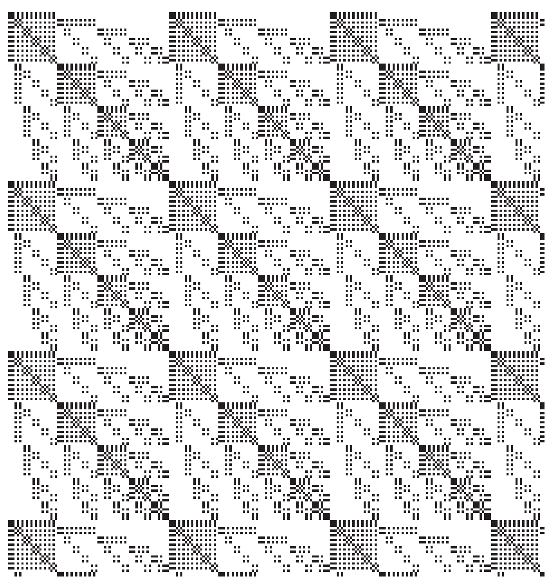


$(a + b + c + d + e)^3$

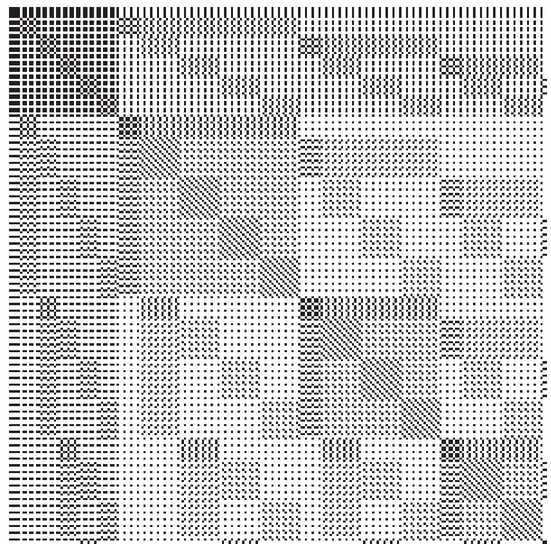


$(a + b + c + d + e)^4$

Examples

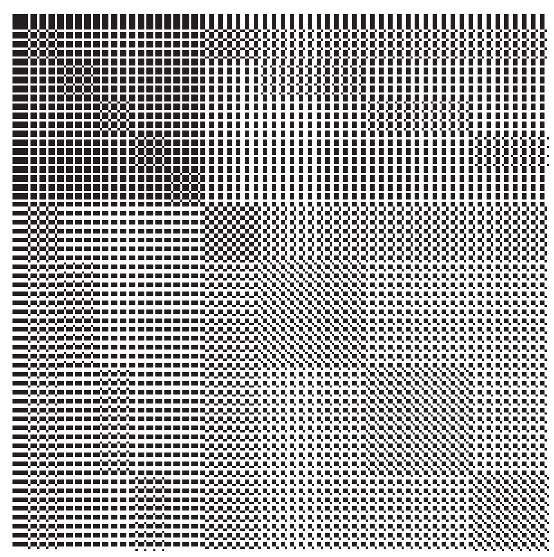


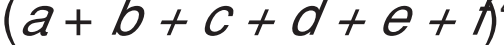
$(a + b + c + d + e + f)^2$



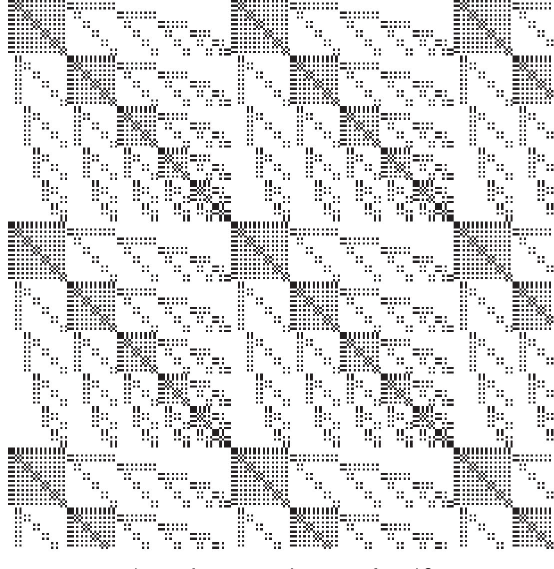
$(a + b + c + d + e + f)^3$

Examples





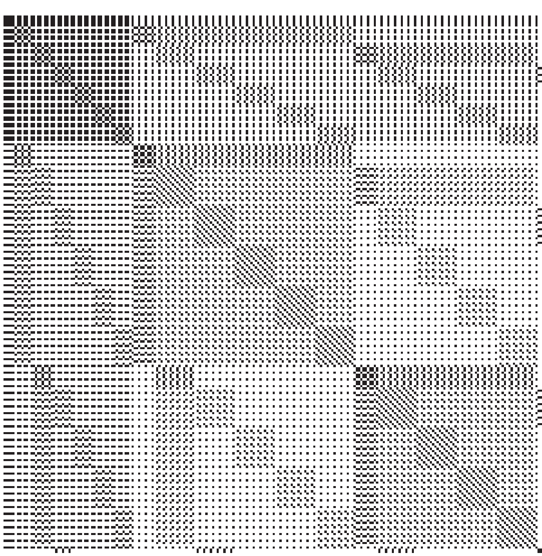
$(a+b+c+d+e+f)^4$



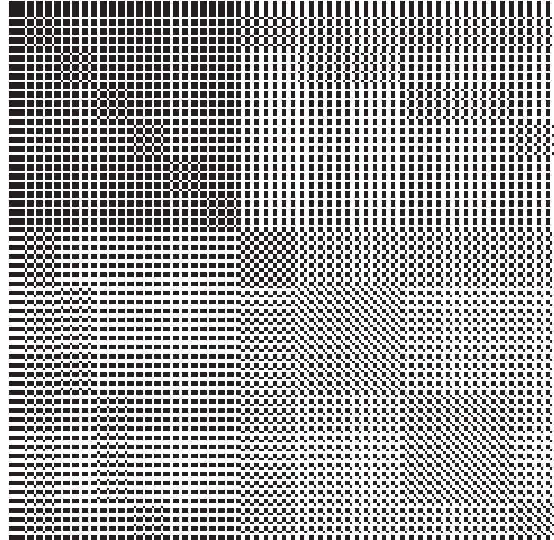
$(a+b+c+d+e+f+g)^2$

03 Jul 2006 H11: AlgebraicExpressions

Examples

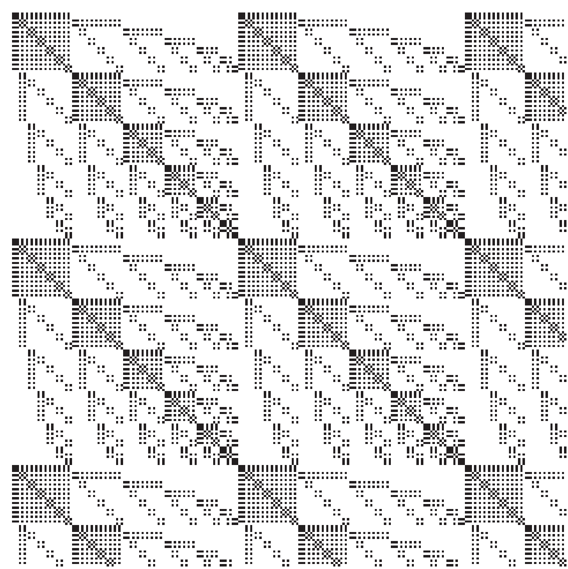


$(a+b+c+d+e+f+g)^3$

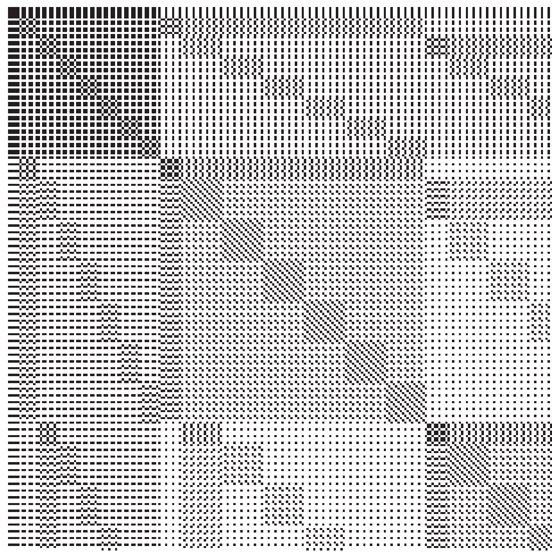


$(a+b+c+d+e+f+g)^4$

Examples

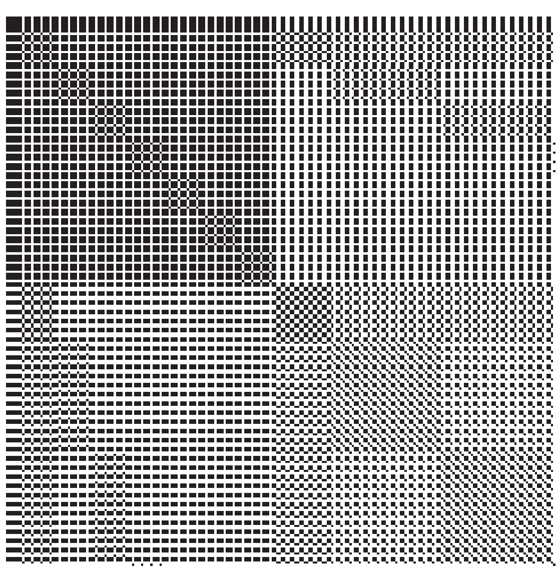


$(a + b + c + d + e + f + g + h)^2$

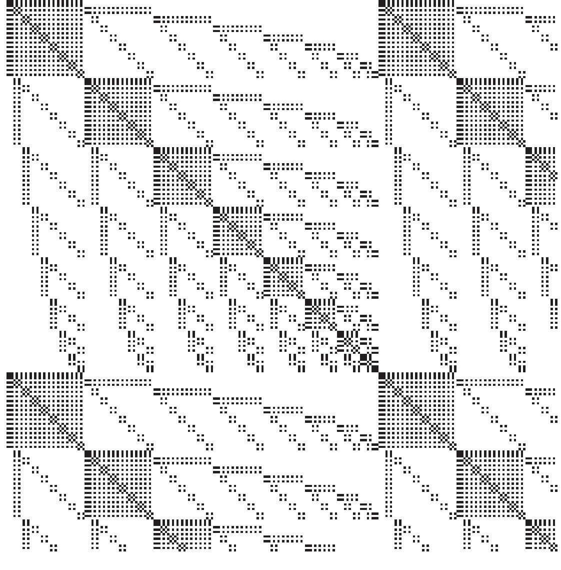


$(a + b + c + d + e + f + g + h)^3$

Examples

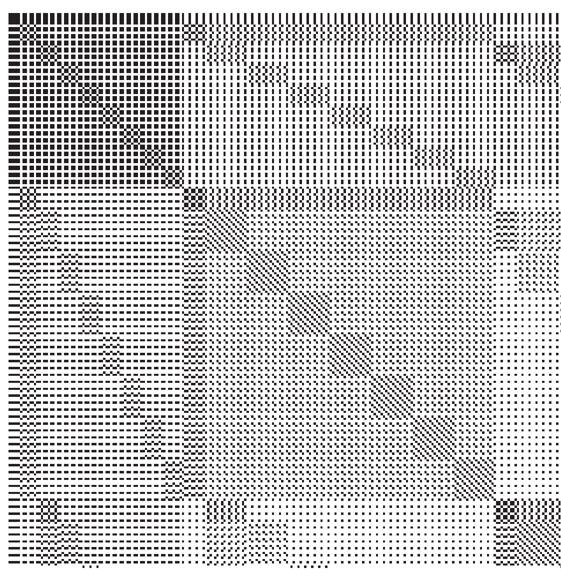


$(a + b + c + d + e + f + g + h)^4$

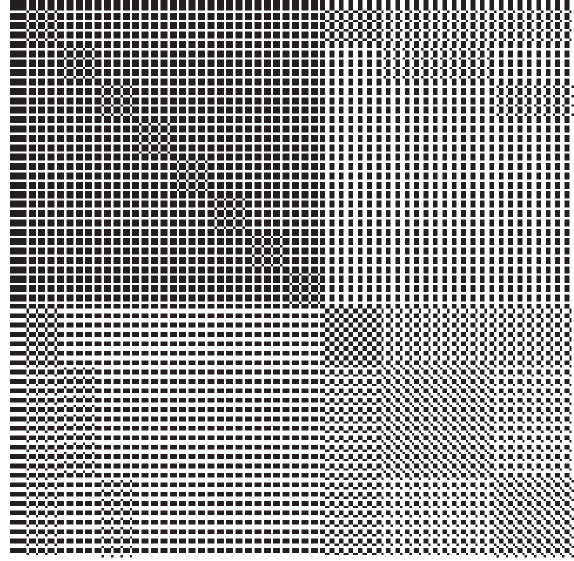


$(a + b + c + d + e + f + g + h + i)^2$

Examples



$(a+b+c+d+e+f+g+h+i)^3$



$(a+b+c+d+e+f+g+h+i)^4$

Meandering Sequences

Given j integers such as 1, 2, 3, …, j, a *j-k-meandering sequence*, or simply *j-k-meander*, contains all subsequences of j integers of length k [1]. Meandering sequences are considered to wrap around from end to beginning for the purpose of representing subsequences. [Note: This started out as meandering strings and has been converted to sequences. Some things need clarification, like the fact that the integer values can be anything.]

For example, the length-2 sequences composed from the 3 integers 1, 2, and 3 are {1, 1}, {1, 2}, {1,3}, {2, 1}, {2, 2}, {2, 3}, {3, 1}, {3, 2}, and {3, 3}. A 3-2-meander that contains all these sequences as subsequences is {1, 1, 2, 1, 3, 2, 2, 3, 3}. The subsequence {3, 1} comes from the last value of the meander followed by the first.

For j integers, it can be shown that:

- A j-k-meander must contain at least j^k values.
- There is a *minimal* j-k-meander that contains exactly j^k values.
- There is a straightforward way of constructing minimal j-k-meanders.

Meanders become long as j and k increase. Here is a table of some values:

j	*k*	*length*
2	2	4
2	3	8
2	4	16
2	5	32
3	2	9
3	3	27
3	4	81
3	5	243
4	2	16
4	3	64
4	4	256
4	5	1024
5	2	25
5	3	125
5	4	625
5	5	3125

Design Uses

Meanders can be used in design in a variety of ways. In this regard, the value of j determines the number of design objects, while, somewhat in the manner of Dietz polynomials, the value of k corresponds to the degree of interaction among the objects.

It doesn't matter what integers are used. They can be interpreted in a variety of ways.

One interpretation of meanders is as T-sequences for threading and treadling. Another is as blocks for profile drafting.

Another interpretation of meanders is as sequences of colors, which can be used, for example, for stripes. In this interpretation, the valuess in the meander are mapped into colors from a palette. For example, 1 might stand for peach, 2 for white, and 3 for sky blue.

Another possible interpretation of the values is as widths. For example, 3 might stand for 3 threads.

There are many other possibilities. For example, values in odd-numbered positions might stand for colors and values in even-numbered positions might stand for corresponding widths.

The possibilities are limited only by your ingenuity.

Some examples based on some of the ideas given above are shown on the last page of this article.

Constructing Meanders

The process of constructing minimal meanders is relatively simple, although for long ones it is tedious and error-prone if done by hand.

1. Put the values in some order. The order does not matter except as it affects the details of the result.

2. Start with a sequence of k–1 copies of the first value.

3. Append the last value to the current sequence, provided it does not produce a duplicate k-length subsequence (wrap-around doesn't apply here). If the last value would produce a duplicate, try the next-to-last, and so on, until one works. If no value works, go to Step 4. Otherwise repeat Step 3.

4. Remove the starting sequence. The result is a minimal j-k-meander.

As an example, suppose $j = 3$, $k = 2$, and the three values are 1, 2, and 3. Then the starting sequence is {1}.

The first application of Step 3 appends 3 to this result, giving {1, 3}. Repeating Step 3, the sequence becomes {1, 3, 3}.

So far, so good. However, in trying Step 3 again, appending 3 would produce {1, 3, 3, 3}. But now there are two (albeit overlapping) instances of {3, 3}, so this isn□’t allowed. Following the instructions, try the next-to-last value, ,2 which gives {1, 3, 3, 3}.

Now when Step 3 is done again, the last value, 3, can be appended to give {1, 3, 3, 2, 3}.

Neither 3 nor 2 can be appending, since these would produce duplicate subsequences. This leaves 1, giving{1, 3, 3, 2, 3, 1}.

Starting Step 3 over, 3 can’t be appended because there already is is an {1, 3}, but 3 can be appended, giving{1, 3, 3, 2, 3, 1, 2}.

Trying Step 3 once again, 3 can’t be appended but 2 can, giving {1, 3, 3, 2, 3, 1, 2, 2}.

At Step 3, 3 and 2 can’t be appended but 1 can, giving {1, 3, 3, 2, 3, 1, 2, 2, 1}.

And Step 3 another time only 1 can be appended, giving {1, 3, 3, 2, 3, 1, 2, 2, 1, 1}.

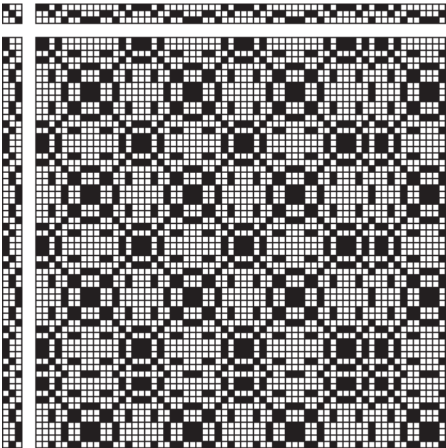
At Step 3 again, nothing works, since there already are subsequences {1, 3}, {1, 2}, and {1, 1}.

On to Step 4. All that remains is removing the initial sequence, 1, giving {1, 3, 3, 2, 3, 1, 2, 2, 1, 1}. That is the result.

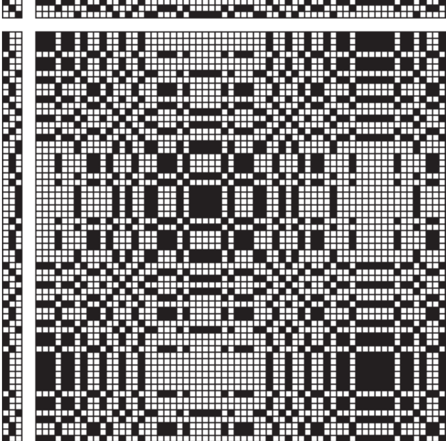
Note how the order in which the values are placed affects their order in the result.

All this detail seems tedious, but it practice it goes quickly, at least for small *j* and *k*. The usefulness of a program for larger values of *j* and k is clear, however.

Designs Based on Meandering Sequences



Profile Draft for Mirrored 3-2-Meander



Profile Draft for Mirrored 3-3-Meander

Designs Based on Meandering Sequences

Stripe Colors from 3-3-Meander

Stripe Colors from 3-4-Meander

Stripe Colors from 4-3-Meander

Friendly Sequences

A *friendly sequence* is one in which successive terms differ by one. Since a friendly sequence may be a repeat on which a longer sequence is based, the first and last terms must be friendly so that repeats are friendly.

Friendly sequences often make good candidates for threading and treadling sequences. And since they have alternating parity, they are applicable to weaves that have this requirement, such as overshot.

Close proximity amounts to friendship. Figure Ω.1 shows a friendly sequence, which labelled ☺. Notice, as required, its first and last terms are friendly.

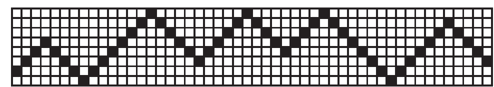


Figure Ω.1. ☺ A Friendly Sequence

Figure Ω.2 shows a fairly unfriendly sequence, labelled 😐, and Figure Ω.3 shows a downright hostile sequence, ☹.

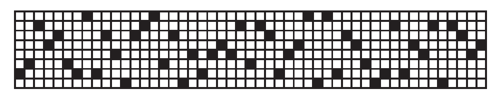


Figure Ω.2. 😐 A Fairly Unfriendly Sequence

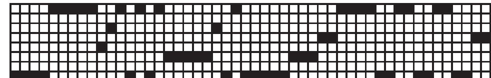


Figure Ω.3. ☹ A Hostile Sequence

☺ exudes good vibes; it's a cheerful sequence. The tension and confusion in 😐 are evident, while ☹ reeks of discord.

Our goal here is to convert unfriendly sequences to friendly ones — to befriend unfriendly sequences. These are the rules:

- Only friendly terms may be added.
- Terms may not be deleted.
- Existing friends may not be separated.

Under these rules, befriending a friendly sequence does not change it.

The most straightforward and conservative approach is to add the fewest terms necessary to achieve a friendly result. This involves inserting a friend between pairs of equal, self-focussed terms and adding a run of friendly terms between unfriendly terms that are some distance apart.

When there is a pair of equal, self-focussed terms, there is a question of whether to insert a term that is one larger or one smaller. This can be done many ways. One natural way is to make the choice at random. Another way is to alternate between the two choices. In the examples that follow, choices are made at random.

A more enthusiastic approach is to allow some leeway in inserting friends between unfriendly terms — letting the friendly path wander a little, adding more friends than are strictly necessary. Wandering implies some degree of randomness. Of course, friend-binding paths are expected to be finite so that the befriending process terminates. For this reason, the choice of direction is biased toward the target friend in a manner that makes the probability of termination very high.

Figure Ω.4 shows the results of befriending 😐 and ☹ in a conservative way. Figure Ω.5 shows the results for more enthusiastic befriending. Note that enthusiastic befriending produces a more lively result than conservative befriending.

Befriending as done here can add values larger or smaller than those in the original sequences.

What is a Friend?

The key question in befriending sequences is what constitutes a friend. What appeared to be a simple statement appears atthe beginning of this article, but sequences for drafting may come from modular reduction in order to bring a sequence within the bounds of the number of shafts or treadles to be used [1]. In such cases, the modulus and 1 are friends.

Modular reduction effectively wraps the sequence around a modular wheel whose modulus, m, is the number of shafts. Values not in the range $1 \le i \le m$ are replaced by their residues. See Figure Ω6.

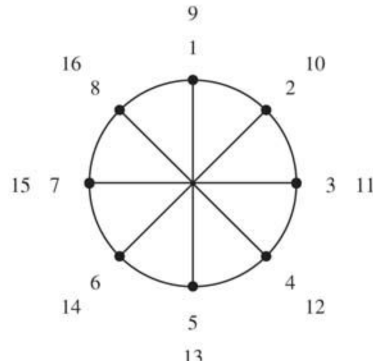


Figure Ω.6. Arithmetic Shaft Modulo 8

The converse operation to modular reduction, called modular expansion, can be used to convert a sequence on m shafts to a sequence on n shafts, $n \geq m$, in which there is no wrap-around. The result is a sequence whose residues, shaft modulo m, produce the original sequence.

Figures Ω.7 and Ω.8 show an example of modular expansion.

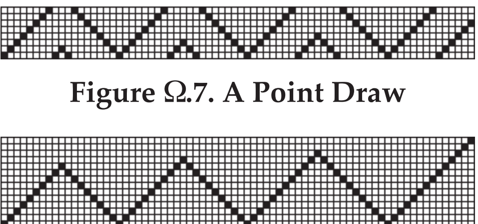


Figure Ω.7. A Point Draw

Figure Ω.≠8. The Modular Expanded Point Draw

Note that modular expansion exposes the underlying pattern in this point draw.

The process of modular expansion is simple and relies on the fact that 1 and m are adjacent on the modular wheel.

Starting with $i = 1$, if term $t_i = m$ and $t_{i+1} = 1$, add m to t_{i+1} and all the remaining terms (shifting them upward by m). Similarly, if $t_i = 1$ and $t_{i-1} = m$, subtract m from t_{i-1} and all the remaining terms (shifting them downward by m). Note that adding or subtracting a multiple of m does not affect the residues.

When the process is complete, add enough multiples of m to bring the smallest value in the range 1 to m. (The smallest value can be less than 1 but it cannot be greater than m, since t_1 is not greater than m and is not changed by the process.)

If the expanded sequence is not friendly, it can be made friendly and then

reduced according to the original modulus. Figure Ω.9 shows a sequence that is not friendly when it is expanded, as shown in Figure Ω.10. Figure Ω.11 shows the result of conservatively befriending this sequence and Figure Ω.12 shows the result of modular reduction of the sequence by its original modulus.

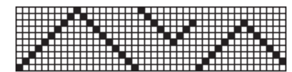


Figure Ω.9. A Sequence

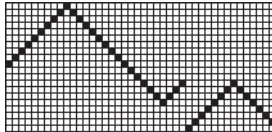


Figure Ω.10. The Modular-Expanded Sequence

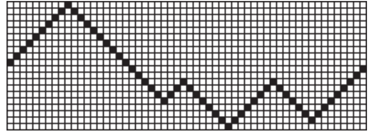


Figure Ω.11. The Befriended Modular-Expanded Sequence

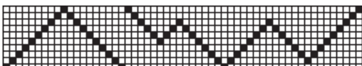


Figure Ω.12. The Modular-Reduced Befriended Sequence

Design Applications of Friendly Sequences

Virtually any sequence can be befriended to produce a threading or treadling sequence that gives an aesthetically pleasing weave.

There are many sequences that can benefit from befriending. For example, the modular reduction of sequences of mathematical origin often produces a periodic but unfriendly sequence. Figure Ω.13 shows the Fibonacci sequence shaft-modulo 8, which has period 12, and Figure Ω.14 shows the result of conservatively befriending it, which has period 42.

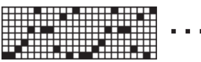
 . . .

Figure Ω.13. The Fibonacci Sequence Modulo 8

Figure Ω.14. The Befriended Fibonacci Sequence Modulo 8

Although these are very different sequences, the second is derived from the first in a well-defined way.

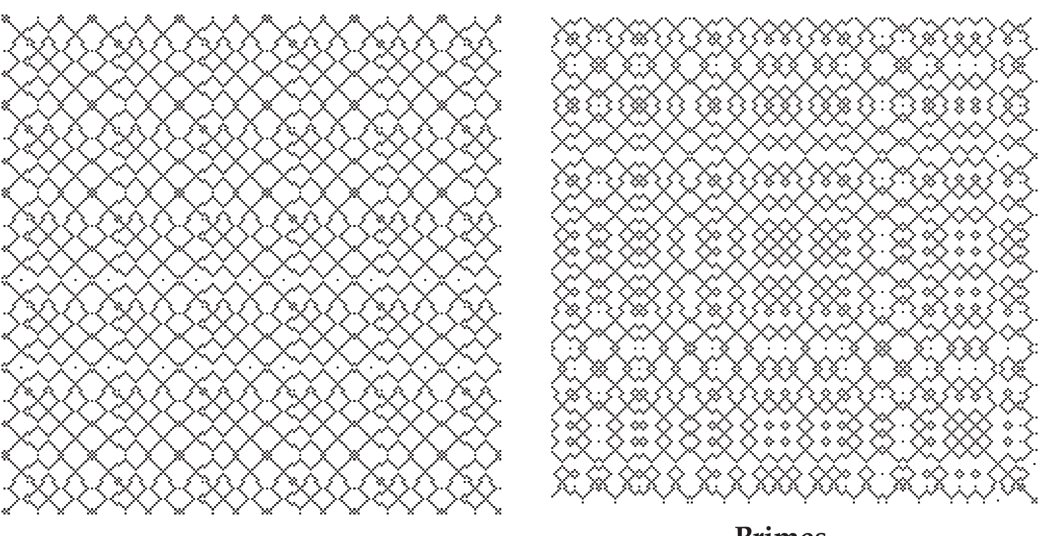
The Appendix shows some examples of weaves derived by befriending equences with mathematical origins.

There are, of course, endless other possibilities for using friendly sequences. That is the challenge of creative weave design.

Appendix

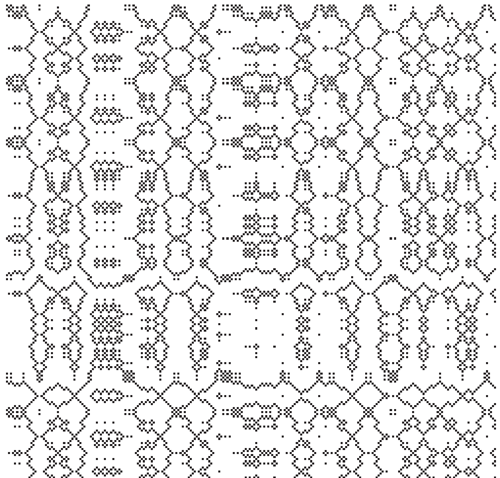
These drawdown images have 240 ends and 240 picks. All for eight shafts and eight treadles, treadled as drawn in. Befriendings are conservative except as noted

Direct Tie-Ups



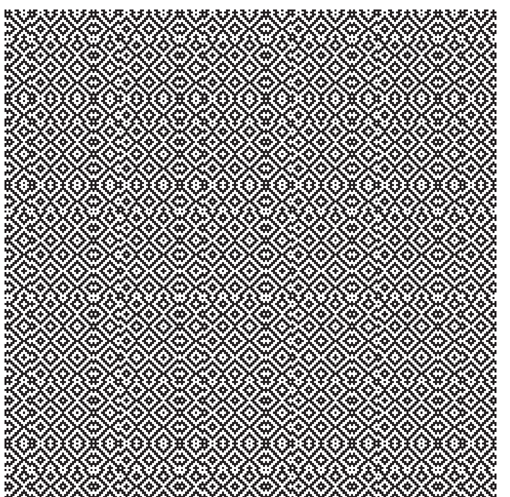
Primes

Fibonacci Sequence

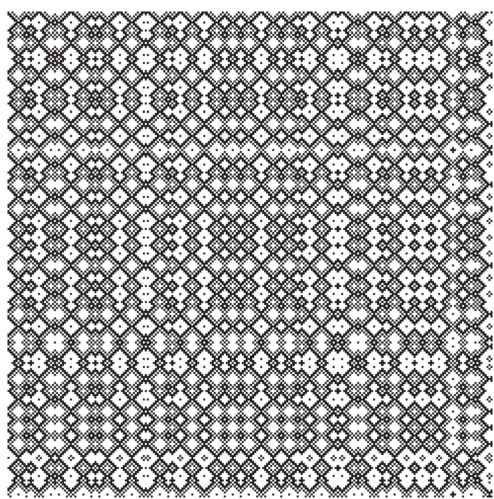


Fibonacci Sequence, Enthusiastic

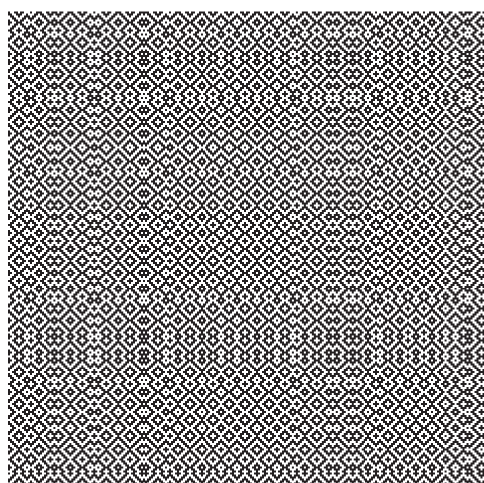
Twill Tie-Ups



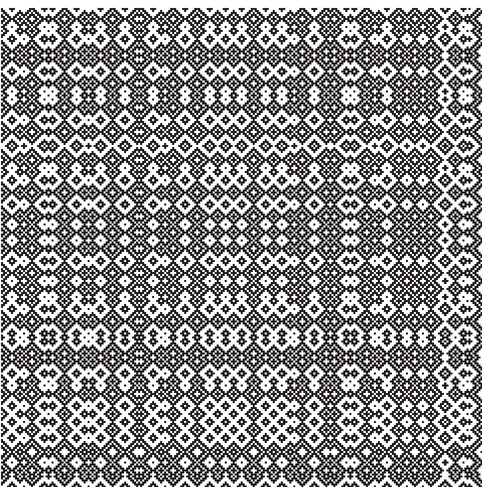
Fibonacci Sequence, 2/2



Primes, 2/4/1/1

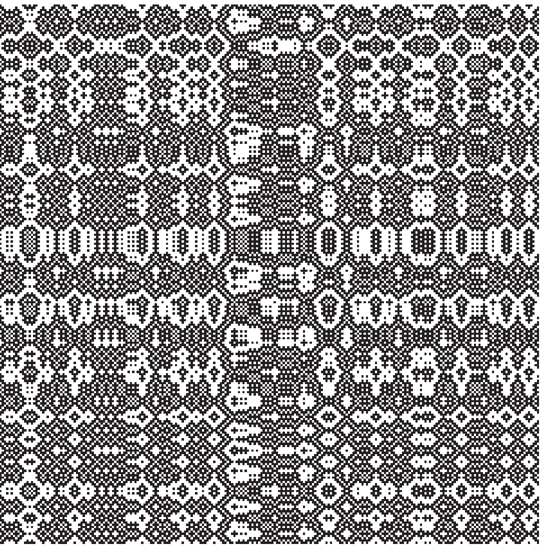


Primes, 2/2

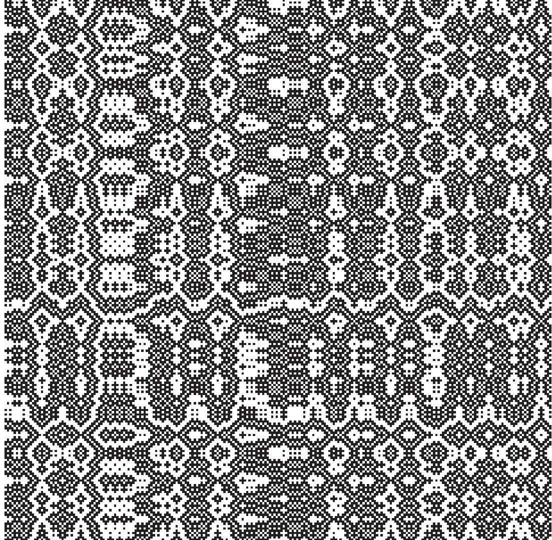


Primes, 2/1/2/3

Twill Tie-Ups



Primes, 2/1/2/3, Enthusiastic



Fibonacci Sequence, 2/1/2/3, Enthusiastic

Smarandache Sequences

All kinds of things can be found among integer sequences, including the weird and nonsensical. Enter Smarandache sequences (S. sequences, for short), which are integer sequences due to Florentin Smarandache and his disciples.

Some S. sequence are related to number-theoretic topics. Others, at first glance (second, third, …) seem downright silly.

An example is the progressive concatenation of the digits of the Fibonacci numbers:

1, 11, 112, 1123, 11235, …

Sequences like this one, based on digit manipulation, do not have any natural important mathematical properties, since they depend on base-10 representation of numbers and not on the properties of the numbers themselves.

On the other hand, weave design depends on patterns and not on the actual values of numbers. The numbering scheme used for shafts and treadles does not rely on any mathematical properties of the numbers.

With this said, it is worth exploring S. sequences to see what patterns emerge, starting with concatenation sequences and going on to other kinds of S. sequences in subsequent sections.

Florentin Smarandache was born in Romania in 1954. He describes himself as poet, playwright, novelist, writer of prose, tales for children, translator from many languages, experimental painter, philosopher, physicist, and mathematician.

He presently is associate professor of computer science at the University of New Mexico.

In 1980 he set up the "paradoxism" movement, which has many advocates in the world. It is based on an excessive use of antitheses, antinomies, contradictions, paradoxes in creation paradoxes — both at the small level and the entire level of the work — mathematics, philosophy, and literature.

Concatenation Sequences

Concatenation sequences have the property that terms are derived from other sequences by concatenation

Examples of concatenation sequences found in the literature [1] include the natural number repetition sequence

1, 22, 333, 4444, 55555, 666666, 7777777, 88888888, 999999999,
10101010101010101010, 1111111111111111111111, …

the prime concatenation sequence

2, 23, 235, 2357, 235711, 23571113, …

and the cube concatenation sequence

1, 18, 1827, 182764, 182764125, …

The terms in repetition sequences are given by the rule

$$t_i = s_i^i$$

where s_i is the ith term of the *base sequence* on which the repetition sequences in built, t_i is the ith term in the repetition sequence, and a^i denotes i copies of a.

Another possibility that produces the same results as the one above for the natural numbers is

$$t_i = s_i^{s_i}$$

That is, replicate the ith term of the base sequence by its value. For this rule, the repetition of the primes would be

22, 333, 55555, 7777777,…

instead of

2, 33, 555, 7777, …

by the first rule.

For concatenation sequences, the terms are given by

$$t_i = t_{i-1}s_i$$

There are endless possibilities for other rules of these general types, such as

$$t_i = s_i t_{i-1}$$

which for the primes produces

2, 32, 532, 7532, 117532, 13117532, …

And, of course, repetition and concatenation sequences can be used as base sequences, and so on, although in most cases the size of terms gets out of hand.

Yes, this is all silly — digit play, not mathematics. But can any interesting weave designs come from it?

From S. Sequences to T-Sequences

For t-sequences (threading and treading sequences) [2], values need to be limited to the number of shafts/treadles available. Direct conversion of an S. concatenated sequence to a t-sequence can be done by modular reduction [3].

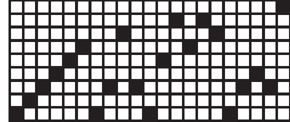
Since the terms in S. concatenated sequences get longer and longer (at least for the rules shown above), another alternative is to interpret a term as a sequence of digits. For example, the ninth term in the concatenated cube sequence is

182764125216343512729

Converting the digits to terms gives

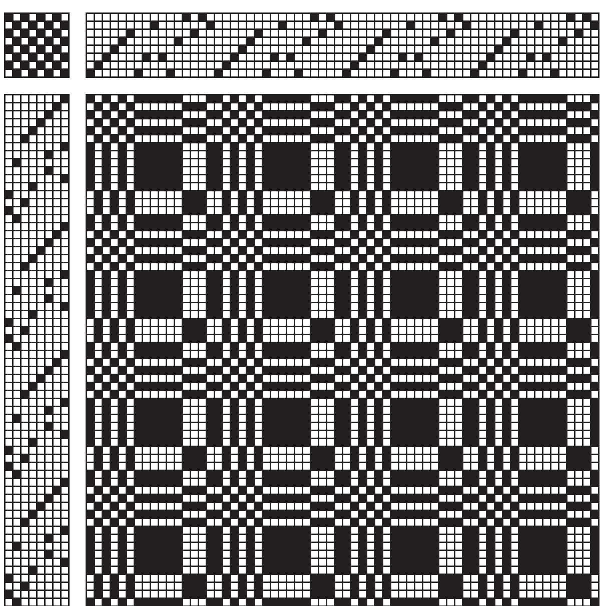
1, 8, 2, 7, 6, 4, 1, 2, 5, 2, 1, 6, 3, 4, 3, 5, 1,
2, 7, 2, 9

Normalizing this [4] and representing the result graphically produces:



With a tabby tie-up and treadled as drawn in the draft is

02 Aug 2006 H14: Smarandache



This interpretation of terms in an S. sequence can produce t-sequences with at most 10 values (0 can be converted to 10 arbitrarily, or one can be added to all values).

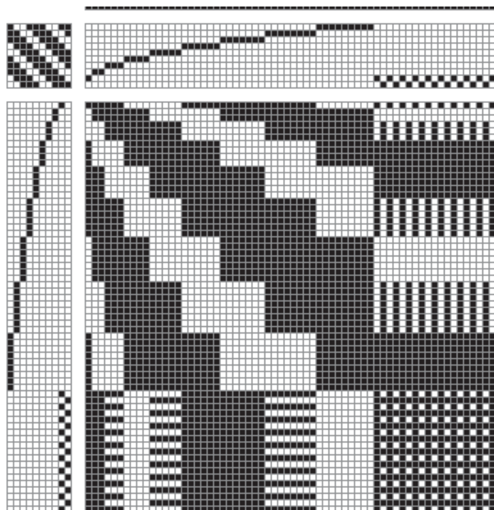
To get good results with this method, S. sequences and their terms need to be selected with care. Even then, some will need to be modified.

What about the more conventional approach of using modular reduction to bring large values into the domain of t-sequences?

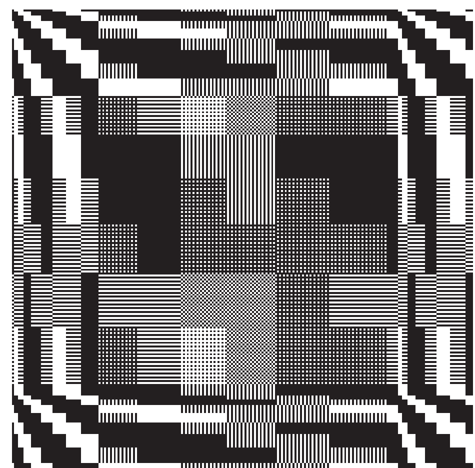
The natural number repetition sequence is about as unpromising a candidate for design as one might find among concatenation sequences. But modular reduction for 10 shafts produces this sequence:

2, 3, 3, 4, 4, 4, 5, 5, 5, 5, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 9, 9, 9, 9, 9, 9, 9, 9, 10, 10, 10, 10, 10, 10, 10, 10, 10, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 2, 2, 2, 2, 2, 2, …

Here is one possible draft:



and the weave pattern is



Conclusion

S. concatenation sequences are clearly artificial from a mathematical point of view, but they do produce patterns and among them there are possibilities for novel weave designs.

The next section will look at a more promising kind of S. sequence: S. palindromes.

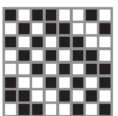
Sound Interlacements

There are two parts to conventional fabric analysis [1]:

1. Determining the interlacement of the warp and weft threads.
2. Producing a draft from this interlacement.

The two parts may be done separately or in combination, depending on the particular technique used. For the purposes of this section, it is convenient to view them separately with the first part producing a "drawdown" pattern.

[Redundant material.] Drawdowns can be represented in various ways. For visual understanding, a rectangular grid of cells, with each cell representing a point of interlacement, works best. In drawdown systems, black grid cells indicate where a warp thread is on top and white cells indicate where the weft threads are on top. Here is an example:



drawdown

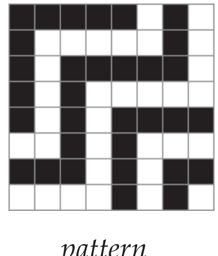
A drawdown that accurately represents the interlacement of a sound fabric can be used to "draw up" a draft for weaving the fabric.

This draw-up process can be used for producing drafts from patterns obtained from sources other than fabric analysis. Some weaving programs provide this capability. But there may be a problem.

A Problem

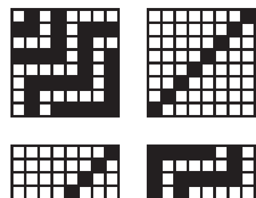
In their seminal paper on weave structures [2], Grünbaum and Shephard of tiling and pattern fame [3], pointed out that patterns that look perfectly reasonable may not produce interlacements that "hang together"; if woven, some interlaced warp and weft threads may not be interlaced with the rest of the fabric. A fabric woven based on such a pattern would come apart in pieces. Such an interlacement is *unsound*. *Note:* For some drafts, such as some kinds of double weave, this is the expectation, not a problem. But for patterns produced by the various methods described in this book, it is a problem.

Consider this pattern and the corresponding drawn-up draft:



pattern

tie-up *threading*

treadling  *drawdown*

draft

At first glance, the draft looks perfectly reasonable. If woven, however, the result would be pieces that would not hang together.

What distinguishes patterns whose corresponding interlacements are not sound from ones that are?

Interlacements with long floats, may, of course, not produce stable fabrics, but they hang together, however loosely. Patterns with complete rows or columns of cells of the same colors obviously have unsound interlacements. But there seems nothing obvious about the pattern and draft shown above that would indicate a problem.

Determining whether or not a pattern when drawn up represents an interlacement that hangs together — or doesn't — cannot be done by simple visual inspection. Instead, an algorithm (procedure) is needed. Grünbaum and Shephard's paper was followed by several papers giving algorithms for determining whether or not a fabric hangs together [4-6].

These algorithms are written in the language of mathematics and are not easy to follow, even for an educated layperson.

21 Jun 2006 I1: ProblemDrafts

A Mathematical Method

The following procedure is due to Clapham [4].

First note that the columns and rows of a drawdown can be rearranged without affecting whether or not the fabric hangs together.

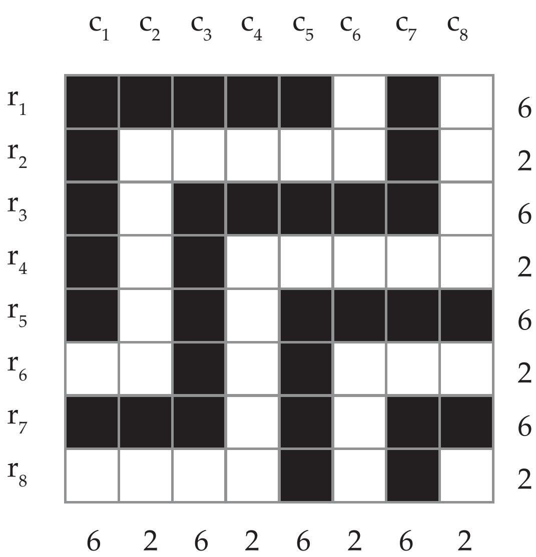
The basic idea is to rearrange the rows and columns so that the bulk of the black (warp) cells are at the top right and the bulk of the white (weft) cell are at the bottom left.

If the resulting pattern can be divided up in this fashion, with the bottom-right corner of the all-black area just touching the upper-right corner of the all-white area,

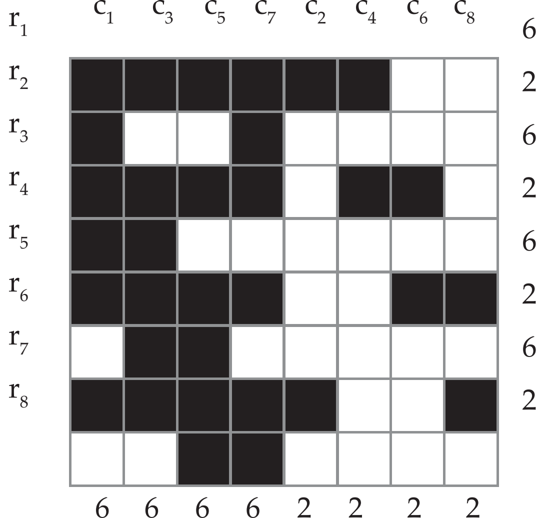
all black	*anything*
anything	*all white*

then the fabric would not hang together. The black and white areas need not be square; they can be any rectangular shape, all that is required is that their corners touch as shown.

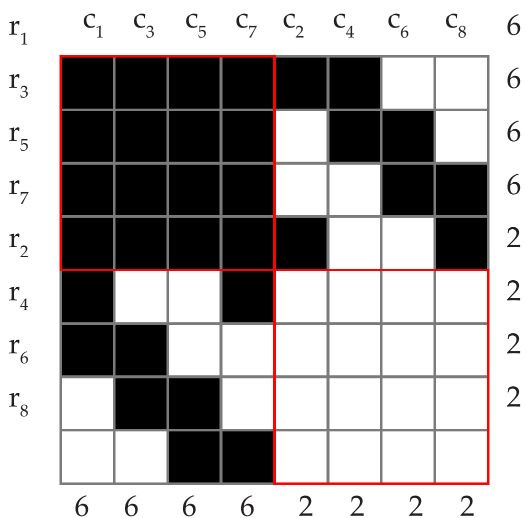
The method of rearranging the rows and columns is first to rank them according to the number of black cells they have. For the example above, this is



The next step is to rearrange the columns so that they are in order of their ranks. If more than one column has the same rank, as in this example, it does not matter in what order the columns with the same rank are placed.



The last step is to rearrange the rows so that they are in the order of their ranks:



The red lines show that the all black and all white blocks meet as required, so the fabric does not hang together. Had they not so met, the fabric would hang together.

An Alternative Method

An alternative method, described by Grünbaum and Shephard [2], is to lift off a thread, lift off what it lifts, and so on. If all threads are lifted off by this process, the fabric hangs together. If not, it doesn't.

This can be done methodically as follows.

First, if there is a solid-colored row or column, it represents a thread that is not interlaced at all and obviously the fabric doesn't hang together.

Otherwise, two lists are needed: one to keep track of threads that have been lifted and another to keep track of threads that have been lifted but not checked to see what they lift.

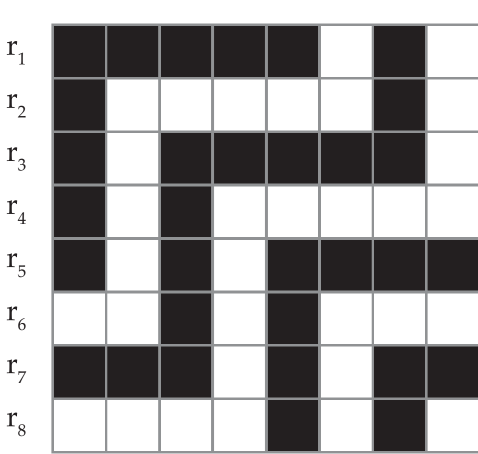
1. Start by putting the first column (warp thread) on the two lists. (Any other thread could be used to start.)
2. If the list to be checked is empty, go to Step 4. Otherwise pick a thread, remove it, and continue to Step 3 with this thread.
3. (a) If the thread is a warp thread, put all the weft threads that are on top of it (white cells in the column) but not already lifted on both lists.

   (b) If the thread is a weft thread, put all the warp threads that are on top of it (black cells in the row) but not already lifted on both lists.

   Go to Step 2.

4. If all warp and weft threads have been lifted, the fabric hangs together. If not, the lifted threads come off the rest and the fabric does not hang together.

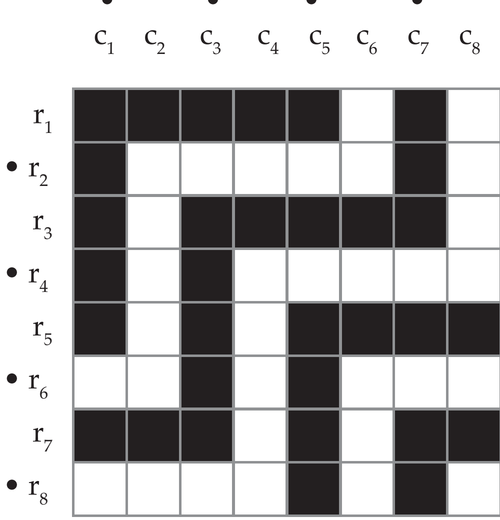
Here is how this procedure works on the previous example.



step	*thread*	*to check*	*lifted*
1	c_1	c_1	c_1
2	c_1		c_1
3	c_1	$r_6\, r_8$	$c_1\, r_6\, r_8$
2	r_6	r_8	$c_1\, r_6\, r_8$
3	r_6	$r_8\, c_3\, c_5$	$c_1\, c_3\, c_5\, r_6\, r_8$
2	r_8	$c_3\, c_5$	$c_1\, c_3\, c_5\, r_6\, r_8$
3	r_8	$c_3\, c_5\, c_7$	$c_1\, c_3\, c_5\, c_7\, r_6\, r_8$
2	c_3	$c_5\, c_7$	$c_1\, c_3\, c_5\, c_7\, r_6\, r_8$
3	c_3	$c_5\, c_7\, r_2$	$c_1\, c_3\, c_5\, c_7\, r_2\, r_6\, r_8$
2	c_5	$c_7\, r_2$	$c_1\, c_3\, c_5\, c_7\, r_2\, r_6\, r_8$
3	c_5	$c_7\, r_4$	$c_1\, c_3\, c_5\, c_7\, r_2\, r_4\, r_6\, r_8$
2	c_7	r_4	$c_1\, c_3\, c_5\, c_7\, r_2\, r_4\, r_6\, r_8$
3	c_7	r_4	$c_1\, c_3\, c_5\, c_7\, r_2\, r_4\, r_6\, r_8$
2	r_4		$c_1\, c_3\, c_5\, c_7\, r_2\, r_4\, r_6\, r_8$
3	r_4		$c_1\, c_3\, c_5\, c_7\, r_2\, r_4\, r_6\, r_8$
4			$c_1\, c_3\, c_5\, c_7\, r_2\, r_4\, r_6\, r_8$

21 Jun 2006 l1: ProblemDrafts

Since not all threads have been lifted, the fabric does not hang together. The pattern with the lifted threads marked is



Another Result

A theorem due to Clapham [7] is of interest:

If every weft thread passes both over and under more than one quarter of the warp threads, and if every warp thread passes both over and under more than one quarter of the weft threads, then the fabric hangs together. [Gregg: I find it hard to parse also. This is verbatim. Suggest complete rewording if you wish.]

The converse is not true; there are many fabrics, notably satins, the hang together but do not meet the requirements of this theorem.

Here are the required over/under numbers to assure a fabric hangs together for some warp/weft threads counts:

threads	*required*
4	2
5	3
6	3
7	3
8	3
9	4

10	4
11	4
12	4
13	5
14	5
15	5
16	5

Creating Interlacements that Don't Hang Together

Note that it is easy to create unsound interlacements. Using the ideas in the mathematical method, just create a pattern with all-white and all-black blocks that meet as required, fill in the other two blocks in any fashion, as long as they are not all black or all white, and rearrange the rows and columns to hide the problem. Why you would want to do this, except to create examples like the one here, is another question.

Dealing with Problem Patterns

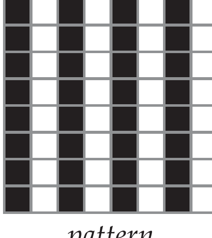
The difficulty with problem patterns is that it is not possible to produce a sound interlacement, one that "hangs together" and at the same time produces the pattern, using only black warp threads and white weft threads.

In many cases, a sound interlacement can be found if the warp and weft threads are allowed to be both black and white. This is familiar from color-and-weave effects [2], although the motivation here is different.

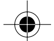
Be aware that there are patterns for which no sound drafts exist. Here's one:

But in most cases, there are sound interlacements for problem patterns.

An extreme example, stripes, illustrates how such interlacements can be found:

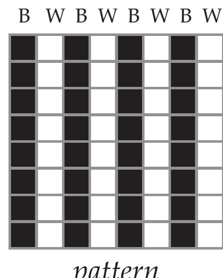


pattern

21 Jun 2006  I1: ProblemDrafts

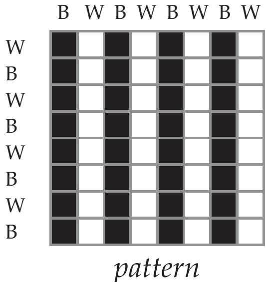
If this pattern is drawn up in the standard fashion, its interlacement obviously would not hang together: There is nothing to hold down the warp threads.

In this example, the solution is obvious: Select colors for the warp threads that match the colors of the stripes:



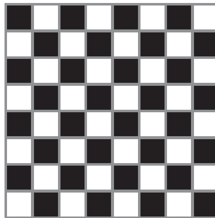
pattern

The weft thread colors can be chosen in a variety of ways, keeping in mind that they need to interlace with the warp threads. One way is



pattern

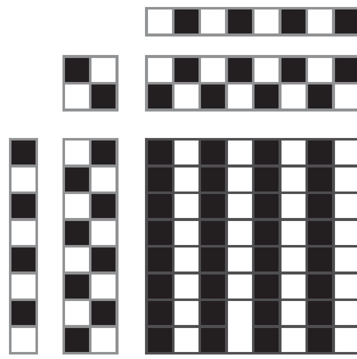
The next step is to decide on the interlacement. Plain-weave interlacement clearly works:



interlacement

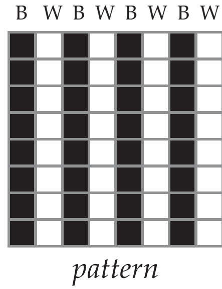
The draft, with the thread colors shown in bars at the top and left, is

21 Jun 2006 I1: ProblemDrafts



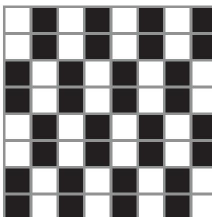
draft

Although the colors chosen above for the weft obviously work, there are other choices that do also. One is



pattern

An interlacement is



interlacement

This version has floats of length two.

This example illustrates the aspects of the general problem:

1. The first step is to find warp and weft colors that “satisfy” the pattern. (If it is a problem pattern, by definition all-black warp and all-white weft will not work.) There may be more than one choice.

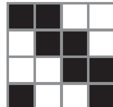
2. The next step is to produce an interlacement. Again, there may be more than one choice.

21 Jun 2006 I1: ProblemDrafts

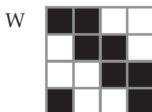
3. Finally, the interlacement must be tested for soundness. If it is not sound, it is necessary to retreat to Step 2 or even Step 1.

There is no simple, general method of solving the problem. A naive approach would be to try all possible warp and weft colors and all possible subsequent interlacements. For all but trivially small patterns, this will not work: There simply are too many possibilities.

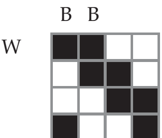
Consider this simple twill:



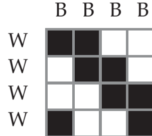
To start, a color needs to be assigned to some thread. Suppose the first weft thread is white:



This choice has immediate implications. To satisfy the pattern, the first and second warp threads must be black: These colors are *forced*:



The black warp threads in turn force the rest of the weft threads to be white, and these force the rest of the warp threads to be black:



The important point is that the choice of a color for one thread forces the colors for all the rest of the threads and produces a standard coloring from which a sound standard draft can be made.

If the first weft thread had been assigned black, the result would have been all weft threads black and all warp threads white: a complementary coloring. Any other choice for any thread would have produced one of these two; there are no others. The draft for this pattern is unique up to complementation.

This is not true of all patterns; some can be assigned thread colors in many ways. If this is the case for a problem pattern, there may be a sound draft for it. (By definition, the standard draft for a problem pattern is not sound.)

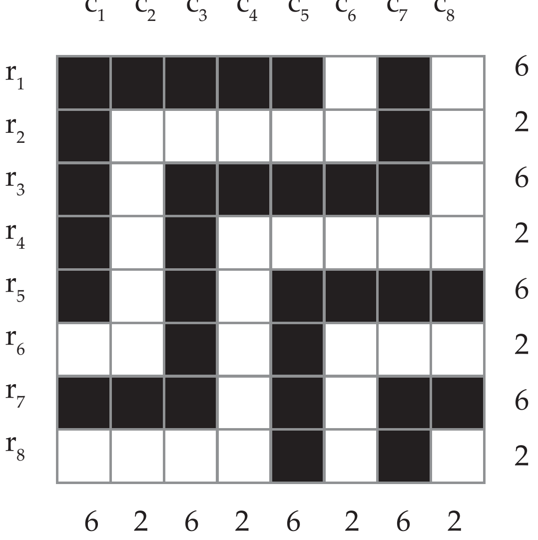
21 Jun 2006 I1: ProblemDrafts

The way to find thread colorings for problem patterns is the same as for the twill: Pick a color for one thread and see what colors are forced as a result. If they are all forced, there is not a sound draft. But if all the colors are not forced, there may be a sound interlacement, and the process continues by picking a color for some thread that is not already assigned a color.

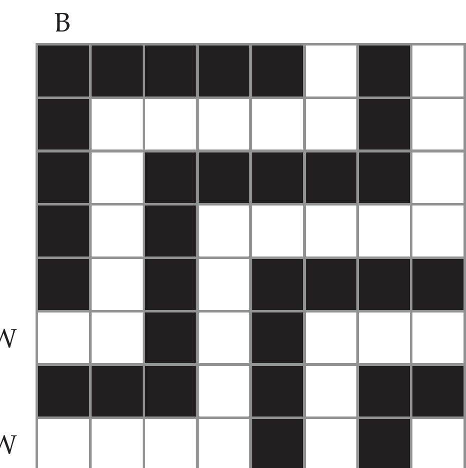
Although any thread and color could be used to start, it is reasonable to start with a "promising" thread and color. For example if one row or column is predominantly one color, it is reasonable to pick that thread and color accordingly.

The procedure can be made systematic by ranking rows and columns by the number of black cells they have.

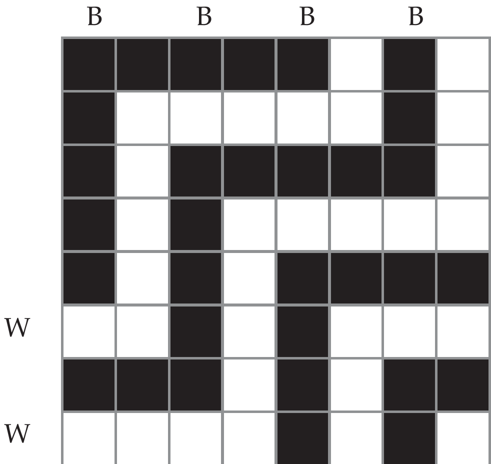
The first example in this section illustrates the process:



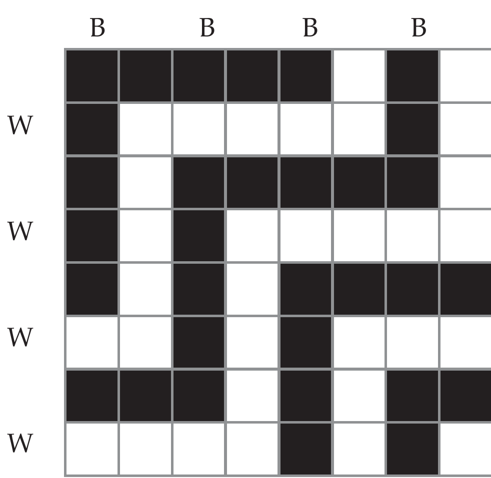
Starting with one of the highest ranking columns, the first, and assigning black to its warp color, the result after forcing for this column is



The forced weft threads in turn force three warp threads:

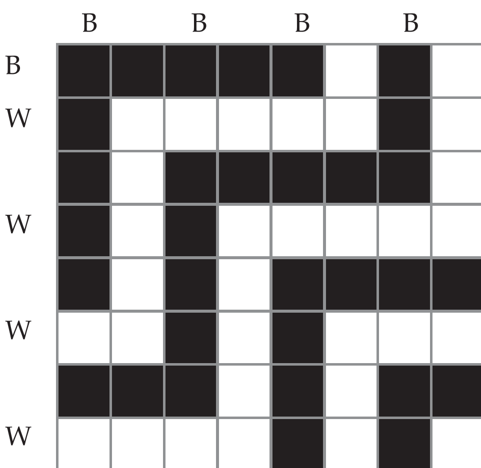


And these force two more weft threads:



At this point, all the forcing from the initial choice has been done and eight threads remain to be assigned colors.

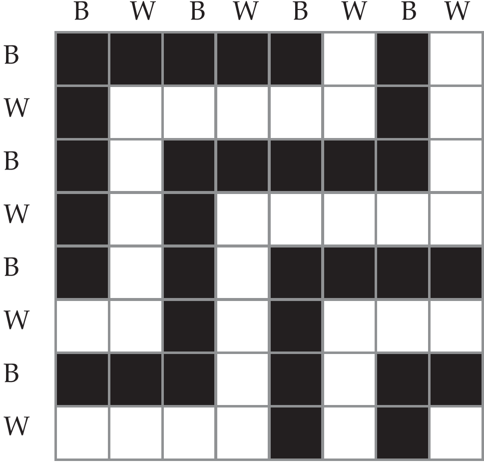
The first row is predominantly black, so the next choice is black for the first weft thread:



This results in forcing the remaining colors:

The next step is the interlacement. There are two considerations. The primary one is getting a sound interlacement. The secondary one is getting an interlacement with acceptable float lengths.

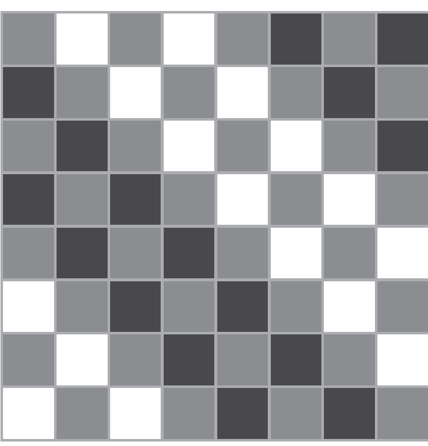
The colors for the example pattern in the last article are:



pattern

Two cases arise in determining an interlacement. If the warp and weft colors are different there is no choice: The thread that is on top is forced by the color of the corresponding cell in the pattern. The second case is where the warp and weft threads are the same color and either could be on top. This is an option cell.

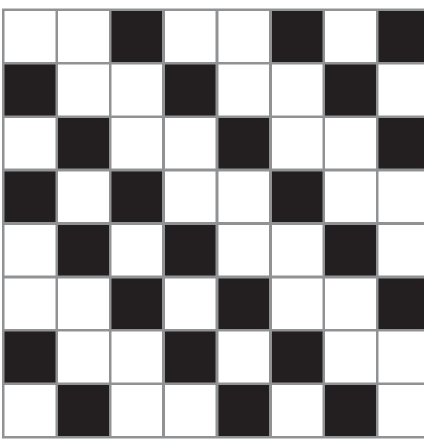
The second case is, of course, the important one. The situation for the pattern above is illustrated by a partial interlacement diagram in which the cells at option paints are gray:



partial interlacement

The large number of option points and their placement is a strong indication that there is a sound interlacement without long floats. Note the twill effect; this suggests the pattern is a color-alternate twill that does not hang together [2].

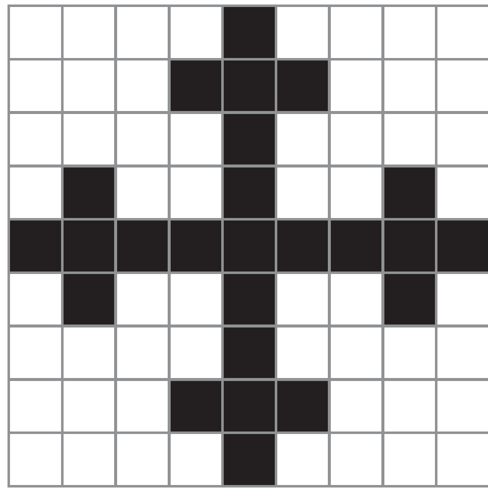
In this example, it is easy to get a sound interlacement. The idea is to make choices that prevent long floats. One solution is



interlacement

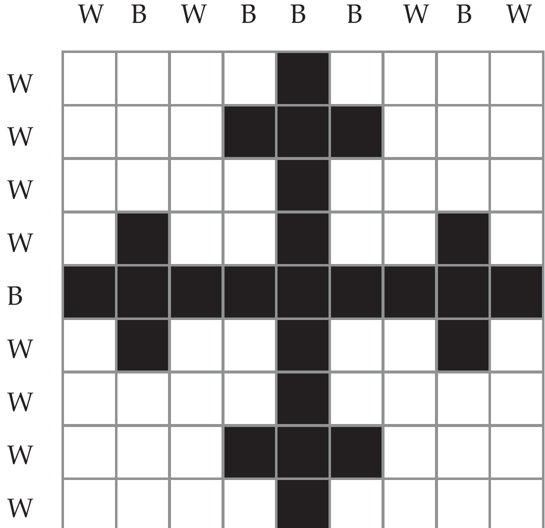
This is a twill with floats of length two.

Not all problem patterns have such good interlacements. Here's an example from a series of block-substitution fractals [3]:

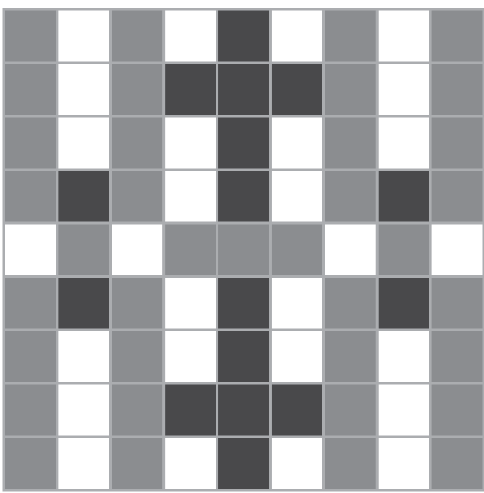


pattern

One possible thread coloring is



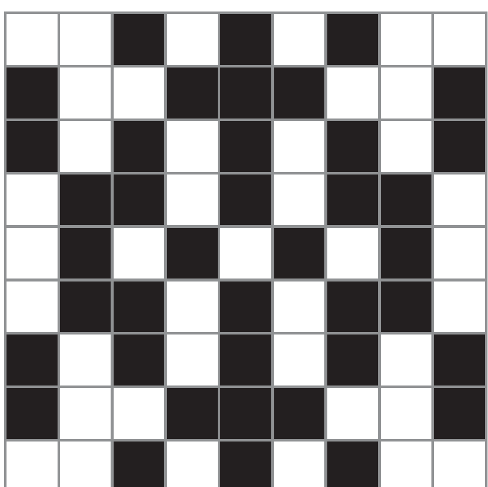
A partial interlacement is



partial interlacement

Note that whatever choices are made at option points, there will be a float of length eight if this interlacement is repeated vertically.

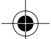
Here is one sound interlacement; at least it hangs together:



Of course, there are other possible colorings and interlacements, but the intersection of a solid-colored row and a solid-colored column suggests that not much better can be done.

These interlacements were done by hand. Done that way, the process is tedious and error-prone. To be able to get sound interlacements for many large patterns requires a method that can be incorporated in a program. Unfortu-

nately, such a program is far from trivial.



Draftable Color Patterns

Not being a mathematician, I am not obligated to complicate my explanations by excessive mathematical rigor.

— Petr Beckmann, *The History of Pi*

Suppose you are looking for a design for dish towels to weave on your floor loom. A little searching produces a design in the style of Mondrian; see Figure Ω.1. But when you try to set up your loom, you can't figure out how to assign color threads for the warp and weft to get the pattern you want. A little more work convinces you it is impossible — there are parts where there really is no way to assign thread colors to get the desired result.

So you try replacing yellow by white, as in Figure Ω.2. And you get the same result, although are only four colors. Pushing on, you replace blue by white, as in Figure Ω.3. You still can't find a way to make a draft. At this point, you begin to wonder if you're doing something wrong. Finally, you replace red by white. and, of course, a draft is easy — all-black warp and all-white weft will do it.

What's going on. Did you make mistakes in trying to assign thread colors in the other cases? Certainly, there are many weaves with lots of colors. How to they differ from what you're trying?

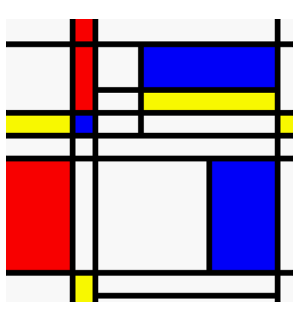


Figure Ω.1 Mondrian Design

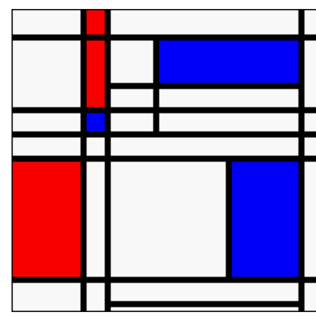


Figure Ω.2 Yellow Deleted

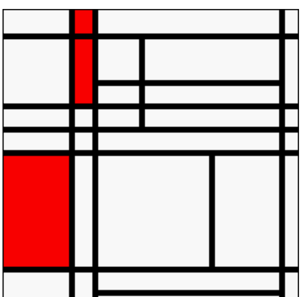


Figure Ω.3 Blue Deleted

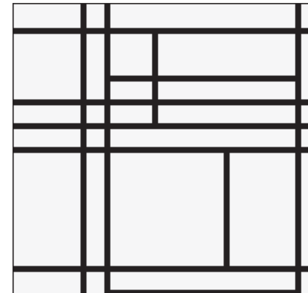
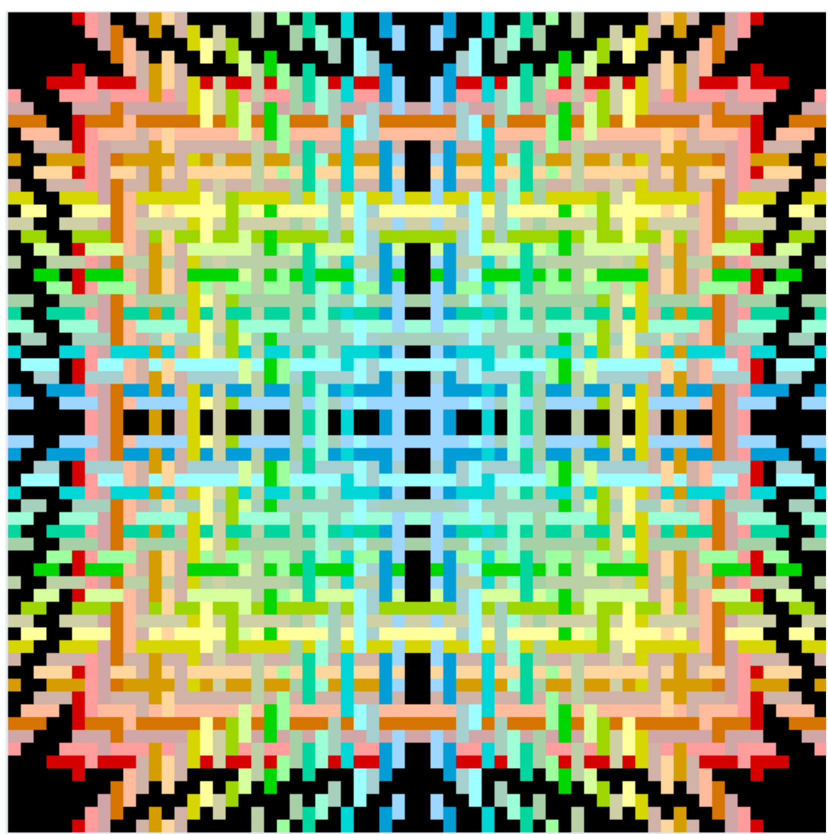


Figure Ω.4 Only Black and White

To answer the last question first, the difference between what you tried to do and most colored weaves is that the latter are designed by assigning thread colors to an existing draft; of course they are draftable. Here's a draftable color pattern, obtained by design:



In the context of loom-controlled weaving (as opposed to, say, tapestry weaving) there are many patterns that can't be woven. In fact, most can't. The problem is determining if a color pattern can be woven as the perpendicular interlacement of two sets of parallel threads.

That is one part of the problem, and generally the hardest. The other part of the problem is producing a draft, if the thread colors can be assigned.

The Color Problem

A rectangular color pattern can be considered as a grid of colored cells. If the color pattern is an image, the cells might be single pixels.

In a loom-controlled weaving, every cell in the grid corresponds to a point

of interlacement between a vertical (warp) thread and horizontal (weft) thread. Therefore, either the warp thread or the weft thread must be the color of the cell.

To simplify the description that follows, letters will be used to stand for colors. See the end of this section for equivalent color patterns. Columns correspond to warp threads and rows correspond to weft threads. In order for the grid to be draftable, the columns and rows must be labeled in a way that the label for every square is its column label or its row label — "satisfied". Figure Ω.5 shows an example grid.

Should colors be used here instead of letters? See the images on the last pages.

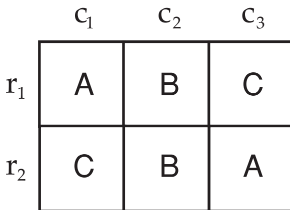


Figure Ω.5. A Labeled Grid

Column and row labels are assigned as follows. Starting with square (c_1, r_1), either c_1 or r_1 must be A. Arbitrarily pick c_1 to be A. This forces r_2 to be C. Figure Ω.6 shows the labeling to this point.

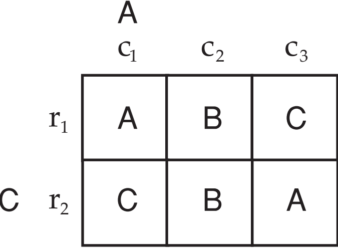


Figure Ω.6. First Labeling Step

<u>Since (c_3, r_2) is A and r_2 is C, c_3 must be A</u>. This requires r_1 to be C and hence c_2 must be B. Figure Ω.7 shows the final labeling.

		A	B	A
		c_1	c_2	c_3
C	r_1	A	B	C
C	r_2	C	B	A

Figure Ω.7. The Final Labeling

So far, so good. But what about the grid shown in Figure Ω.8?

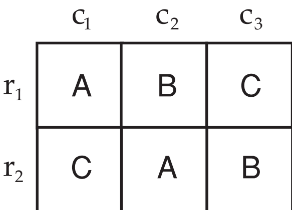


Figure Ω.8. Another Grid

Starting as before, assign A to c_1. This forces r_2 to be C, which in turn forces c_2 and c_3 to be A and B, respectively, as shown in Figure Ω.9.

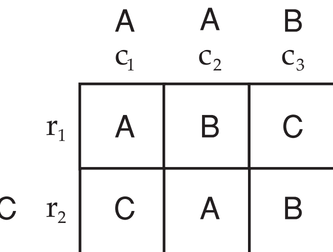


Figure Ω.9. First Step in Labeling

Now there is no way to proceed: r_1 cannot be both B and C. Starting anywhere else and trying any other combination of labelings leads to the same result. It's not possible to satisfy the grid: The pattern cannot be woven.

Note that if a larger pattern contains such a subpattern, the larger pattern cannot be woven either. Furthermore, the rows and columns do not have to be adjacent. The pattern shown in Figure Ω.10 is equivalent to the pattern shown in Figure Ω.9 as far as draftability is concerned.

	c_1		c_2			c_3
r_1	A	?	B	?	?	C
	?	?	?	?	?	?
	?	?	?	?	?	?
r_2	C	?	A	?	?	B

Figure Ω.10. Separated Rows and Columns

25 Jul 2006 I2: ColorDraftability

This is a very small pattern by weaving standards and it has only three colors. What then of more colors and larger patterns?

The Number of Colors

Suppose a pattern has k colors. For $k = 2$, all patterns can be drafted — simply assign one color to all columns (warp threads) and the other color to all rows (weft threads) and pick one or the other depending on the color at every intersection.

As illustrated by the example on the previous page, for $k = 3$, there are some patterns that cannot be drafted. For larger k there is a more fundamental problem. If a pattern has m columns and n rows, there are only $m + n$ colors available. If k is greater than $m + n$, then the pattern can't be drafted at all. Thus, there are 2×3 patterns that can't be woven for this reason. See Figure Ω.11.

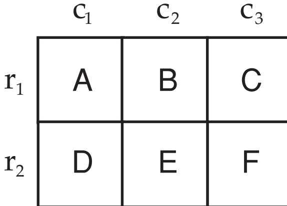


Figure Ω.11. A Pattern with Too Many Colors

For what follows, examples are limited to $k \leq m + n$.

Approaches to Solving the Problem

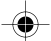
There are several possible ways the problem might be solved.

One way would be to try assigning the color of every cell to the columns and rows in all possible ways. This clearly is hopelessly time consuming except for tiny patterns.

Two colleagues produced viable methods and wrote programs to determine if a color image is draftable. One method recognizes the problem as an instance of the 2-satisifiabilty (2SAT) problem, for which there is a known algorithm [1]. The other solution is heuristic in nature.

The heuristic solution is described here for several reasons:

- It's original.
- It's interesting.
- It's fast for most patterns.
- It illustrates an approach that is worth considering for other problems.



The Heuristic Solution

Heuristics

A word about heuristics is in order, since they often are misunderstood. Heuristics use insights into the nature of a problem and intelligent guesses to build a solution method tailored to the problem.

Using heuristics doesn't mean wild guessing or proceeding blindly, just hoping to find a solution. Nor need a heuristic solution give incorrect answers, although proving a heuristic method is correct and terminates — and hence is an algorithm — may be difficult.

Heuristics can be used in many ways. For the problem here, one possibility would be look for a fast way to reject a pattern because it contains an unsolvable subpattern (such as the ones shown earlier). Of course, the absence of a known undraftable subpattern does not prove the whole pattern is draftable — so that problem would still exist.

Checking for special cases such as this one often takes more time on average than it saves. Since it's difficult — even impractical — to analyze the effects of such heuristics without implementing them and doing performance testing, such heuristics should be viewed with skepticism.

A good heuristic method relies on understanding the nature of the problem and, if possible, breaking the problem down into smaller, more tractable, subproblems.

Insights into Color Draftability

For the color draftability problem, the following observations are particularly useful.

- If a row or column is all one color, that color can be assigned to the corresponding row or column without affecting the rest of the problem. Hence such rows and columns can be eliminated from further consideration.
- Duplicate rows and columns can be eliminated for the same reason.
- The pattern can be rotated without changing the problem; in this sense, there is no difference between rows and columns.
- Rows can be interchanged (rearranged) without changing the problem, and the same is true of columns.

To get ideas for the heuristic approach to the problem, small subpatterns can be examined to see what implications they have for a pattern as a whole.

Two-colored patterns aren't particularly interesting, since all can be satisfied.

There are only two distinct three-colored 2×2 patterns; all others are equivalent to these by rotation or row and column interchange. See Figures Ω.12 and Ω.13.

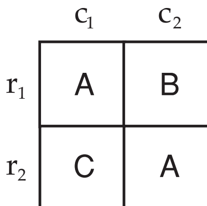


Figure Ω.12. Three-Colored 2×2 Pattern One

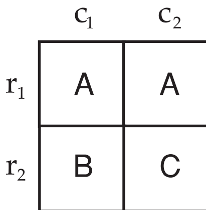


Figure Ω.13. Three-Colored 2×2 Pattern Two

The pattern in Figure 8 imposes some constraints on any larger pattern in which it is embedded: c_1 must be A or C, c_2 must be B or A, and similarly for the two rows.

For the pattern in Figure Ω.13, however, c_1, c_2, and r_2 are not constrained but r_1 is completely determined. It must be A for the entire pattern in which this subpattern is embedded.

This particular subpattern turns out to provide a sufficient basis for a heuristic solution; no others need be considered. This AA/BC pattern is called the *forcing pattern*.

A Program

What follows is a sketch of a program that implements the heuristic solution. The complete program is available on the Web [2].

Data Structures

The representation used for the pattern data is crucial. The main data structure is a vector that is used for both rows and columns.

A vector has several components, including:

an index of the row or column

a label differentiating rows and columns

a list of colors in cells

an identification of being "active" or not

An active vector is one still to be assigned a color. All vectors are active initially.

Program Structure

The program starts by reading an image file for the pattern and initializing data.

Next, duplicate rows and columns, as well as solid-colored vectors, are marked inactive. This may reduce the problem size significantly, especially for patterns with symmetries.

The main loop in the program then iterates over the pattern, developing constraints and setting colors determined by forcing patterns.

If at any time the pattern can be completely solved by simple means (see below), the problem is solved. Otherwise, all 2×2 subpatterns are examined for instances of the forcing pattern.

If a forcing pattern is found, the colors it forces are set and the loop continues. Since the cells of a forcing pattern need not be adjacent, all possible combinations of rows and columns are examined for forcing patterns.

When there are no more instances of the forcing pattern, an attempt is made solve the pattern by simple means. If this succeeds, the pattern is solved. If it fails, the pattern cannot be solved.

A pattern has a simple solution if one of the following applies:

1. The pattern is $1 \times n$ (or, equivalently, $n \times 1$) or 2×2, for which there are obvious solutions. See Figures Ω.14 and Ω.15.

	A	D
B	A	B
C	C	D

Figure Ω.14. A 2×2 Pattern

25 Jul 2006 I2: ColorDraftability

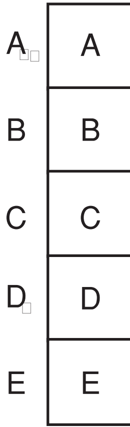


Figure Ω.15. A 1 × 5 Pattern

2. The pattern is solid colored except for a diagonal or part of one. Again, a solution is simple. See Figure Ω.16 for an example.

	B	C	D	A
A	B	A	A	A
A	A	C	A	A
A	A	A	D	A
A	A	A	A	E

Figure Ω.16. A 4 × 4 Diagonal Pattern

3. It can be solved by setting the color of a vector to one possibility, chosen arbitrarily, then setting colors of other vectors this forces, and continuing until all vectors have been assigned colors.

Output

On completion, the program indicates whether or not the pattern could be solved and produces lists of the row and column colors. An enlarged version of the pattern then is displayed in a window with row and column color assignments along the top, bottom, and sides. If the pattern could not be solved, the colors just reflect the program state at termination. Figure Ω.17 shows a solved color pattern.

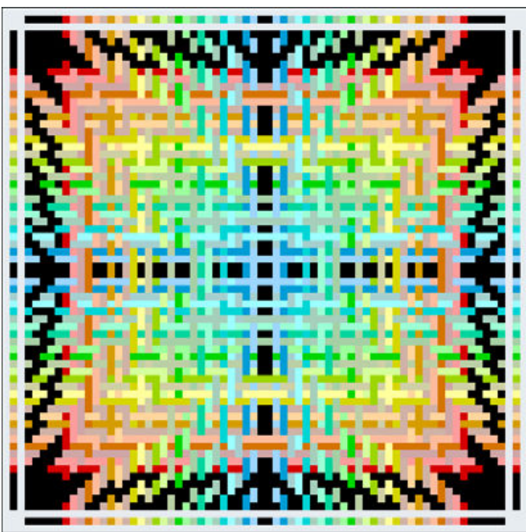


Figure Ω.17. A Solved Pattern

Sketch of a Proof

A formal proof that the method described above is correct and terminates with a result — and hence is an algorithm — would require a formalism and a lengthy and not particularly illuminating argument.

In the spirit of the quotation at the beginning of this article, here's just a sketch of a proof that, hopefully, will provide some insight.

Consider what remains after eliminating duplicate rows and columns, solid-colored rows and columns, and applying all the forcing patterns.

If the remaining pattern is $1\times n$, 2×2, or diagonal, the solution is trivial as shown earlier. Otherwise there is a 3×2 (or, equivalently, 2×3) or larger pattern containing no AA/BC forcing pattern.

If there are no rows or columns with duplicate colors, then the pattern is insoluble: a 3×2 color pattern with no duplicate color in one row or column is insoluble, as is every larger pattern of which it is a part.

The other possibility is that there is a AA/AB pattern There also may be AA/BB patterns, but there must be at least one AA/AB pattern, for otherwise the AA/BB pattern would identify two identical rows and columns, but they were eliminated earlier.

Given an AA/AB subpattern, there are only two possibilities that lead to a solution: The pattern has only two colors or it is a diagonal pattern.

This exhausts the possible patterns. Every original pattern eventually reduces to one of these cases, so the procedure terminates with a definite answer.

The Drafting Problem

What remains is to use the results of a solution to create a draft — threading and treadling sequences and a tie-up.

From the color assignments for columns and rows a drawdown can be obtained by looking at the color of each point of intersection. Then from this, a draft can be obtained.

For every cell in the pattern, there are three possibilities for a drawdown:

1. The corresponding row and column colors are the same, in which case either the warp or weft thread can be on top.
2. The column color is the same as the color of the point, in which case the warp thread is on top.
3. The row color is the same as the color of the point, in which case the weft thread is on top.

The first case, an option point, presents a problem — how to choose? The choice potentially is important, because it can affect the length of floats and the loom resources required.

For many patterns that might be candidates for weaving, the number of option points is huge. For the pattern shown in Figure Ω.17, 256 of the 4,096 points are option points. So there are 2^{256} possible drafts.

It's clearly hopeless to explore even a small fraction of possible drafts that result from making different decisions at option points.

The program that creates a draft [3] provides four ways of handling option points:

choose the warp or weft at random
always chose the warp
always chose the weft
chose the warp and weft alternately

Trying each of the four methods generally gives an idea of how important the method used is and which is best. Figure Ω.18 shows a warp-choice draft for the pattern shown Figure Ω.17.

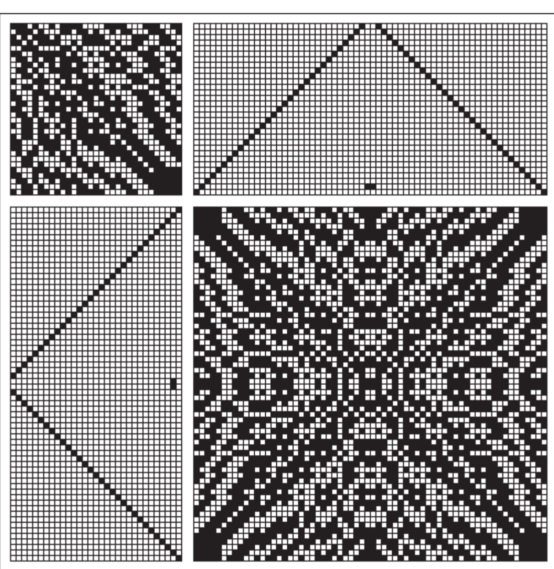


Figure Ω.18. Warp-Choice Draft

The effects on float lengths of the method of making decisions at option points are shown in Figures Ω.19 through 22.

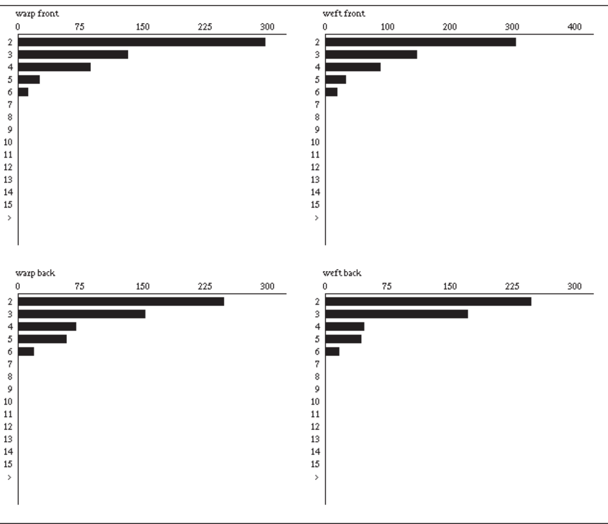


Figure Ω.19. Random-Choice Floats

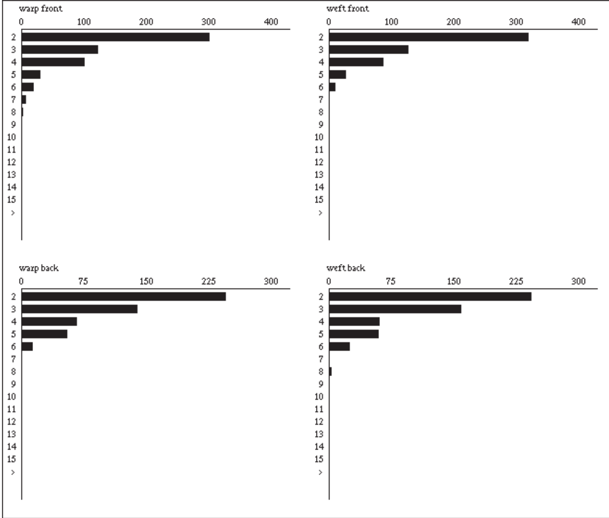


Figure Ω.20. Warp-Choice Floats

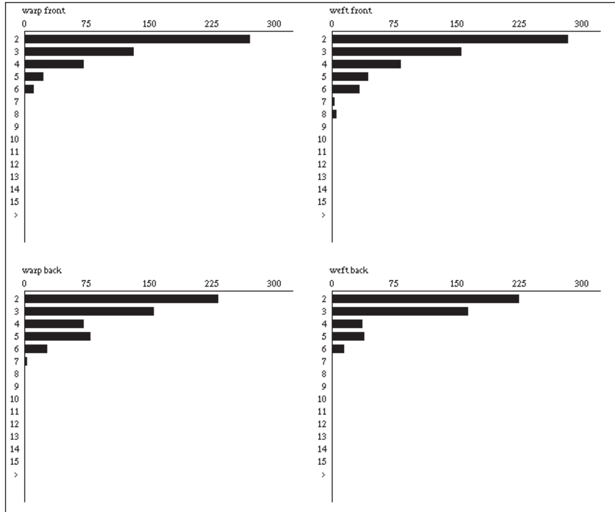


Figure Ω.21. Weft-Choice Floats

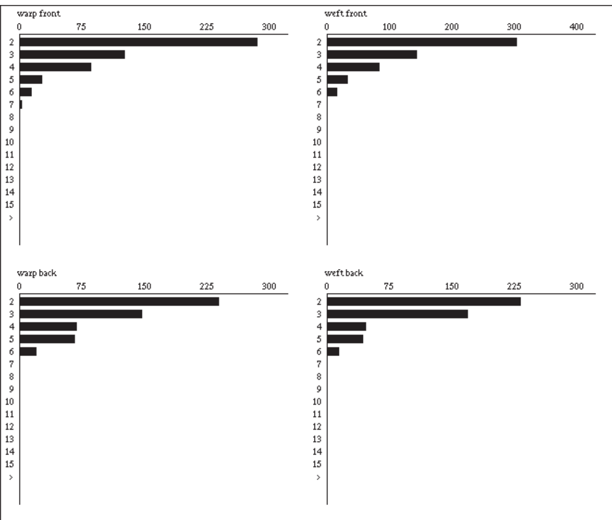


Figure Ω.22. Alternating-Choice Floats

Another, often more important, consideration is the number of shafts and treadles the draft requires. The warp-choice draft shown in Figure Ω.18 requires 31 shafts and 31 treadles. The weft-choice draft, shown in Figure Ω.23, requires only 16 shafts and 16 treadles.

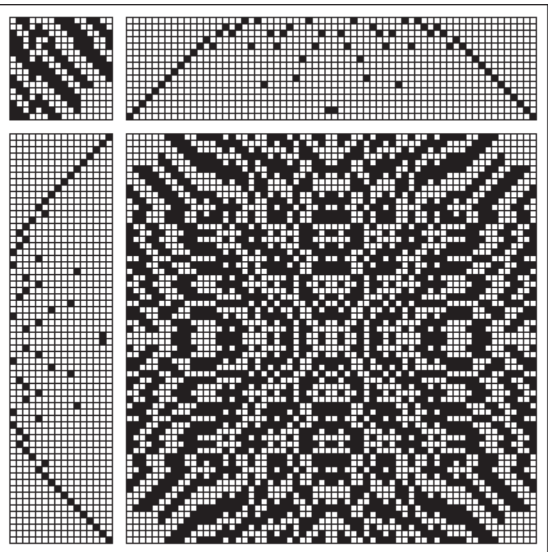


Figure Ω23. Weft-Choice Draft

More strikingly, the random-choice draft requires 56 shafts and 52 treadles, while the alternating choice draft requires 62 shafts and 31 treadles, making them out of the question for actual weaving.

July 24, 2006

Exercises and Further Explorations

The heuristic method only gives one set of colors for a solution. In some cases there may be other color assignments that satisfy the pattern, and they, in turn, might give solutions with different float structures and loom requirements. The number of alternative color assignments may be impossibly large and preclude systematic searching. The exploration of alternative solutions under the guidance of a sophisticated user might be worthwhile.

A question of more practical concern is what changes might make an undraftable pattern draftable. One possibility is to examine the effect of changes in color in forcing patterns.

Another question that could be studied is the determination of draftable subpatterns within a larger undraftable pattern.

There are many possibilities for producing better drafts (fewer shafts and treadles, shorter floats), once thread colors are determined.

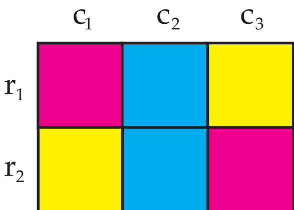


Figure 1. A Labeled Grid

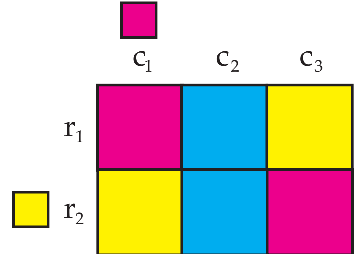


Figure 2. First Labeling Step

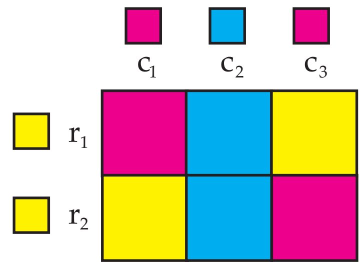


Figure 3. The Final Labeling

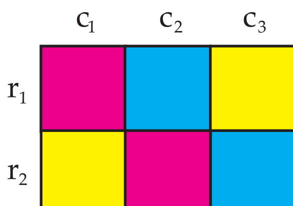


Figure 4. Another Grid

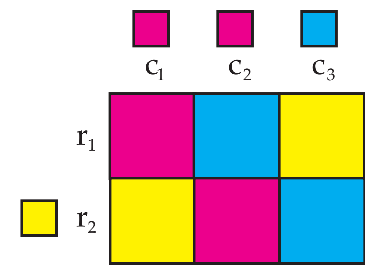


Figure 5. First Step in Labeling

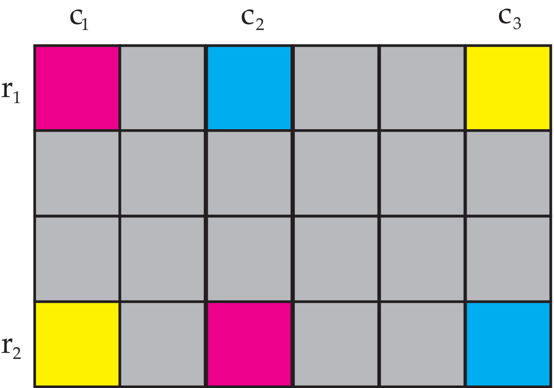


Figure 6. Separated Rows and Columns

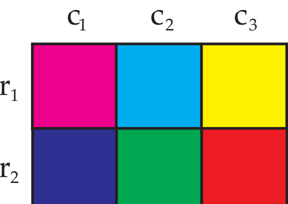


Figure 7. A Pattern with Too Many Colors

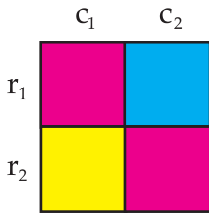


Figure 8. Three-Colored 2×2 Pattern One

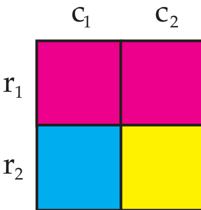


Figure 9. Three-Colored 2×2 Pattern Two

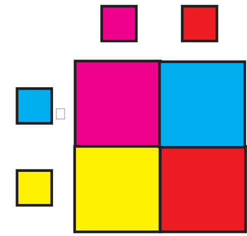


Figure 10. A 2× 2 Pattern

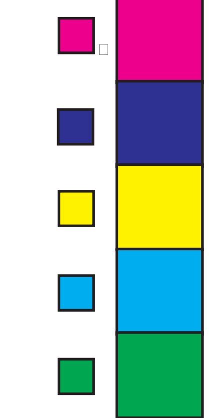


Figure 11. A 1× n pattern

25 Jul 2006 I2: ColorDraftability

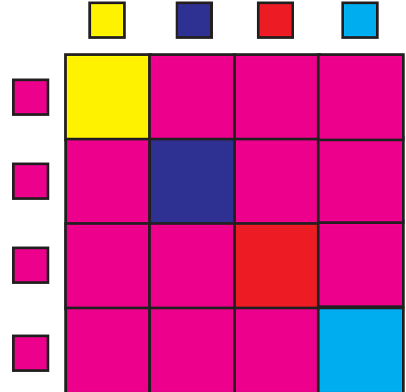


Figure 12. A 4×4 Diagonal Pattern

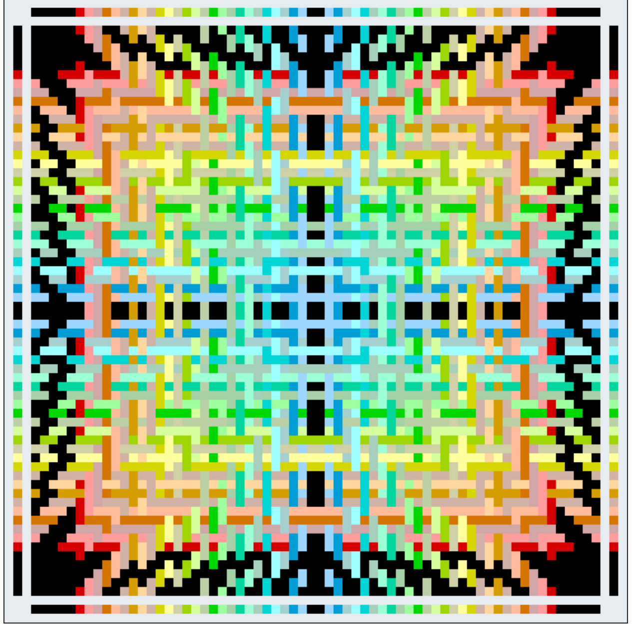


Figure 13. Solved Pattern

Maximal Color Patterns

The last section described how to determine if a color pattern is weaveable and, if so, how to create a draft for it.

This section looks at weaveable color patterns from a different perspective: How to create color patterns that are guaranteed to be weaveable and have as many colors as are possible.

In this context a color pattern is an $i \times j$ array of colored cells. An array in which every column and row is labeled with a different color is called *maximal*.

One question is how many cells are needed to create a weaveable pattern that has k different colors. Obviously, this can be done with a $1 \times k$ or $k \times 1$ pattern: a single row or column with a cell for each different color. These cases, however, are degenerate and uninteresting.

For maximal patterns, k is partitioned into two parts. There are $k - 2$ non-degenerate size combinations, given by

$$i = k - j \qquad 2 \le j \le k - 2$$

Since these arrays have $i \times j$ cells, the largest array occurs for $i = j$ or $i = j \pm 1$, depending on whether k is even or odd.

Suppose there are eight colors and a 4×4 array as shown in Figure Ω.1.

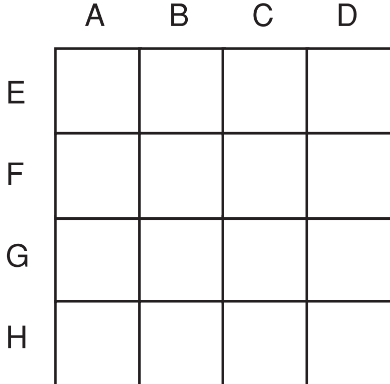


Figure Ω.1. A 4 × 4 Array

It is obvious that it's possible to have all k colors in such an array. Figure Ω.2 shows one such pattern.

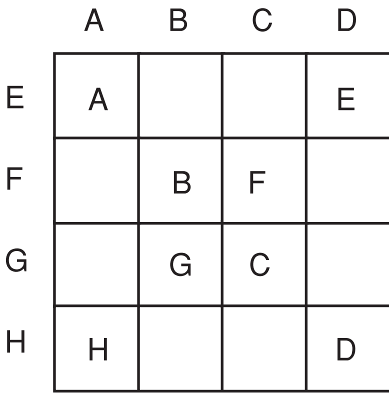


Figure Ω.2. An 8-Color 4×4 Pattern

The remaining cells in Figure Ω.2 can be colored in any of the ways the column and row labels allow. Since there are eight cells of unspecified color, there are 2^8 = 256 possible patterns based on Figure 2.

For k a multiple of four and $i = j = k / 2$, it is possible to assign colors to cells so that each color occurs $k / 4$ times ($i \times j = k^2 / 4$). Here is a coloring algorithm for constructing such color-balanced patterns:

- For each odd-numbered row, assign alternate cells the column and row colors.
- For each even-numbered row, assign alternate cells the row and column colors.

Figure Ω.3 shows the result for a 4×4 array.

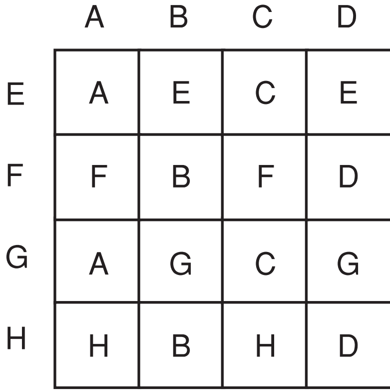


Figure Ω.3. A Balanced 4×4 Color Pattern

For other array shapes, color balance is not possible, but the coloring

algorithm given above assures k-colored patterns.

The patterns produced by this algorithm can be quite attractive. See Figure Ω.4 for an example.

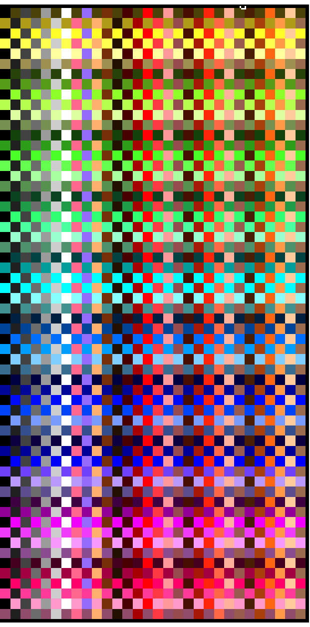


Figure Ω.4. An Algorithmic Pattern

Transformations that Preserve Weaveability

Given a weaveable color pattern, there are several kinds of changes that can be made to it that preserve Weaveability:

1. duplicating existing rows and columns
2. deleting rows and columns
3. rearranging rows and columns
4. rotating the pattern in 90° increments
5. flipping the pattern horizontally, vertically, or diagonall
6. adding solid-colored rows and columns

These changes do not require knowledge of the colors assigned to columns and rows. Here are two that do:

7. adding a column whose cells are colored either with the new column color or their corresponding row colors, and similarly for rows

8. setting the color of a cell to the color of its column or row

The first kind of change, duplicating existing rows and columns, offers many design possibilities. For example, duplicating adjacent rows and columns can be used to produce bands of any desired width. Mirroring, horiztonal, vertical, or both also follows. Figure 5 shows a weavable color pattern created using only duplications of the rows and columns of an algorithmic pattern:

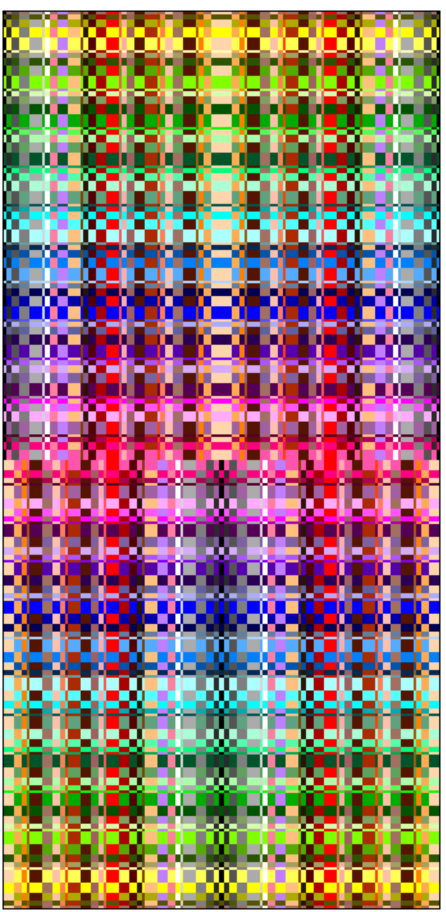
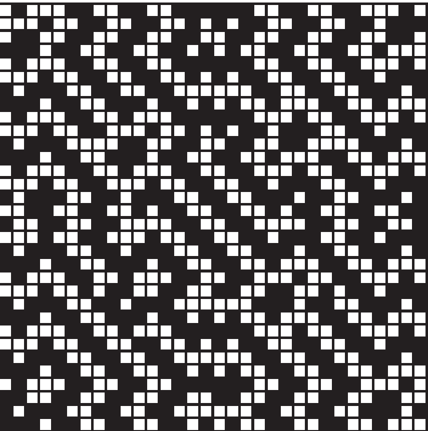


Figure Ω.5. A Weavable Color Pattern

Boolean Design of Patterns

Basic weave structures — interlacement patterns — can be described in many ways, but they all come down to representing the crossings of warp and weft threads. One or the other is on top. This is represented in drawdowns by a grid in which the cells represent the intersections and (usually) a cell is black if the warp thread is on top but white if the weft thread is on top. See Figure Ω.1.



Note: This material should be covered in the introduction to the book.

Figure Ω.1. Drawdown

From a computational point of view, 1s and 0s instead of colors are the natural representation of interlacement patterns. See Figure Ω.2, which corresponds to the drawdown of Figure Ω.1. A 1 corresponds to black for the warp thread on top and a 0 corresponds to white for the weft thread on top.

```
0100011001100111111110011001100010
0001001100110010101011001100110111
0110011001100110011001100110011001
1110110011001101010100110011001000
0100011001100111111110011001100010
0001001100110010101011001100110111
1011100110011000000001100110011101
1110110011101101010100100011001000
0100011001000111111110001001100010
0001001100010010101011011100110111
1011100001100110011001100110000111
1110110011101100110011001000110010
0100011001000110011001100010011000
0001001100010011001100110111001101
1011100110111001100110011101100111
0011001100101100110011001011001100
1001100110000110011001100001100110
0001001100010011001100110111001101
1011100110111001100110011101100111
1110110011101100110011001000110010
0100011001000110011001100010011000
0001001011001100110011001100101101
1011100110111000000001110110011101
1110110011101101010100100011001000
0100011001000111111110001001100010
0001001100110010101011001100110111
1011100110011000000001100110011101
1110110011001101010100110011001000
0100011001100111111110011001100010
1100110011001100110011001100110011
1011100110011000000001100110011101
1110110011001101010100110011001000
```

Figure Ω.2. Binary Interlacement Pattern

This binary representation suggests the use of Boolean operations for weave design.

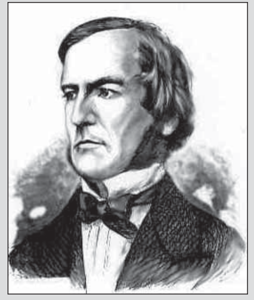
Boolean Operations

George Boole invented the mathematical system named after him to describe logical operations on *truth values* — true or false [1].

Although Boole was motivated by logic, his system applies equally well to contexts in which there are two mutually exclusive values, such as “on” and “off”. In the case of weave structures, the values are “warp on top” and “weft on top”, or alternatively, “it is true that the warp is on top” and “it is false that the warp is on top”. (The choice of warp rather than weft is arbitrary, as is the choice of 1 and 0 to represent truth values.)

In Boolean algebra [2], variables, which are indicated by x, y, and so forth, can have only two values — 1 or 0. Boolean operations produce values depend-

George Boole



1815 - 1864

George Boole grew up in poverty in England. He early exhibited intellectual powers and showed a special aptitude for mathematics.

He was unable to pursue a formal education because he had to work to support his parents. He undertook to learn mathematics on his own and soon began to do original work.

His work attracted the attention of prominent mathematicians and he began to publish in mathematical journals. He eventually was awarded a gold medal by the Royal Society.

He contributed to several areas of mathematics, but he is best known for his seminal work on the mathematics of logic. This work is the foundation for modern computing and information technology.

Because of his need to work and lack of formal training in mathematics, he started late. Unfortunately, he died early.

He had hereditary lung disease. Caught in a drenching rain and late to a lecture, he continued without changing to dry clothes. He subsequently contracted pneumonia. His wife, who thought the only way to cure a disease was to apply its cause, drenched him with cold water as he lay feverish in bed. He died shortly thereafter.

We can only speculate as to what Boole might have accomplished had he had the opportunity to get a formal education and had lived longer.

I’m trying Gregg’s suggestion of using a smaller type size for sidebars.

ing on the values of the variables to which they are applied. The operations, however, are not the familiar arithmetic ones, but rather logical operations like *and* and *or*.

Boolean operations are described by *truth tables* that detail the results depending on the value of the variables to which the operators are applied. An example is *not*, also known as *complement* and *negation*, indicated by the symbol ~. This operation has the truth table

x	$\sim x$
1	0
0	1

not

Other basic Boolean operations are *or* (+), also known as *disjunction*, and *and* (×), also known as *conjunction*. Their truth tables are

x	y	$x + y$
1	1	1
1	0	1
0	1	1
0	0	0

or

x	y	$x \times y$
1	1	1
1	0	0
0	1	0
0	0	0

and

Note that $\sim(x + y) = (\sim x \times \sim y)$ and $\sim(x \times y) = (\sim x + \sim y)$. These are known as De Morgan's Laws, named after the 19th century logician and mathematician Augustus De Morgan, who propounded them.

A particularly interesting Boolean operation is *exclusive or* (⊕), which has the truth table

x	y	$x \oplus y$
1	1	0
1	0	1
0	1	1
0	0	0

exclusive or

That is, $x \oplus y$ has the value 1 if exactly one of x and y is 1 — it excludes the case that both are. The + operation sometimes is called *inclusive or* to distinguish it from *exclusive or*.

Exclusive or has an interesting and important property:

$$(x \oplus y) \oplus y = x$$
$$(x \oplus y) \oplus x = y$$

That is, either x or y can be extracted from $(x \oplus y)$ by applying *exclusive or* with the other.

PDF problem.

The complement of *exclusive or* is *equivalence* ($\leftrightarrow$), which has the value 1 only if x and y are the same.

x	y	$x \leftrightarrow y$
1	1	1
1	0	0
0	1	0
0	0	1

equivalence

Another important Boolean operation is *implication* ($\rightarrow$), which is based on *modus ponens*, one of the foundations of logical argument: If x implies y and x is true, then y is true.

x	y	$x \rightarrow y$
1	1	1
1	0	0
0	1	1
0	0	1

implication

All together there are 16 Boolean operations of two variables, correspond-

ing to the 16 possible patterns of 0 and 1 for the four possible combinations of the variable values.

If we use the order of the values of x and y as given in the truth tables above, we can represent the 16 Boolean operations of two variables by the patterns of the values they produce.

For example, *or* and *and* have the value patterns

1110 *or*
1000 *and*

It is possible to use only a small *functionally complete* set of Boolean operations from which all 16 can be composed. Two functionally complete sets are {~, +} and {~, ×}. There are several other small functionally complete sets. In fact, there are single operations from which all others can be composed. See the side-bar **Sheffer Strokes** on the next page.

So far, we have identified Boolean operations by names and operator symbols. We can also identify them by the hexadecimal characters for their value patterns. For example, *or* has the hexadecimal identification e and *and* has the hexadecimal identification 8.

The complete list of 2-variable Boolean operations, with hexadecimal identifications and names and symbols for common operations is

0000	0		
0001	1		
0010	2		
0011	3		
0100	4		
0101	5		
0110	6	*exclusive or*	$x \oplus y$
0111	7		
1000	8	*and*	$x \times y$
1001	9	*equivalence*	$x \leftrightarrow y$
1010	a		
1011	b	*implication*	$x \rightarrow y$
1100	c		
1101	d		
1110	e	(*inclusive*) *or*	$x + y$
1111	f		

It is worth looking critically at this list. In the first place, eight of the operations are complements of the others. For example,

$$x \leftrightarrow y = \sim(x \oplus y)$$

Since complementation only changes 0s to 1s and vice versa, it can be done as a simple operation on the result.

In addition, the result of applying the operation 0, which has the value pattern 0000, is 0, regardless of the values of x and y. This operation therefore is of no use in design based on two values. The same is true of its complement, f.

Furthermore, the operation c, which has the value patterns 1100, always produces x regardless of the value of y. Similarly, the operation a, which has the value pattern 1010, always produces y regardless of the value of x. The comple-

Sheffer Strokes

In a 1913 paper [1], Henry M. Sheffer showed that one of the 16 Boolean operations constitutes a functionally complete set in itself. This operation, called the Sheffer Stroke and designated by the symbol ↓, has the value pattern 0111 and the hexadecimal identification 7. This operation can be described as *not both x and y*.

It can be shown that

$$\sim x = x \downarrow y$$

and

$$x + y = (x \downarrow y) \downarrow (y \downarrow x)$$ PDF problem.

Since it is well known that {~, +} constitutes a functionally complete set, then {↓} does also.

The second Sheffer Stoke, designated by the symbol ↑, has the value pattern 0001 and the hexadecimal identification 1. It also constitutes a functionally complete set.

While it may be interesting to know that only one operation is necessary to form all Boolean operations of two variables, using just one operation is complicated and unintuitive. These operations *are*, however, good for making up homework problems.

Reference

1. "A Set of Five Independent Postulates for Boolean Algebras, with Application to Logical Constants", *Transactions of the American Mathematical Society*, Vol. 14 (1913), pp. 481-488.

ments of these operations produce ~x and ~y, respectively.

Eliminating all the operations whose results do not depend on both x and y leaves 10 operations, of which five are complements of the other five.

The question then is what five operations to choose for design. It really doesn't matter, as long is none is the complement of another. Four that are considered most fundamental and add one other, *reverse implication*, $x \leftarrow y$, will do.

$x \times y$
$x + y$
$x \oplus y$
$x \rightarrow y$
$x \leftarrow y$

Thus, a functionally complete *design* set is $\{\sim, \times, +, \oplus, \rightarrow, \leftarrow\}$.

Boolean Operations on Arrays

In the examples above, Boolean operations are applied to individual variables, such as x and y. The operations also can be applied to arrays of Boolean variables, denoted by X, Y, and so forth. When an operation is applied to two arrays, it is applied to all the values in corresponding positions of the two arrays to give a new array. Figure Ω.3 shows an example.

X		Y		Z
1 0 1 0		1 1 0 0		0 1 1 0
0 1 0 1		0 1 1 0		0 0 1 1
1 0 1 0	⊕	0 0 1 1	=	1 0 0 1
0 1 0 1		1 0 0 1		1 1 0 0
1 0 1 0		0 0 1 1		0 1 1 0

Figure Ω.3. Boolean Operation on Arrays

Boolean operations on arrays are the basis for the methods of weave design described here. Boolean operations can be performed on interlacement patterns cast in the form of arrays of Boolean values, as shown in Figure Ω.2. For example, Figure Ω.3 shows the result of applying *exclusive or* to a small plain weave and a 2/2 twill to produce a new interlacement pattern (which is just a shifted 2/2 twill).

An Application — Diversified Weaves

Oelsner, whose well-known book was first published in 1915 [3], includes a chapter on what he calls *diversified weaves*. The chapter opens as follows:

> Very attractive patterns can be obtained by adding or removing risers from a ground weave. These alterations are made according to a previously selected motif.

Many of his examples classify as spot weaves, but he goes beyond that.

The most common ground weave is plain weave, although others can be used. Figure Ω.4 shows a spot weave obtained by applying *or* to a plain weave and a repeated motif.

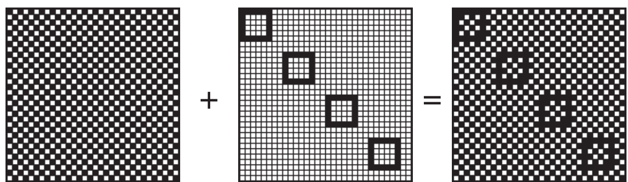


Figure Ω.4. A Spot Weave

Other Boolean operations produce different patterns. The result of applying *exclusive or* is shown in Figure Ω.5.

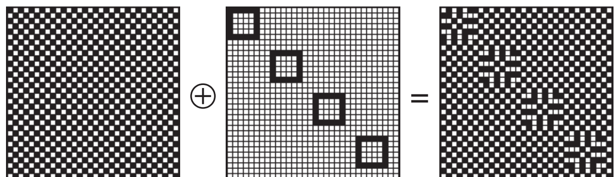


Figure Ω.5. *Exclusive Or* Spot Weave

Other weaves can be used for the ground. Figure Ω.6 shows a 2/2 twill in place of the plain weave in Figure Ω.5.

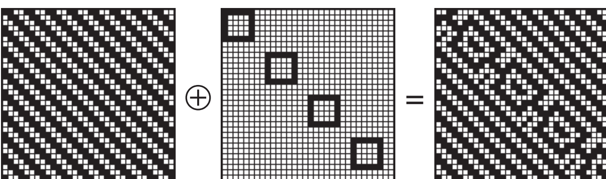


Figure Ω.6. Twill-Based Spot Weave

Other Applications

The concept of combining figure and ground implies a fundamental distinction between the two. Boolean operations can, of course, be applied to any two weaves. The question is what kinds of weave and which operations produce good results. Figure Ω.7 shows the result of combining a 2/2 twill with a basket weave.

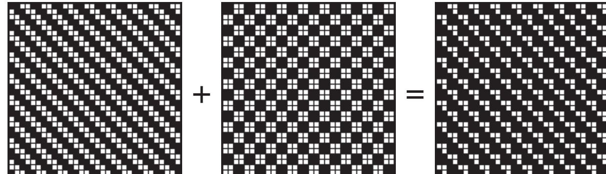


Figure Ω.7. Combining Plain and Basket Weaves

Comments

Boolean operations provide a natural way to combine two (or more) interlacement patterns to form new ones. As in all matters of design, artistic sense and understanding of the tools used are essential.

There are two potential problems associated with using Boolean operations in weave design: long floats and the loom resources required.

In Boolean operation value patterns, 1s tend to add warp floats and 0s tend to add weft floats.

The "balanced" Boolean operations, which have two 1s and two 0s in their value patterns generally cause fewer problems with floats than the unbalanced ones. Note that there is only one balanced operation, *exclusive or*, in the functionally complete design set described earlier.

The problem of loom resources is more serious and difficult to gauge in design. One of Oelsner's examples of diversified weaves has 60 ends and 60 picks — and would require 60 shafts and 60 treadles (the maximum possible for a weave of this size). While this weave could be done on a drawloom, it really is in the province of Jacquard weaving.

Advanced Boolean Design

In this article, one Boolean operation has been applied to all the variables in two interlacement patterns.

The next article describes the use of arrays of Boolean operations, in which different operations can be applied to different parts of the interlacement patterns.

"Next article" to be added here.

L-Systems

Civilization is founded on language. Our accumulated knowledge has been preserved in writing. We communicate using language.

Everyone knows at least one *natural language*, such as German, French, or English. The term natural has arisen because of languages that have not evolved in the normal course of human civilization. The most well known of these "unnatural" languages are programming languages, such as FORTRAN, C, and Java, which contain instructions for running computers.

Strings

A central component of languages is the *string*, which is a sequence of characters, such as a word, a phrase, or a telephone number. The characters may be letters, digits, punctuation marks, dollar signs, and so forth. Uppercase letters are used in examples to make them stand out, as in ABCBA.

The characters in strings may or may not have meanings associated with them. For the time being, they will just be abstract symbols. Strings may of course, contain patterns; in fact, this is the major interest here. For example, the string ABCBA is a palindrome, reading the same way forward and backward. Another example is DEDEDEDE, which consists of four repetitions of DE.

A string within a string is called a *substring*. For example, DE, ED, and DED are among the many substrings in DEDEDEDE.

The length of a string is the number of characters in it. For example, the length of ABCBA is 5. The *empty string*, consisting of no characters, has length 0. The empty string is denoted by Θ.

A single character is a one-character string.

Concatenation consists of appending one string to another. For example, the concatenation of ABCBA and DE produces ABCBADE.

Formal Language Systems

Linguists studying languages have developed *formal language systems* as models for exploring the expressive powers of different kinds of languages. Most formal language systems are arcane and known only to specialists. A few, however, have practical applications — sometime surprising ones.

A formal language system consists of three parts:

- (1) an *alphabet*, which might consist of letters such as A, B, C, …
- (2) a *grammar* that determines how *sentences* in the system are produced

- (3) a *seed*, which is the starting sentence.

A sentence is a string of symbols in the alphabet, such as BA, BCD, DA, … A *language* for the system is a set of sentences the grammar can produce.

Grammars usually consist of *rewriting rules* that describe how one sentence is produced from another.

Here is an example:

Alphabet: A and B

Rewriting Rules:

1: A ➛ BA

2: B ➛ AB

Seed: A

Grammar: a randomly chosen letter of the current sentence is replaced according to its rewriting rule to produce the next sentence.

The resulting language might be

sentence	*rules*	*generation*
A	seed	0
BA	Rule 1 applied to first character (A)	1
ABA	Rule 2 applied to first character (B)	2
ABBA	Rule 1 applied to last character (A)	3
AABBA	Rule 1 applied to second character (B)	4

…

Different random choices would, of course, produce different languages.

The process of producing a language, starting with a seed, is called *generation*. The seed is generation 0; subsequent generations are numbered 1, 2, 3, … a shown bove.

Phrase-Structure Languages

One common formal language system uses *phrase-structure grammars*. In such grammars, there are alternative rewriting rules, such as

A ➛ BA

A ➛ AB

The rewriting rule used for a symbol is chosen at random. Sentences are produced starting with the seed and picking one symbol in the sentence to

replace, also at random.

Here's an example:

Alphabet: W, C, S, P, 1, 2, 3, 4

Rewriting Rules:

1: W ➛ 1

2: W ➛ 2

3: W ➛ 3

4: W ➛ 4

5: C ➛ c

6: C ➛ y

7: C ➛ m

8: S ➛ WC

9: P ➛ S

10: P ➛ SP

Seed: P

One language for this grammar is

sentence	*character*	*rule*	*generation*
P			0
SP	1	10	1
SS	2	9	2
WCS	1	8	3
WCWC	3	8	4
4CWC	1	4	5
4CWm	4	7	6
4C2m	3	2	7
4c2m	2	5	8

That's the end of the language, since there are no more symbols with rewriting rules.

This example may seem pointless, but suppose W stands for width, C for color, S for stripe, P for pattern, c for cyan, y for yellow, and so on, with the

numerals standing for themselves. Then this grammar generates examples of simple stripe patterns. The one above has two stripes: cyan of width 4 and magenta of width 2.

Note that P is defined in terms of itself in rule 10:

P ➛ SP

This introduces the concept of recursion, which is a source of complexity. More on this important mechanism later.

L-Systems

Phrase-structure grammars are interesting, but they lack expressive power — that is, the complexity of patterns they can produce is limited and not very interesting. Among the more powerful and interesting formal language systems are Lindenmayer Systems, or L-Systems for short, which are named after their inventor, Aristid Lindenmayer.

Aristid Lindenmayer

Aristid Lindenmayer was a Hungarian biologist. His invention of the string rewriting system named after him "grew out of an attempt to describe the development of multicellular organisms in a manner which takes genetic, cytological and physiological observations into account in addition to purely morphological ones" [2].

1925 - 1985

It was a surprise to him when his system began to be used in many other disciplines and in many ways. In fact, he first rejected as a gimmick the graphical interpretation of L-System strings to draw plants, something he never envisioned when he started to manipulate strings of characters. Drawing ultimately proved to be the most influential use of L-Systems.

There are L-Systems of many different kinds and degrees of complexity, including ones that can be used to represent geometrical forms and even three-dimensional plants in color and [3-5]. Active research in the area continues to this day.

The distinguishing characteristic of L-Systems is that all rules are applied in parallel for each generation and every symbol is rewritten. In the simplest kind of L-System, there are no alternative rules.

Here is an example L-System:

seed: ABCD
rules: A ➛ BD
B ➛ B
C ➛ ACA
D ➛ Θ

The first rule specifies that A is replaced by BD. The second rule specifies that B is replaced by B; that is, it is unchanged. The third rule specifies that C is replaced by ACA, while the fourth rule specified that D is to be replaced by the empty string; that is, deleted.

The generation goes like this:

string	*generation*
ABCD	0
BDBACA	1
BBBDACABD	2
BBBBDACABDB	3
BBBBBDACABDBB	4
BBBBBBDACABDBBB	5
BBBBBBBDACABDBBBB	6
…	…

Bs accumulate and each generation is longer than the previous one.

Despite their apparent simplicity, L-Systems are very powerful. Their languages may contain intricate patterns with infinitely varying subtlety. Among other things, they can describe plant development (their original use), fractals, and complex geometric designs.

The power of L-Systems comes from parallel rewriting and repeated application (iteration) of the rules. These properties are of fundamental importance and apply to entirely different mechanisms, such as cellular automata [1], and to many processes in the physical world.

Example L-Systems

Example 1: The Morse-Thue sequence [6] is produced by a very simple L-System:

seed: A
rules: A ➛ AB
B ➛ BA

The generation goes like this

A
AB
ABBA
ABBABAAB
ABBABAABBAABABBA
...

Note that the lengths of the strings double with each generation.

Example 2: The Fibonacci string sequence, analogous to the Fibonacci number sequence [7], is produced by this L-System:

seed: A
rules: A ➛ B
B ➛ AB

Generation goes like this:

A
B
AB
BAB
ABBAB
BABABBAB
ABBABBABABBAB
BABABBABABBABBABABBAB
...

Note that the lengths of the generations give the Fibonacci numbers: 1, 1, 2, 3, 5, 8, 13, 21, …

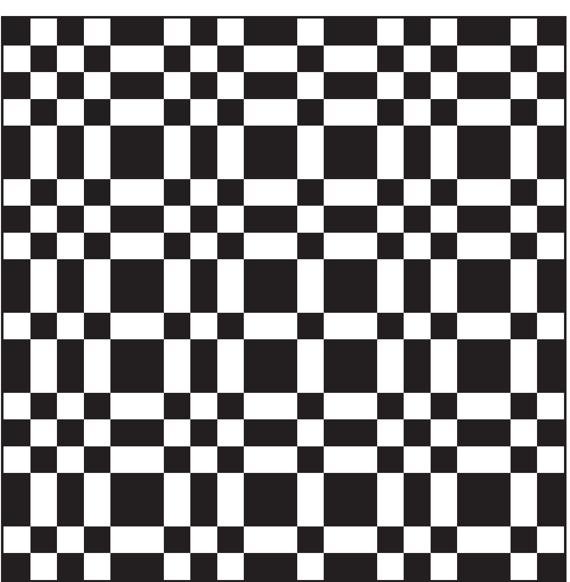
Incidentally, no phrase-structure grammar can generate either the Morse-Thue string sequence or the Fibonacci string sequence.

Interpreting L-System Languages

The strings produced by L-Systems such as those given above have evident patterns but the characters themselves have no meaning. If A and B in the Morse-Thue example are interpreted as 0 and 1, respectively, the results is the usual interpretation of the Morse-Thue sequence as a sequence of binary digits.

Many other kinds of interpretation are possible. One of the most striking methods interprets characters as drawing commands. The next section describes the drawing interpretation.

A very natural interpretation of strings like the Fibonacci strings is as profile sequences. Here is a profile pattern for the seventh-generation Fibonacci string:



Profile Drafting

In fact, the simplest and most straightforward use of L-Systems is in the design of profile drafts.

Here is another example:

seed: A

rules: A ➛ ABB

B ➛ BCC

C ➛ CAA

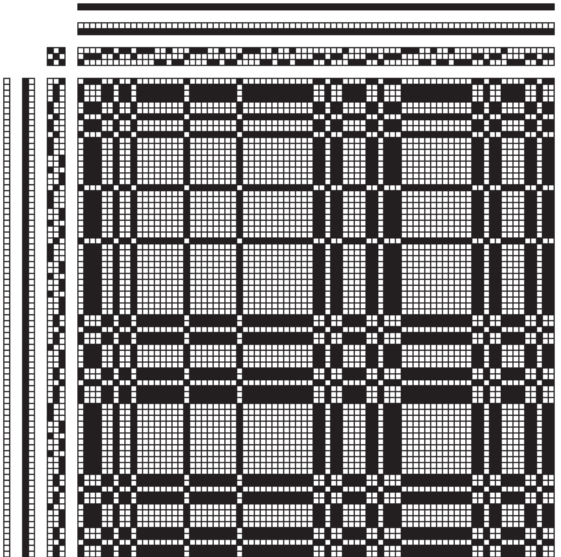
The first four generations are

	generation
ABB	1
ABBBCCBCC	2

ABBBCCBCCBCCCAACAABCCCAACAA 3

ABBBCCBCCBCCCAACAABCCCAACAA 4
BCCCAACAACAAABBABBCAAABBAB
BBCCCAACAACAAABBABBCAAABBABB

Any of these generations beyond the trivial one can be used for profile “threading” and “treadling” sequences. From here on out, we’ll drop the quotes. Here is a draft that uses generation 4 for threading and treadling:



The number of blocks is, of course, the number of symbols for which there are replacement rules. By the nature of L-Systems, useful ones for profile drafting are limited to four or five blocks; otherwise the generated strings quickly get too long.

In designing L-systems for profile drafts, the distinguishing characteristic is the repetition of blocks to create varying pattern widths. A profile sequence without block repetitions is, of course, possible, but it is indistinguishable from a treadling sequence, and has limited utility for profile design.

Block repetition is easy to build into L-Systems. The problem is more one of avoiding excessively long generation strings. See the preceding example, which has only three symbols. In this L-System, every generation is three times as long as the preceding one: 3, 9, 27, 81, 243, … Since profile drafts are the source of threading drafts with more than one thread per block, the limitations are clear.

Another consideration is symmetry. Palindromic mirroring is a powerful

tool for producing attractive, even compelling patterns. Palindromes can be added after the fact, but they can be designed into L-System by the simple expedient of making all the rules palindromic. Here is an example:

seed: A

rules: A ➛ ABBA

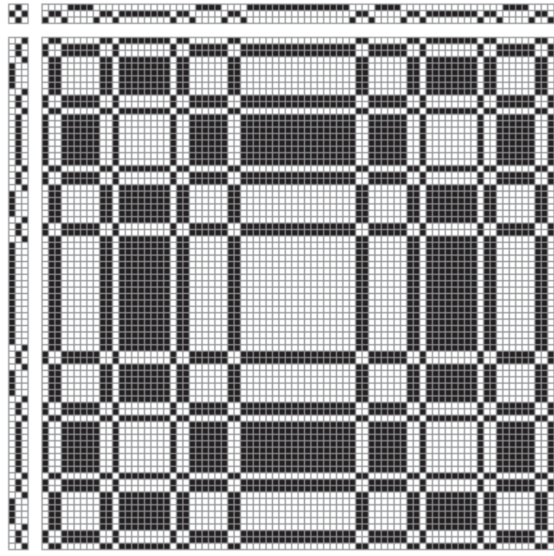
B ➛ CC

C ➛ BB

The generations are

	generation
ABBA	1
ABBACCCCABBA	2
ABBACCCCABBABBBBBBBBABBA CCCCABBA	3
ABBACCCCABBABBBBBBBBABBA CCCCABBACCCCCCCCCCCCCCCC CCABBACCCCABBABBBBBBBBA BBACCCCABBA	4

A profile draft for generation 4 is



Other Ideas

There is a wealth of ways that profile drafts can be produced from L-Systems.

One idea is to use one L-System for the threading and another for the treadling.

Remember that interpretation is a very powerful tool when using L-Systems. Among the many possibilities is the use of a width sequence to replicate each successive block in a string by some number of times.

T-Sequence Design

One obvious ue of L-Systems is in the design of threading and treadling sequences.

Here is an example:

seed: 123456787654321

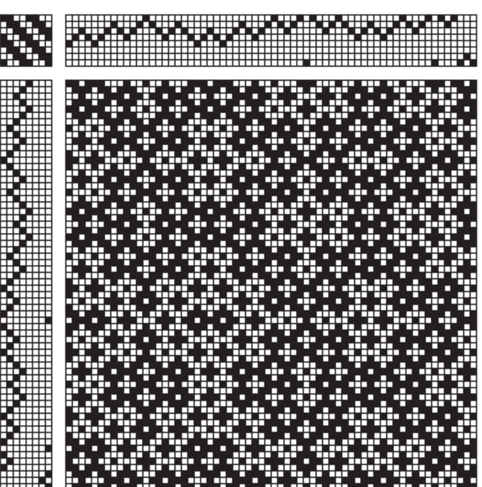
rules: 1 ➛ 23432
2 ➛ 34543
3 ➛ 45654
4 ➛ 56765
5 ➛ 67876
6 ➛ 78187
7 ➛ 81218
8 ➛ 12321

Thus every 1 in a string is replaced by 23432 and so on.

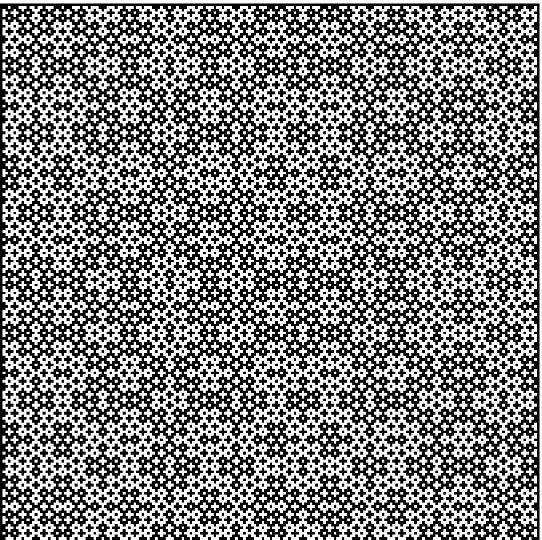
The length of successive generations increases by a factor of 5. The first generation is

234323454345654567656787678187812I8
12321812187818767876567654565434
5432343

Here is a partial draft using the third generation of this L-System for threading and treadling sequences:



Here’s the weave design:



Note: This is a crackle weave.

A Side Trip to Graphics

The first section on L-Systems showed an example in which the characters W, C, S, and P had mnemonic value to suggest widths, colors, stripes, and patterns. These characters themselves have no meaning — they are just suggestions. Striped patterns were an interpretation of the characters. The characters could just as well have stood for wind direction, capacity, speed, and pulse.

L-Systems generate abstract patterns of characters. Interpretation translates these patterns into meaningful, concrete objects.

One of the most striking and well-known interpretations of L-Systems is to produce graphic images. This section illustrates that. It has nothing to do with weaving, *per se*, but it does suggest how L-Systems might be used in weave design.

Consider this L-System:

seed: A

rules: A ➛ BCDDAEFAEFBDFBAECA

B ➛ BB

It's not at all obvious what motivates this particular L-System or why it might be interesting, although the complexity of the first rule suggests some intent. The lack for rules for C, D, E, and F seems curious, although they proliferate during rewriting since the default in such cases is to replace such characters by themselves.

Now consider this L-System, which is the same as the one above except that different characters are used.

seed: X

rules: X ➛ F–[[X]+X]+F[+FX]–X

F ➛ FF

The characters look a bit strange — this is the first L-System with characters other than letters. There is a reason for the characters chosen, however. They serve as mnemonic devices for the intended interpretation, which is as commands for a drawing program:

F move forward a specified length, drawing a line

f move forward a specified length, without drawing a line (not included in the example above)

\+ turn right a specified number of degrees

– turn left a specified number of degrees

[save the current position and direction

] restore the previously saved position and direction

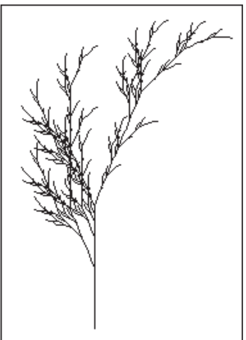
The character X in this L-System is a placeholder. It participates in an important way in the patterns produced, but it is ignored in interpretation.

Two parameters are needed to carry out the interpretation:

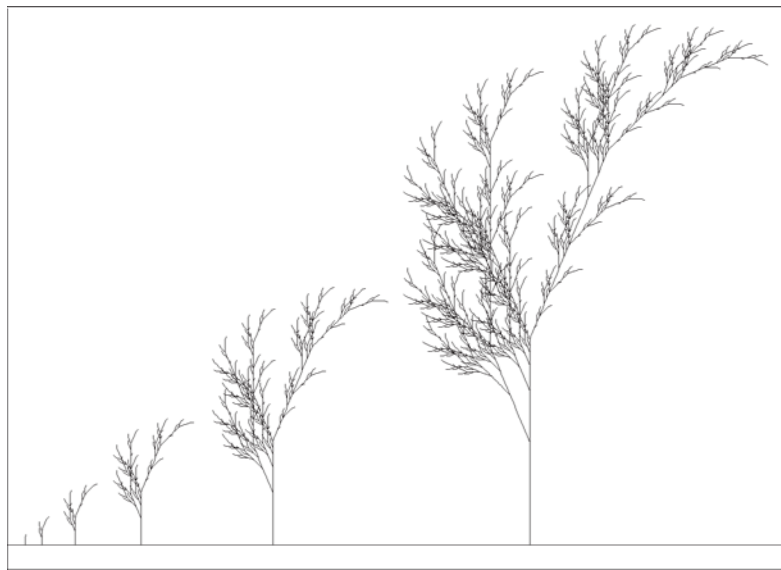
- The length for a move, which determines the scale of the drawing.

- The angle for turns, which is fundamental to the appearance of the result produced. For this L-System it is 22.5º — 1/4 of 90º.

The drawing is accomplished by producing several generations of the L-System and then interpreting the last one. For five generations, the image from interpretation is



Each generation increases the size and detail of the “tree”.

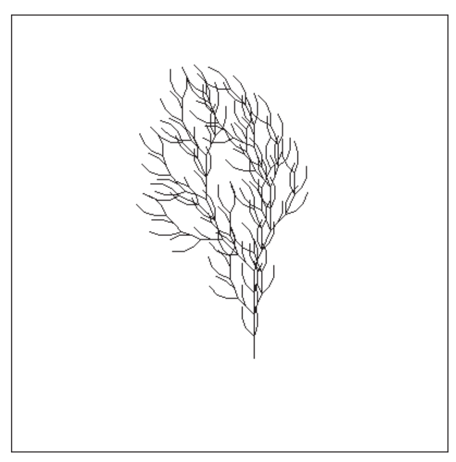


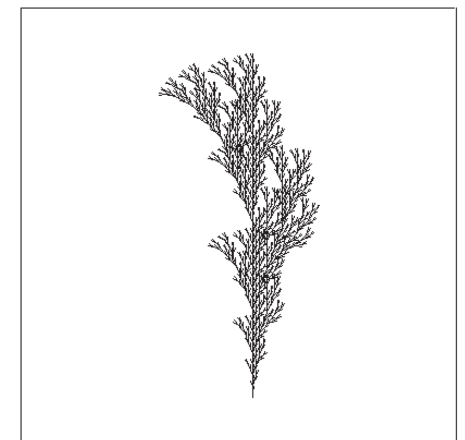
The strings produced by this L-System become very long as rewriting continues:

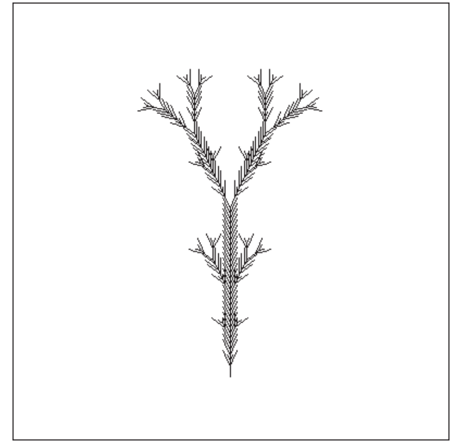
generation	*length*
1	90
2	380
3	1,552
4	6,264
5	25,160
6	100,840
7	403,752
8	1,615,784

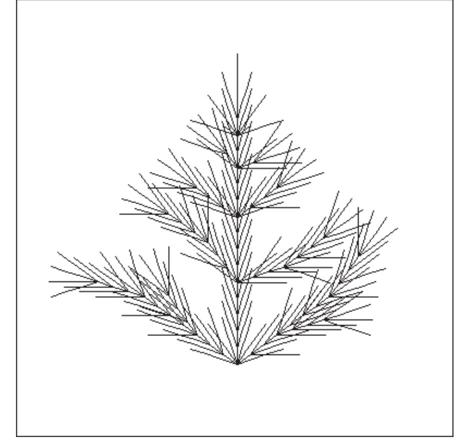
Nevertheless, it is not necessary to look at these strings or even produce them all at once. Only the drawing program uses them, and it can take the characters one by one. What these lengths *do* indicate is how many drawing actions are needed to produce the detail in the images.

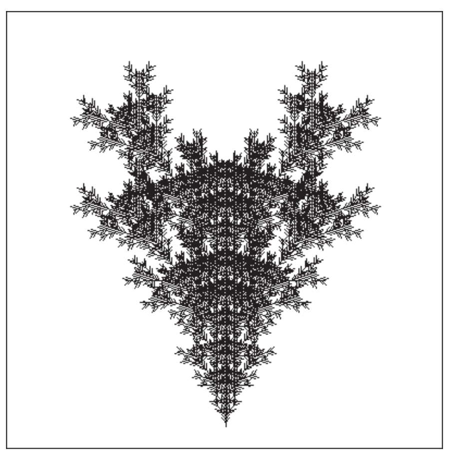
Plants drawn by L-Systems may attempt reality or approch the bizarre. Some examples are shown on the opposite page.

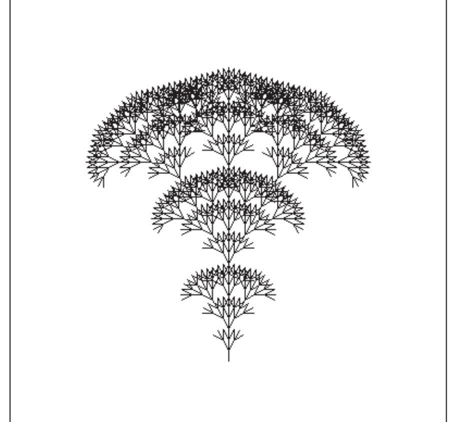




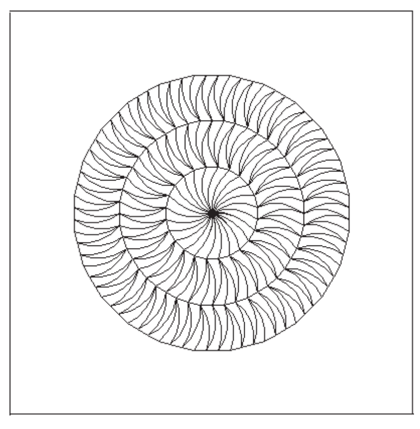


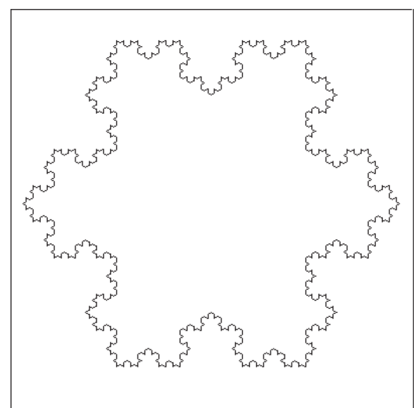


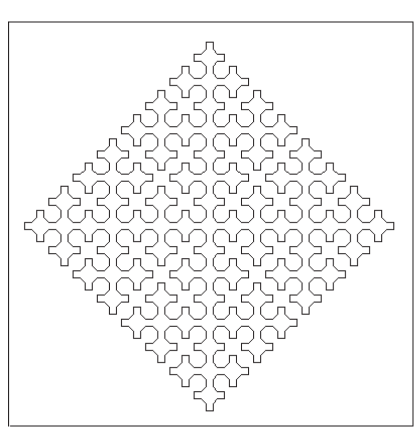


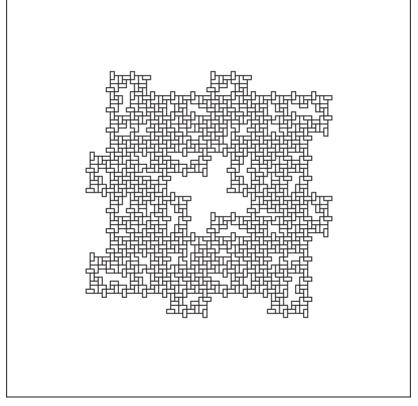


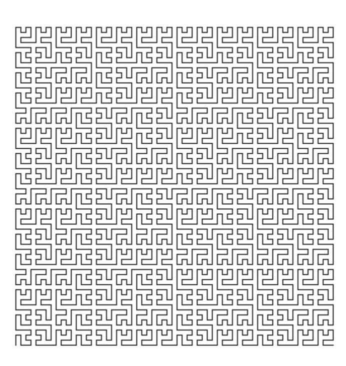
Although the most well-kown drawings produced by L-Systems are those of plants, L-systems also can be used to draw fractals.

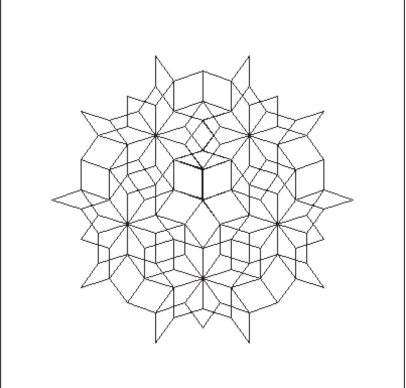












T-Sequence Expressions

The last section showed showed how L-Systems. with appropriate interpretation, can be used to produce pictures. A very different kind of interpretation can be used to produce t-sequence expressions. [Possible problem of reference order.]

A t-sequence expression with undefined variables represents all the possible t-sequences that can be produced by giving all possible values to the undefined variables during interpretation.

The usefulness of this idea is illustrated by the following examples.

Example 1

seed: S
rules: S ➛ pal(T)
T ➛ motif(X,V)
X ➛ hor(Y)
Y ➛ motif(U,V))

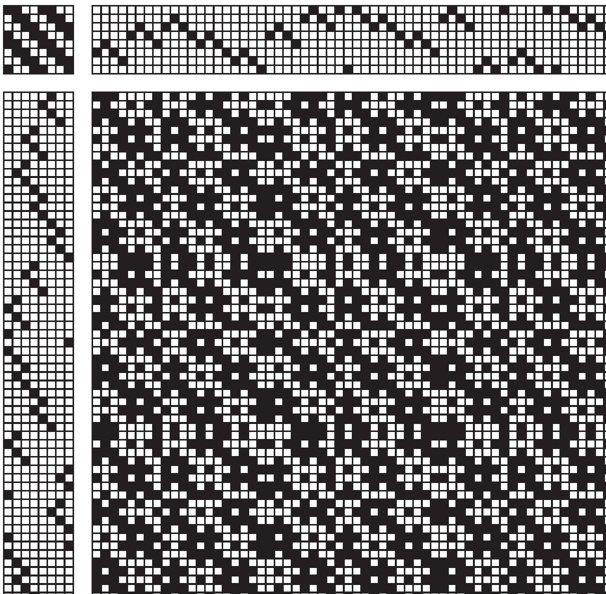
The terminal generation is

pal(motif(hor(motif(U,V)),V))

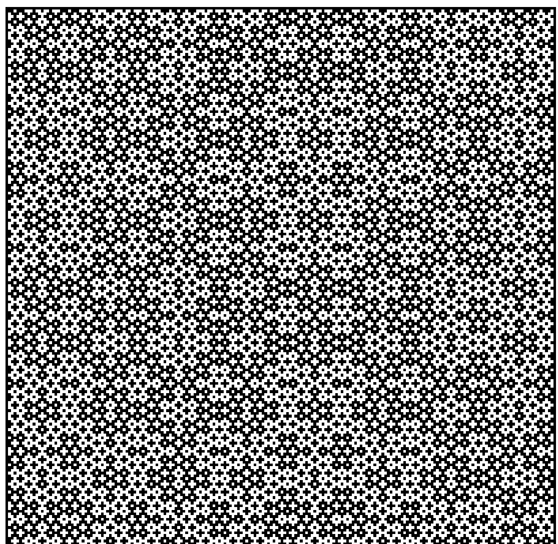
Given the values

U := [1,2,3,2]
V := [1,3, 5, 4, 2]

a draft based on the resulting sequence is:



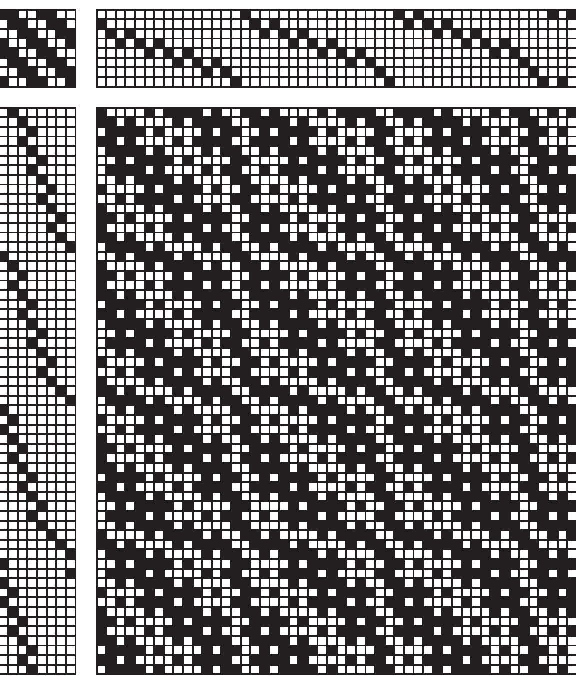
Here is the weave pattern:



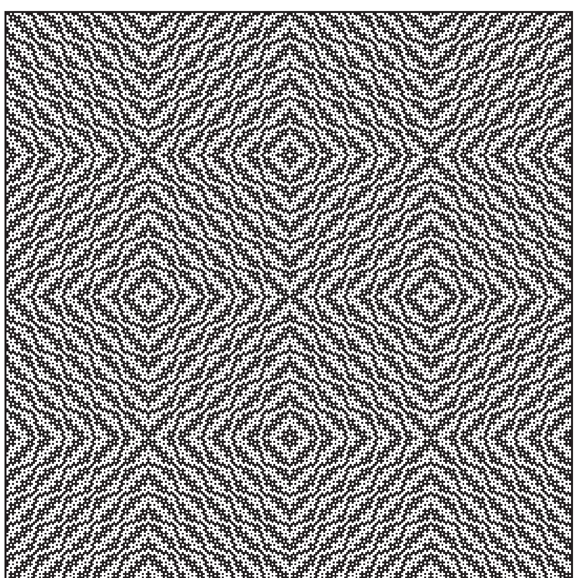
On the other hand, given the values

```
U := [1,2,3]
V := [1,2,3,4,5]
```

a draft based on the resulting sequence is:



Here is the weave pattern:



Example 2

seed: S

rules: S ➛ pal(T)

T ➛ coll(U,V)

U ➛ pal(X)

V ➛ pal(Y)

The terminal generation is

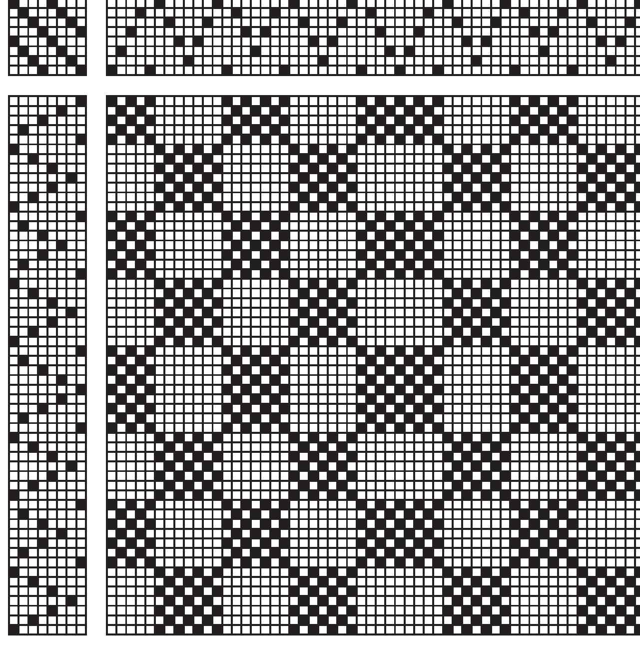
pal(coll(pal(X),pal(Y)))

Given the values

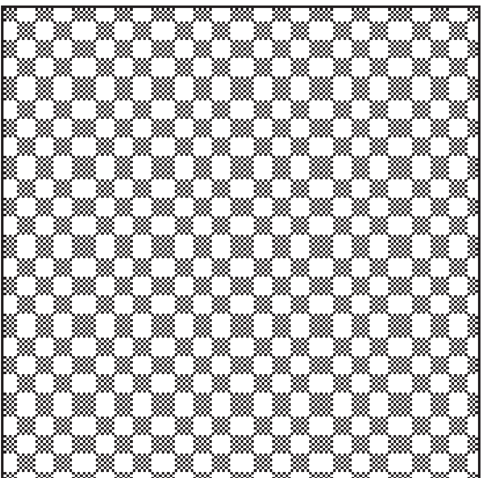
X := [1,3,5,7,9,8,6,4,2]

Y := [6,4,2,7,5,3]

a draft based on the resulting sequence is:



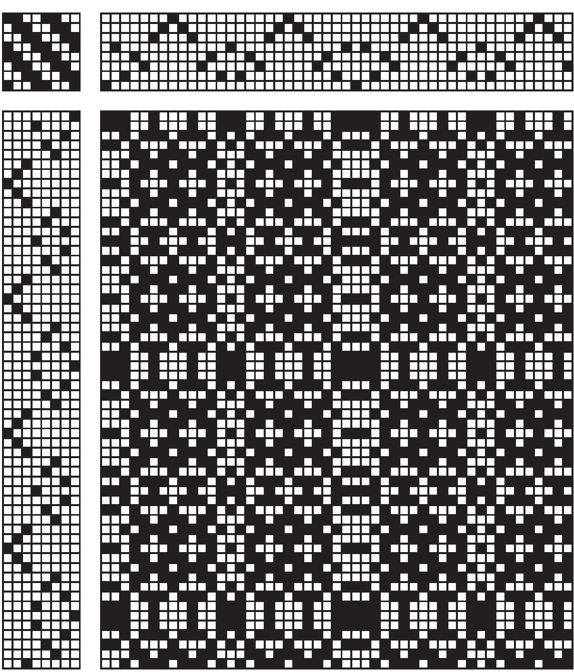
Here is the weave pattern:



On the other hand, given the values

X := [1,5,2,4,3,6,7,8]
Y := [6,4,2,7,5,3]

a draft based on the resulting sequence is:



Here is the weaving pattern:

Resources

Many L-System programs are available on the Web as freeware and shareware programs. Almost all of them are designed to produce images — to such an extent that a person who didn't know otherwise might assume that's all there is to L-Systems.

Lparser [2] is a particularly capable freeware L-System application. Here is an "air horse", which shows what is possible using Lparser:

Articulated L-Systems

Variables and Constants

L-Systems have no concept of characters that play different roles. Most rewriting systems distinguish between variables and constants. Variables have associated replacement strings; constants do not and just stand for themselves.

In an L-system, if there is no rule for a character, it is replaced by itself. In practical terms, it's a constant.

Articulated L-Systems distinguish between variables and constants. The mechanism for rewriting and producing successive generation remains the same. The only difference is in the classification of characters. In articulated L-Systems, characters are divided into two classes as described above: constants and variables.

This L-System to produce graphic images illustrates the difference:

seed: X
rules: X ➛ F–[[X]+X]+F[+FX]–X
F ➛ FF

Here X and F are variables, while –, +, [, and] are constants.

To help distinguish variables from constants in examples that follow, uppercase letters are used for variables and all other characters are constants. This typographic distinction is just a matter of convenience; there is nothing fundamental about it.

Constants are just what their name implies. They are not replaced in rewriting (which is a more useful idea than the one that they are replaced by themselves).

Defined and Undefined Variables

Variables in turn are divided into two classes: defined and undefined. A defined variable is one for which there is a rewriting rule. An undefined variable is one that appears in an L-System but for which there is no rewriting rule. During rewriting, undefined variables are treated like constants, but they play a different conceptual role.

Previous examples of L-Systems have had no undefined variables. An undefined variable can serve two purposes. One is as a placeholder for a long constant string. An example is

seed: S
rules: S ➛ STS

Here, T is an undefined variable. Successive generations are

S
STS
STSTSTS
STSTSTSTSTSTSTS
…

If T had been given the rule

T ➛ abcbbca

the third generation would have 42 more characters.

Undefined variables used as placeholders need not be added to L-Systems; values for them can be provided during interpretation.

Base L-Systems

Another use for undefined variables is in designing *base* L-Systems that can be supplemented by definitions for undefined variables.

Consider the previous L-System supplemented by a definition for T:

seed: S
rules: S ➛ STS
T ➛ aSb

Successive generations are:

S
STS
STSaSbSTS
STSaSbSTSaSTSbSTSaSbSTS
…

On the other hand, with a different definition for T, as in

seed: S
rules: S ➛ STS
T ➛ SabS

successive generations are

S
STS

STSSabSSTS
STSSabSSTSSTSabSTSSTSSabSSTS
…

Although the generations of these two L-Systems are different, they both reflect the common part of their base L-System.

Base L-Systems can be used as a tool for designing L-Systems incrementally by giving them variable definitions.

Termination

Most L-Systems produce longer and longer strings with each successive generation, and do this endlessly. This is intentional in the design of most L-Systems, where successive generations produce more complex and detailed patterns. Generation goes on endlessly because the rewriting rules contain variables [1]. Such L-Systems are called *nonterminating*.

It is possible to design nonterminating L-Systems that "loop" and have only a fixed number of different generations. A simple example is

seed: X
rules: X ➛ Y
Y ➛ X

where the generations are:

X
Y
X
Y
…

Such L-Systems are contrived aberrations and are not interesting for design purposes.

It is also possible to design L-Systems in which generation leads to a string with no defined variables. In this case, all subsequent generations would be the same, and generation effective terminates. Such L-Systems are called *terminating*.

An example of a terminating L-System is

seed: X
rules: X ➛ YY
Y ➛ ZaZ

where the generations are

X
YY
ZaZZaZ

In this case, different strings can be provided for Z during interpretation to give different results.

Although terminating L-Systems are limited in the variety of patterns they can produce, they are nonetheless useful in design.

Consider, for example, this L-System:

seed: X
rules: X ➛ Y, 1, 2, 3, Y

Generation quickly terminates with the string

Y, 1, 2, 3, Y

If a rule for Y is added

Y ➛ 4, 3, 2

the result is

4, 3, 2, 1, 2, 3, 4, 3, 2

On the other hand, if

Y ➛ 1, 2, 3, 4, 3, 2

the result is

1, 2, 3, 4, 3, 2, 1, 2, 3, 1, 2, 3, 4, 3, 2

Put in words, this L-System characterizes all strings that have two instances of a given string separated by 1, 2, 3. This is, of course, obvious. But the idea can be used as a design tool.

For example, the next step might be to provide a rule for Y that contains a variable:

Y ➛ 4, 3, 2, Y

This results in a nonterminating L-System with endless generation:

X
Y, 1, 2, 3, Y
4, 3, 2, Y, 1, 2, 3, 4, 3, 2, Y
4, 3, 2, 4, 3, 2, Y, 1, 2, 3, 4, 3, 2, 4, 3, 2, Y
4, 3, 2, 4, 3, 2, 4, 3, 2, Y, 1, 2, 3, 4, 3, 2, 4,
3, 2, 4, 3, 2, Y
...

Generating T-Sequence Expressions

The section describes L-Sysstems that generate t-sequence expressions, as opposed to actual t-sequences.

This example illustrates the idea:

seed: S
rules: S ➛ pal(T)
T ➛ rpt(U,I)

This is a terminating L-System with only two generations:

pal(T)
pal(rpt(U,N))

The interpretation of these strings, as the characters used suggest, is that pal is a function that produces a palindrome from its argument and rpt is a function that produces a repeat of its first argument a number of times specified by its second argument. Note that although p, a, l, r, and t, are individual constant characters in the L-System, pal and rpt can be treated as strings during interpretation.

If U is given the value [1, 2, 3, 4] and N the value 2 during interpretation, the result is

[1, 2, 3, 4, 1, 2, 3, 4, 3, 2, 1, 4, 3, 2, 1]

The L-System above could, of course, be derived from its final generation:

pal(rpt(U,N))

The value of using an L-System to characterize the patterns rather than just using the t-sequence expression it produces is that the components are represented in separate rules and hence easy to understand, while t-sequence expressions may be complicated and deeply nested, which is difficult for human beings (but not computer programs) to understand, and may be difficult to construct by hand.

The sections on t-sequneces cast operations in an abstract operator notation using a variety of mathematical symbols and typographical devices.

For example, the expression from the example above,

pal(rpt(U,N))

is written in the abstract operator notation as

$\cap(U \times n)$

T-Sequence Models

The last section showed how terminal L-Systems can be used to characterize t-sequences in terms of t-sequence expressions.

A t-sequence expression with undefined variables represents all the possible t-sequences that can be produced by giving all possible values to the undefined variables during interpretation.

The usefulness of this idea is illustrated by the following examples.

Example 1

seed: S

rules: S ➛ pal(T)

T ➛ motif(X,V)

X ➛ hor(Y)

Y ➛ motif(U,V))

The terminal generation is

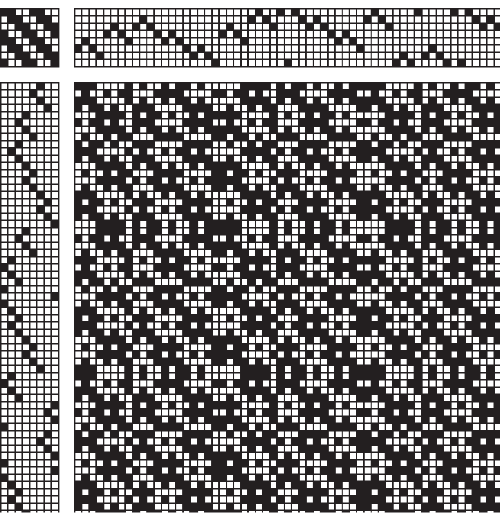
pal(motif(hor(motif(U,V)),V))

Given the values

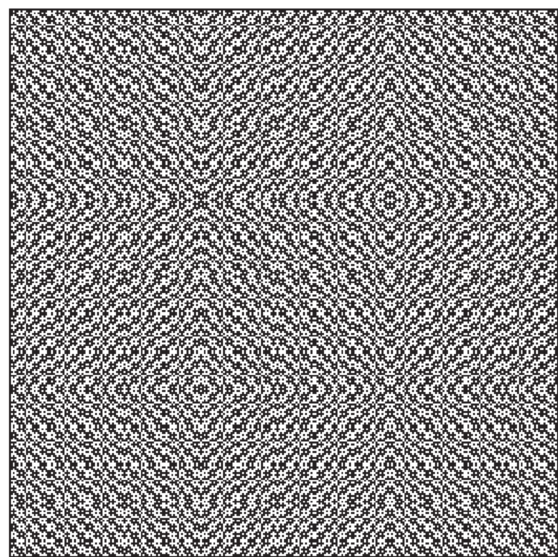
U := [1,2,3,2]

V := [1,3, 5, 4, 2]

a draft based on the resulting sequence is:



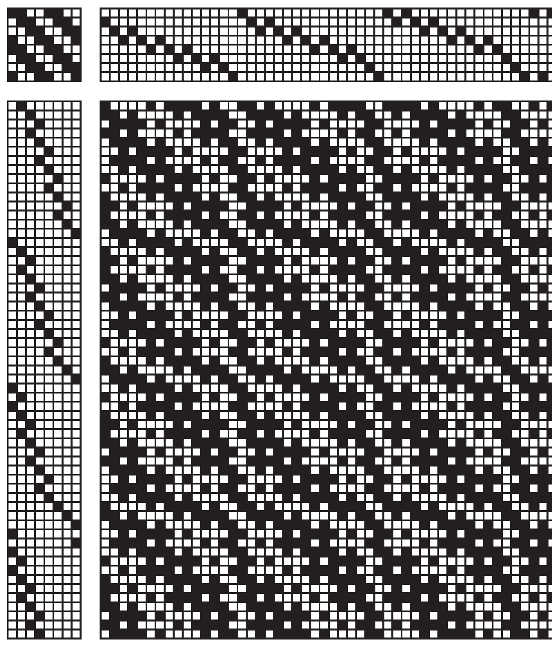
Here is the weave pattern:



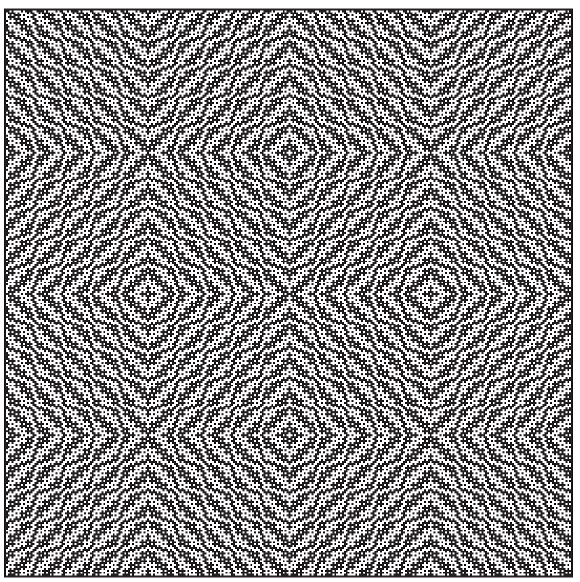
On the other hand, given the values

U := [1,2,3]
V := [1,2,3,4,5]

a draft based on the resulting sequence is:



Here is the weave pattern:



Example 2

seed: S

rules: S ➛ pal(T)
T ➛ coll(U,V)
U ➛ pal(X)
V ➛ pal(Y)

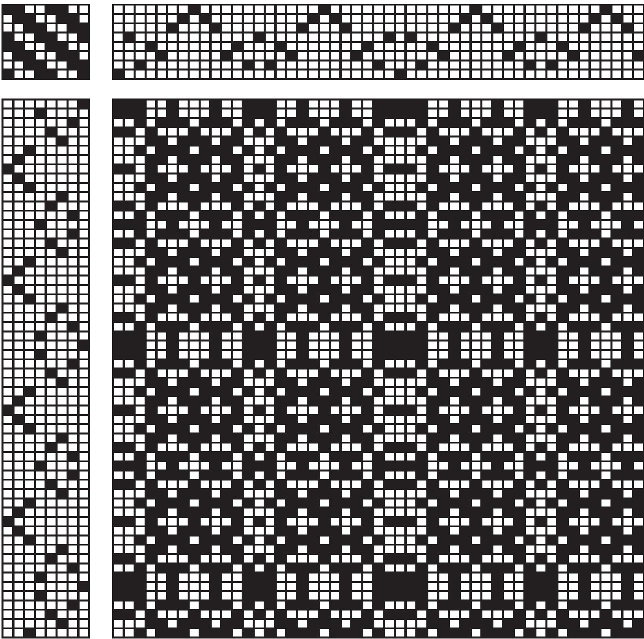
The terminal generation is

pal(coll(pal(X),pal(Y)))

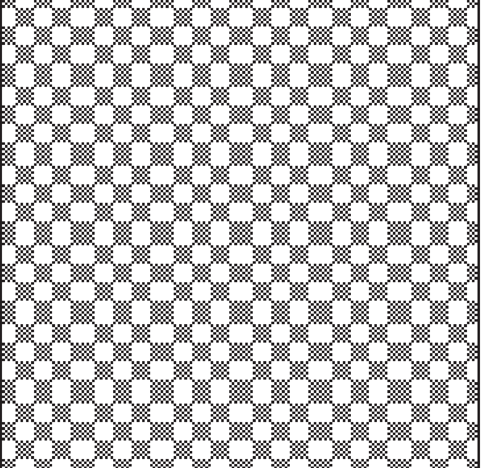
Given the values

X := [1,3,5,7,9,8,6,4,2]
Y := [6,4,2,7,5,3]

a draft based on the resulting sequence is:



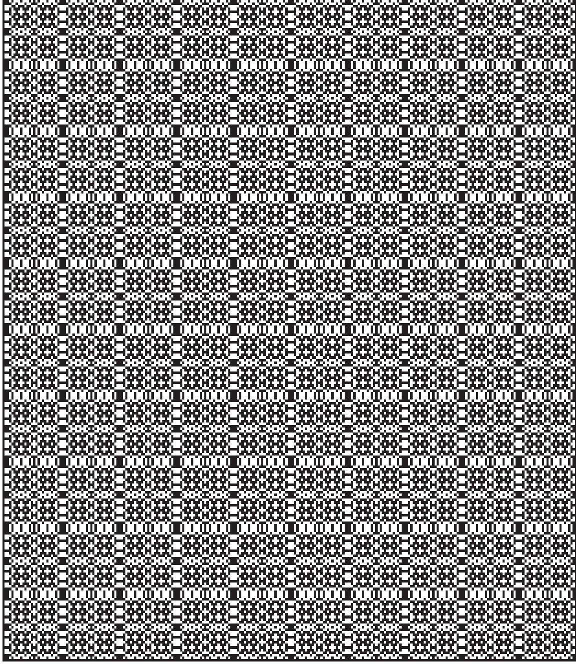
Here is the weave pattern:



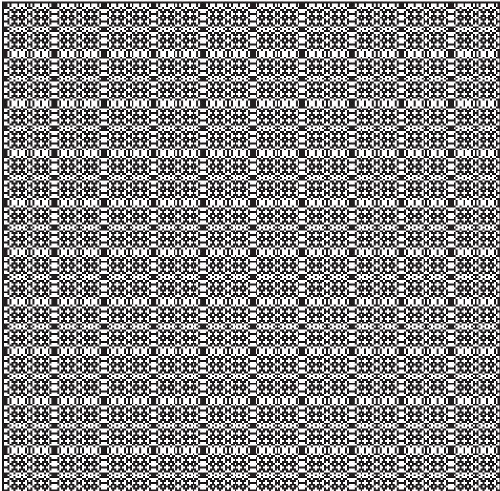
On the other hand, given the values

X := [1,5,2,4,3,6,7,8]
Y := [6,4,2,7,5,3]

a draft based on the resulting sequence is:



Here is the weave pattern:



Of course, just for these two example L-Systems, there is an infinite number of sequences, not to mention how they are used in drafts.

Non-Terminal T-Sequence Generation

The examples of L-Systems for generating t-sequence expressions in previous sections all have been terminal. While terminal L-Systems provide useful models, they do not exploit the power of L-Systems to generate successively more complex and detailed patterns — in this case, t-sequence expressions.

Consider this L-System:

seed: S

rules: S ➛ pal(T)
T ➛ motif(U,V)
U ➛ hor(T)

It generates t-sequences expressions that are palindromes containing motifs along paths with horizontal reflection. But since T and U are defined in terms of themselves, exactly what is going on is hardly clear. The first few generations are:

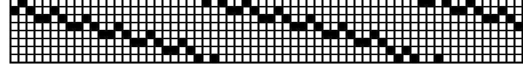
pal(T)
pal(motif(U,V))
pal(motif(hor(T),V))
pal(motif(hor(motif(U,V)),V))
…

The first three of these generations are comprehensible, but the last one is too intricate to comprehend; it is necessary to try examples and see what results.

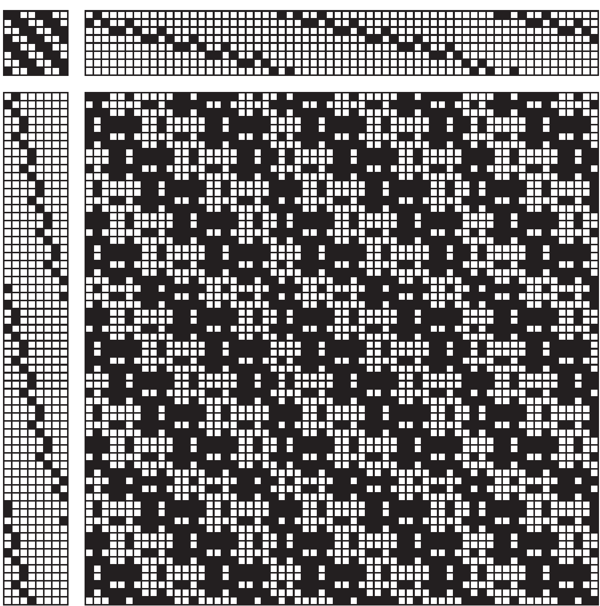
Suppose U and V have the values

U := [1,2,3,2]
V := [1,2,3,4,5,6]

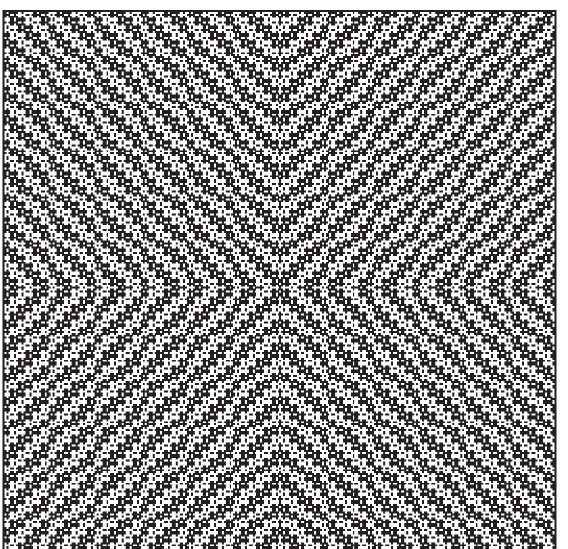
The resulting sequence starts like this:



A partial draft based on this sequence is:



and the weave is:



Here is another example of a non-terminating L-System:

seed: S

rules: S ➛ pal(T)

T ➛ coll(U,V)

U ➛ pal(S)
V ➛ ver(T)

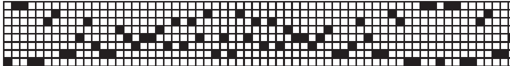
The first few generations are:

pal(T)
pal(coll(U,V))
pal(coll(pal(S),vert(T)))
pal(coll(pal(pal(T)),vert(coll(U,V))))
…

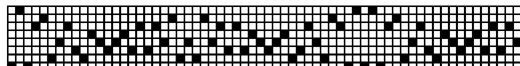
Suppose T, U, and V have the values

T := [1,2,3,4,5,6,7,8]
U := [1,1,2,2,3,3,4,4,5,5]
V := [8,7,6,5,4,3,2,1]

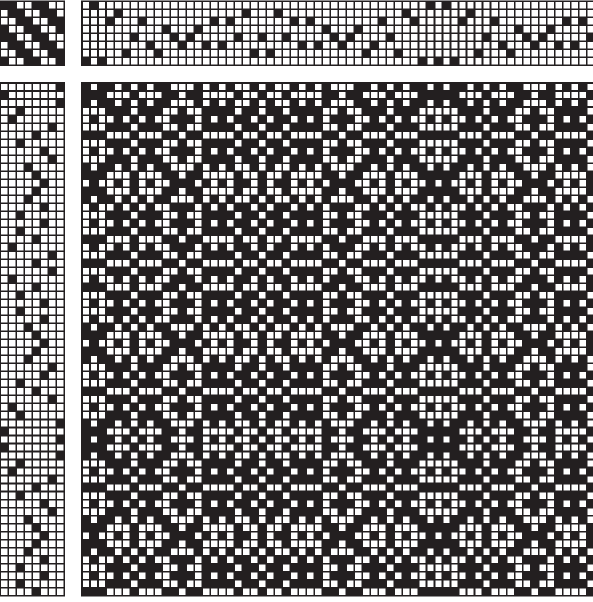
The resulting sequence starts like this:



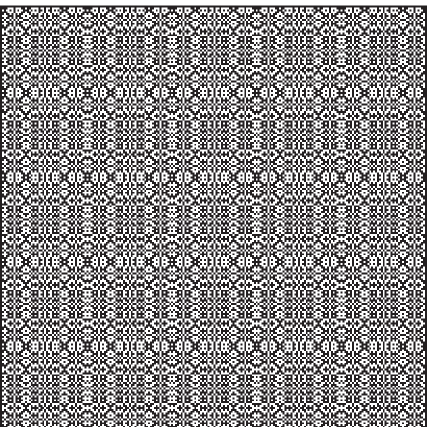
Here is a case of a sequence with adjacent duplicate values. Duplicates are not surprising, since the expression from which it was created is not comprehensible and hence the results unpredictable. Removing adjacent duplicates produces this:



A partial draft based on this sequence is:



and the weave is:



Of course other values for the variables give different results, sometimes very different results.

There are four problems with using L-Systems in the manner described above:

- It is difficult to design useful L-Systems.
- It is difficult to predict the results.
- It is difficult to assign useful values to undefined variables.

• The sequences produced quickly become impossibly long.

The last problem is the easiest to handle: Simply truncate long sequences so they are of a manageable length. Note the t-sequence expressions cannot be truncated; trying to do so generally results in invalid expressions.

The other three problems are essentially intractable if any degree of complexity it to be obtained. The best approach to problems like this is to try many alternatives, extract the useful results, and learn from the process.

For this to be practical, it is necesary to use computer programs.

L-System Design

Creating an L-System for a particular purpose is neither easy nor intuitive, but, when successful, the results can be more than worth the effort. And as is the case with many such things, the process becomes easier with practice and experience.

When designing L-Systems, it is important to keep in mind their basic properties and their inherent problems.

The Fractal Nature of L-Systems

L-Systems fundamentally are fractal generators. Although it is possible to design L-Systems that produce simple, easily understood patterns, the L-System mechanism by its nature is fractal.

This fractal nature comes from three sources:

- parallel operation on characters (global uniformity)
- characters defined in terms of themselves (recursion)
- repeated application of the rules (iteration)

The ways characters can be defined in terms of themselves is of central importance. A simple example is

seed: A
rules: A ➛ ABA
B ➛ BBA

Here both A and B are defined in terms of themselves and each other. Repeated applications of the rules produces increasingly long and intricate combinations of the two characters:

A
ABA
ABABBAABA
ABABBAABABBABBAABAABABBAABA
…

Although the rules are simple, the patterns that develop are nonetheless complex and not easy to characterize.

The Seed

The seed, with which generation begins, is not particularly important. In

most L-Systems, the seed is a single character. The seed can be a string of characters, but an L-System with such a seed can always be replaced by an L-System whose seed is a single character.

Consider this example:

seed: ABCBA
rules: A ➛ BC
B ➛ AB
C ➛ CB

A new character can be added as the seed and a new rule can be added replacing it by the original seed:

seed: D
rules: D ➛ ABCBA
A ➛ BC
B ➛ AB
C ➛ CB

The only difference between these two L-Systems is an additional generation in the second. Note that D only appears once.

Alphabet

The alphabet of the characters used in an L-System really only matters as to the number of characters. Characters are arbitrary. They may be chosen for mnemonic value, but until the interpretation of a string generated by an L-System, they have no meaning.

For example,

seed: A
rules: A ➛ ABA
B ➛ BBA

and

seed: 3
rules: 3 ➛ 3X3
X ➛ XX3

are equivalent.

Generation Length

An inherent property of L-Systems is increase in length of successive generations. In fact, this limited early work on L-Systems at a time when the amount computer memory was very available was very small.

It is possible to design L-Systems in which generation length does not increase. An example is

seed: A
rules: A ➛ B
B ➛ A

which generates

A
B
A
B
…

Such L-Systems are both contrived and trivial.

When a rule specifies replacement by more than one character, generation length increases. This problem is addressed in a subsequent section.

Character Relationships

The way that characters are defined in terms of themselves and each other has many effects on L-System generation.

If not all characters appear in all rules, there many be successive generations that have essentially different characteristics.

A simple and trivial example is

seed: A
rules: A ➛ BB
B ➛ CC
C ➛ AA

for which the generations are

A
BB
CCCC
AAAAAAAA

BBBBBBBBBBBBBBBBBBB

...

A subsequent section addresses the issue of character interaction in more detail.

What is Possible

L-Systems are only one of many kinds of formal grammars [11]. Different kinds of grammars have different "expressive power". The issue of expressive power is of both theoretical and practical importance.

Expressive power, roughly speaking, is a measure of what kinds of patterns a formal grammar can produce. Expressive power is measured more in terms of what patterns can be excluded than what may be included.

For example, almost all kinds of formal grammars can produce palindromes, but they inevitably produce other patterns as well. That is, non-palindromes cannot be excluded by grammars of most kinds. L-Systems can generate purely palindromic sentences and in this sense are more powerful than most other kinds of formal languages.

Interpretation

Although interpretation falls outside the scope of L-Systems proper, it is a power design tool and its possible use needs to be kept in mind when L-Systems are designed.

An L-System intended to produce a profile draft may not require any interpretation other than the think of the characters as blocks. On the other hand, an L-System designed to draw a pattern may require interpretation of characters as navigation and drawing actions [1].

But interpretation can be used to change L-System strings in arbitrary ways, including reordering them, deleting characters, and so forth. In some sense, there is no limit to the power of interpretation.

Cellular Automata

A cellular automaton is an array of identical, interacting cells, as shown in Figure Ω.1

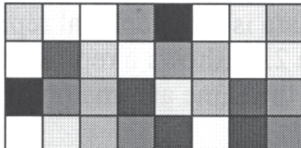


Figure 1. Cellular Automaton

The cells in cellular automata have states, indicated in Figure Ω.1 by different colors. We'll confine our attention to cellular automata in which the cells have only two states, 1 and 0, indicated by black and white respectively. Figure Ω.2 shows an example.

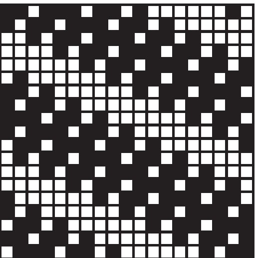


Figure Ω.2. Two-Color Cellular Automaton

Notice the resemblance in appearance of two-color cellular automata to drawdowns. In fact, that's the role of cellular automata here.

A cellular automaton, as a whole, passes through a succession of configurations corresponding to the states of its cells. The automaton goes from one configuration to another at discrete intervals of time, the states of all its cells changing in parallel. The change of state of a cell is determined by a transition rule that depends on the neighbors of the cell and is the same for all cells in the automaton.

The neighborhood of a cell can be defined in different ways. Figure Ω.3 shows one of the most frequently used neighborhoods, which is named after John von Neumann, who used it in his studies of self-reproducing machines. See the side bar on the next page.

Cellular Automata Applications

John von Neumann
1903-1957

John von Neumann, who played a major role in the design of modern computers, was among the first to use cellular automata as models for abstract machines.

He proved that it is possible, in principle, to design machines that not only are capable of reproduction but also of evolving into more complicated machines.

Cellular automata are widely used as discrete models of physical systems and have been used to simulate a wide range of natural processes such as turbulent fluid flow, gas diffusion, forest fires, and avalanches. Cellular automata can even be used to generate pseudo-random numbers.

Considered abstractly, cellular automata exhibit a wide variety of behaviors: self organization, chaos, pattern formation, and fractals.

John Conway's *Game of Life* [?] is the best known abstract application of cellular automata. In it, a wide variety of patterns with life-like properties are born, interact, and die in fascinating and complex ways. Vast amounts of human and computer time have been expended exploring this strange world.

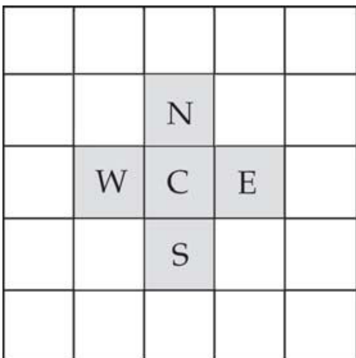


Figure Ω.3. Von Neumann 5-Neighborhood

The cell itself is labeled C. Its four neighbors are labeled according to their relative positions according to the points of the compass.

Figure Ω.4 shows another commonly used neighborhood, named after Edward F. Moore, an early pioneer in studies of cellular automata.

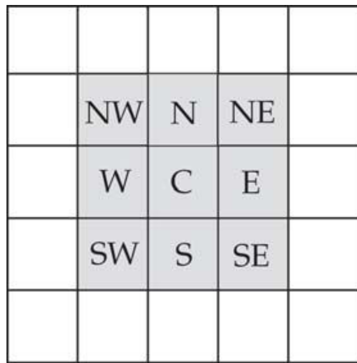


Figure Ω.4. Moore 9-Neighborhood

Subscripts are used to denote times, which proceed 1, 2, 3, … . For example C_{10} is the state of C at time 10.

A typical transition rule is the "parity rule" for the 5-neighborhood:

$$C_{i+1} = (C_i + N_i + E_i + S_i + W_i) \bmod 2$$

That is, $C_{i+1} = 1$ if the sum of the neighborhood states (including C itself) is odd and 0 otherwise.

Another interesting rule is the "voter rule" for the 5-neighborhood:

$$C_{i+1} = 1 \text{ if } (N_i + E_i + S_i + W_i) > 2$$
$$C_{i+1} = 0 \text{ if } (N_i + E_i + S_i + W_i) < 2$$
$$C_{i+1} = \sim C_i \text{ otherwise}$$

where ~C is the complement of C: 1 if C = 0, 0 if C = 1.

Note that in the voter rule, the result may depend on the value of C, while in the parity rule, it does not: In the parity rule, C is treated no differently than its neighbors.

Cellular Automata Topology

There is a sticky issue: What happens to the cells at the edge of an

automaton? What are their neighbors?

This problem can be dealt with in several ways. The way chosen depends on the context in which the cellular automaton is considered.

One way is to consider the cellular automaton to be infinite without edges, with cells extending off indefinitely in all four directions. Another way is to treat the cells at the edges as unchanging, serving as a kind of static border.

A less obvious but natural and useful way in the context of drawdowns is to consider the cellular automaton to wrap around from edge to edge. See Figure Ω.5.

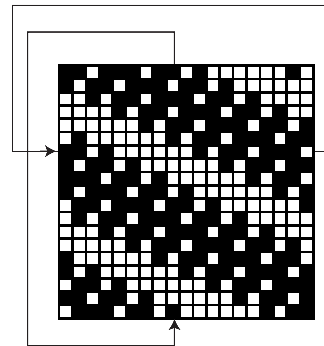


Figure Ω.5. Neighborhood Wrap-Around

Thus, the N neighbor of a cell on the top edge is the cell in the corresponding row on the bottom edge, and so on.

From a topological point of view, this constitutes wrap-around of the horizontal and vertical edges and also of the top and bottom edges. The result is a three-dimensional surface known as a torus, as suggested by Figure Ω.6.

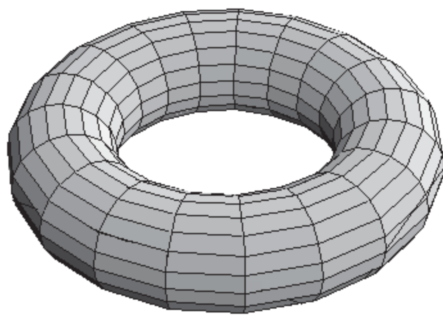


Figure Ω.6. Torus

The cells on this torus are distorted because the "horizontal" circumference is larger than the "vertical" circumference so that the general shape to be seen

more easily. Perspective causes the shapes of the cells to be skewed.

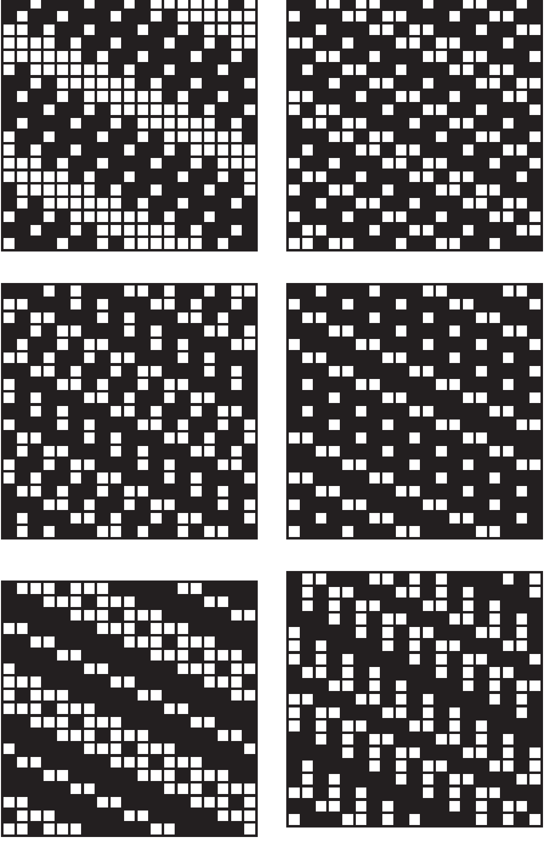
It is not necessary to actually make a toroidal cellular automaton. It is only necessary, when applying rules, to determine the neighbors according to the wraparound topology.

It is worth noting that edge wrap-around is equivalent to an infinite plane of repeats.

Pattern Sequences

When a cellular automaton is started in a specific configuration and a rule is applied repeatedly, a pattern sequence results.

Figure Ω.7 shows the beginning of the pattern sequence that results from applying the 5-neighborhood parity rule to the pattern shown in Figure Ω.2. The complete sequence has 511 distinct patterns; at the next iteration, the original pattern reappears; after this, there are no new patterns.



• • •

Figure Ω.7. Parity Rule Sequence

Figure Ω.8 shows the pattern sequence that results from applying the voter rule to the pattern shown in Figure 2. In this case, there are only three distinct patterns; the fourth is the same as the second.

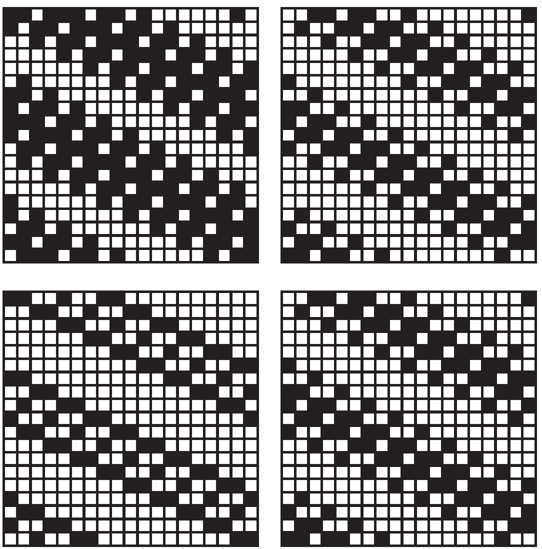
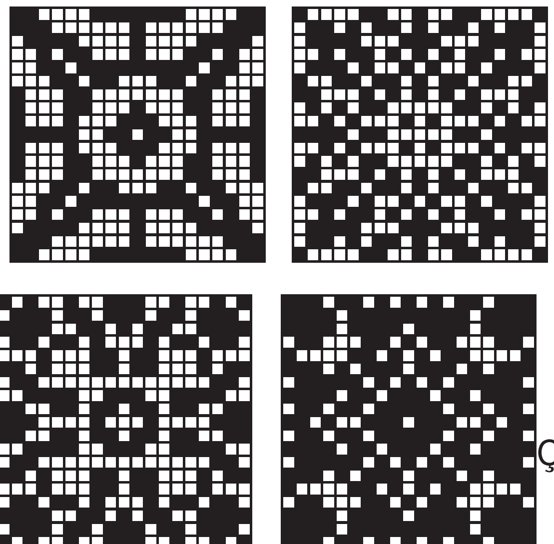
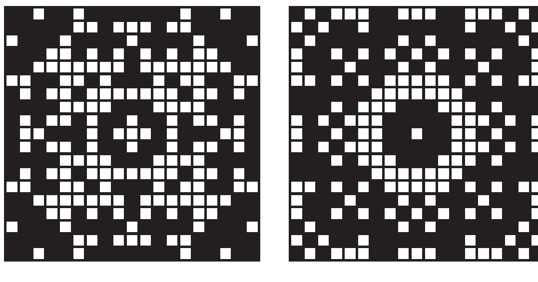


Figure Ω.8. Voter Rule Example

Figure Ω9 shows the beginning of the pattern sequence for the 5-neighborhood parity rule starting with a symmetric pattern. There are 511 distinct patterns in all, the 512th being the same as the first.



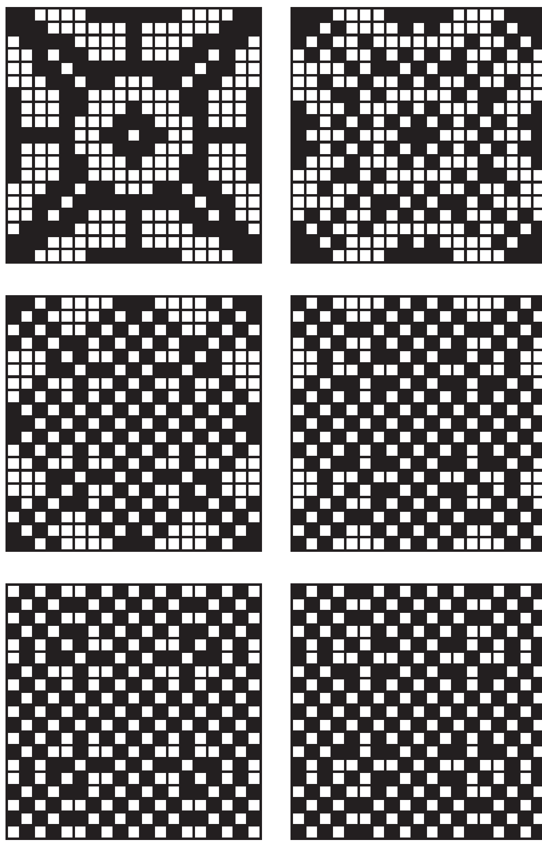
Ç



• • •

Figure Ω.9. Parity Rule with Symmetric Pattern

The voter rule, as in the previous example, yields fewer distinct patterns starting with this initial pattern, the seventh being the same as the first. See Figure Ω.10.



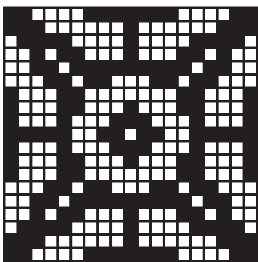
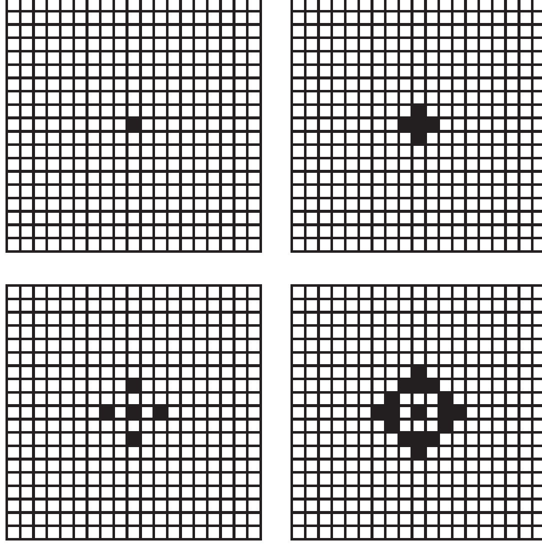


Figure 10. Voter Rule with Symmetric Pattern

An interesting way to explore the effects of a rule is to start with a "seed", a single black cell in a field of white ones.

In such pattern sequences, it usually takes some time for the seed to spread results to a sufficient extent that useful patterns result. Figure Ω.11 shows the pattern sequence for a single seed and the 5-neighborhood parity rule. There are 511 different patterns in all. The first eight are shown in this Figure. Figure Ω.12 shows four of the more interesting patterns from the first 64.



...

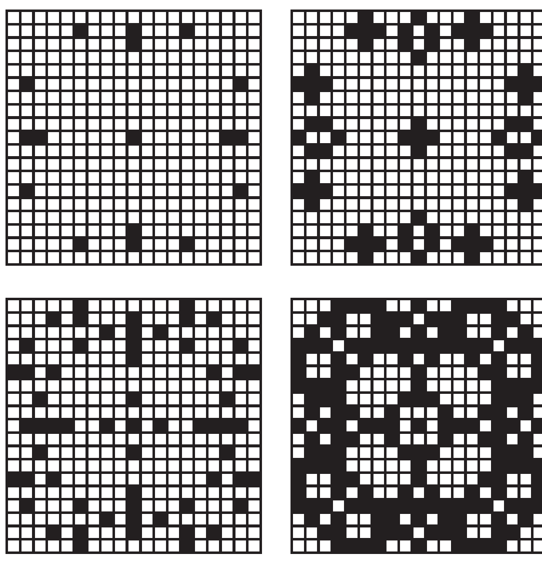


Figure Ω1.1. Parity Pattern Sequence Start-up

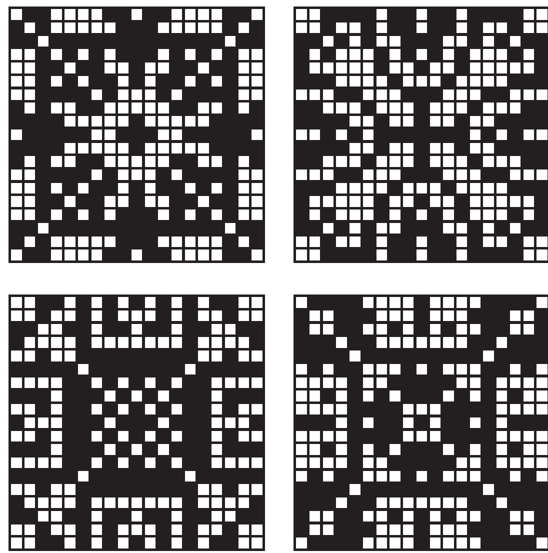
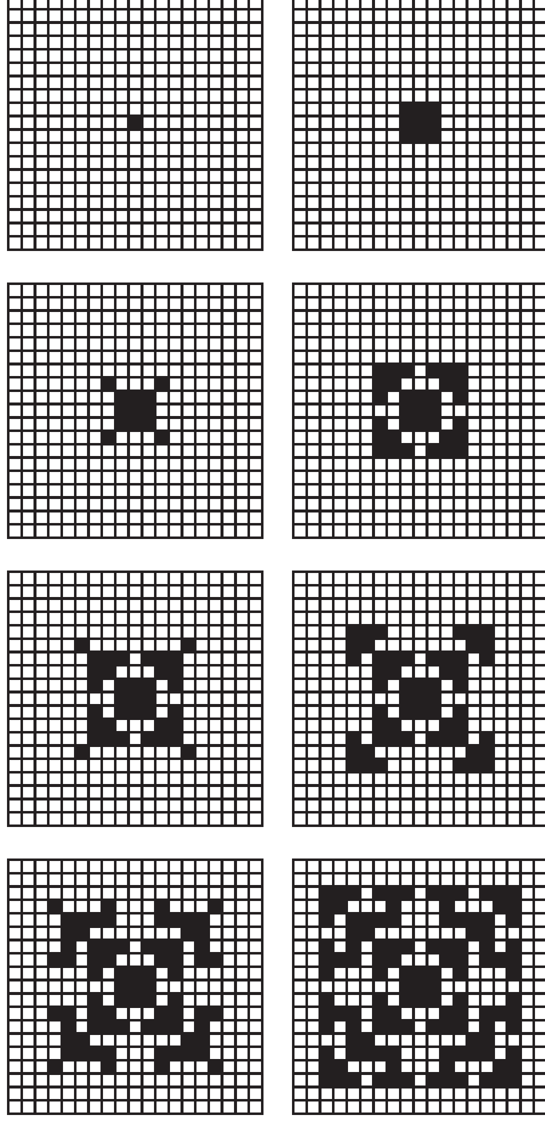


Figure Ω.12. Selections from First 64

An apparently uninteresting 9-neighborhood rule, called "1-of-8", is

$$C_{i+1} = 1 \text{ if } (NW_i + N_i + NE_i + E_i + SE_i + S_i + SW_i + S_i) = 1$$
$$C_{i+1} = C_i \text{ otherwise}$$

This rule, starting with a single seed, produces a fascination fractal pattern. See Figure Ω.13.



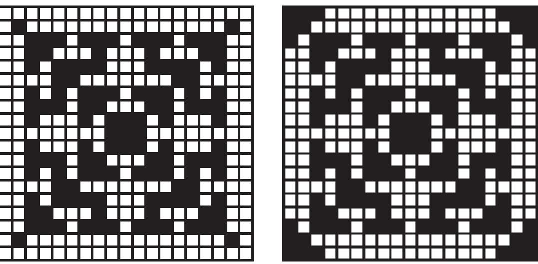
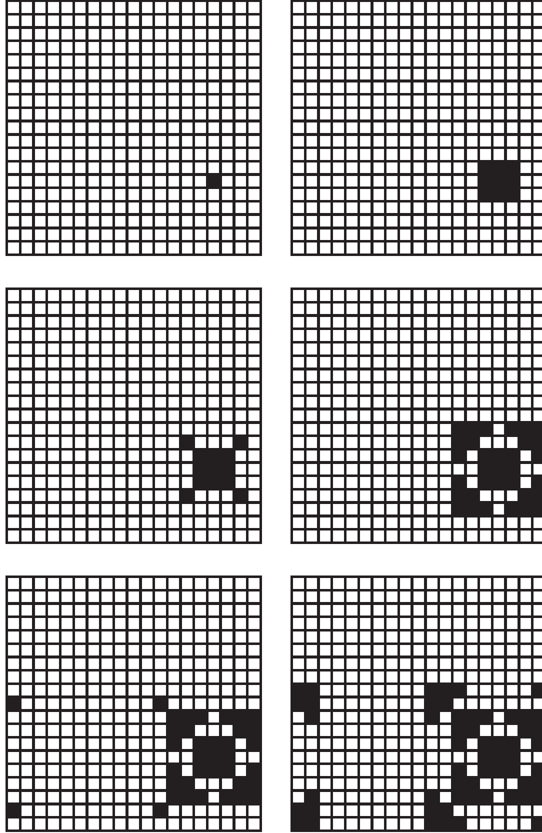


Figure Ω.13. 1-of-8 Rule Fractal Pattern

All patterns after the 10th are the same as the 10th.

Putting the seed off center illustrates the effect of wraparound topology. See Figure Ω.14.



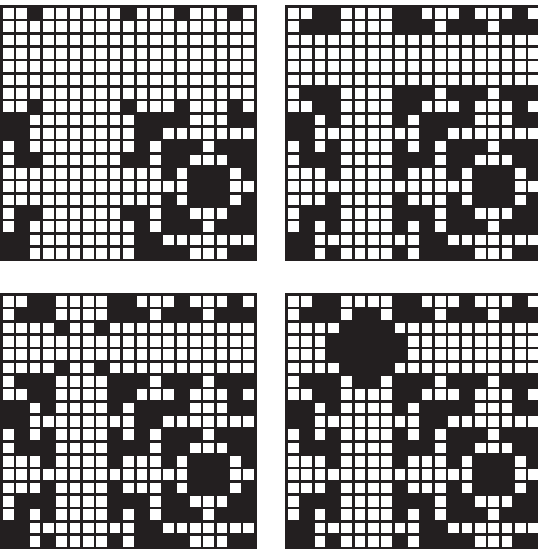


Figure Ω.14. Offset and Wrap-Around

The patterns in Figure Ω.14 are the same as those in Figure Ω.13; they are just at different positions on the torus.

Structural and Aesthetic Concerns

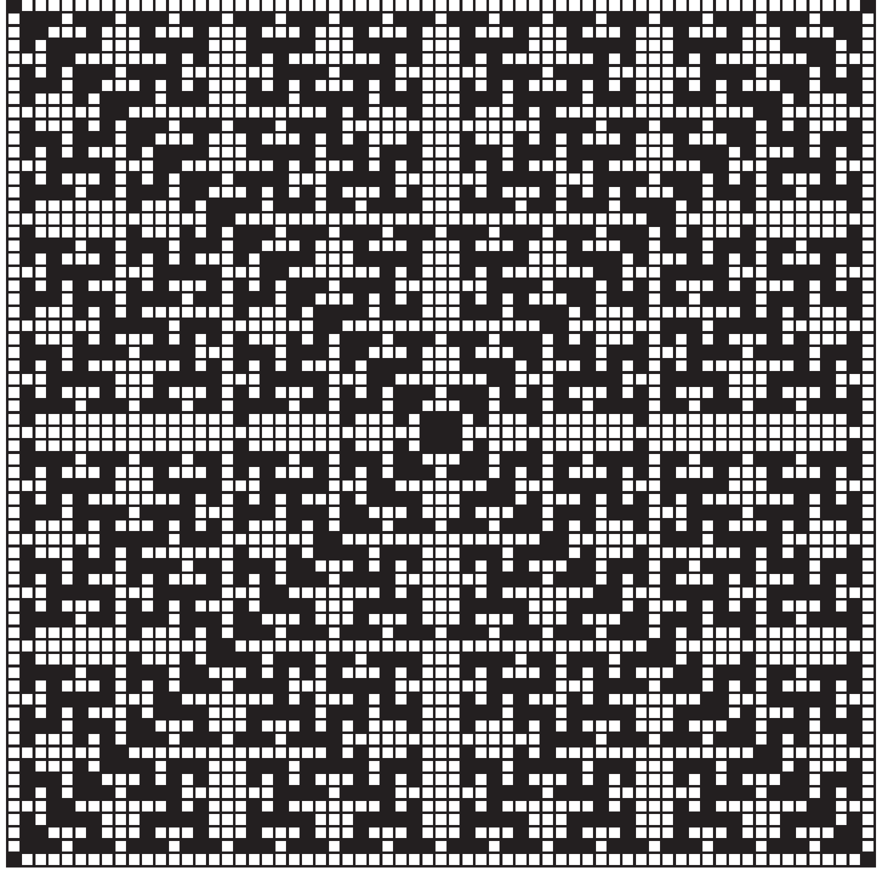
Many patterns produced by cellular automata are unsuitable for weaving for structural reasons. Notable examples are the initial patterns in sequences starting with a single seed. Other patterns simply are unattractive.

Cellular automata can produce thousands of patterns quickly. Even with the rejection of obviously unsuitable patterns, the problem is one of excess. How can really good patterns be found in seas of possibilities?

One approach is to start with a conventional drawdown pattern such as the one shown in Figure Ω.2 and look for interesting examples "of type".

Another approach is to start with an attractive and structurally sound symmetric pattern and apply a symmetric rule (one, like the parity rule, in which the result does not depend on the actual positions of specific neighbors). This avoids the problem with an overwhelming cascade of chaotic patterns that may result by starting with a pattern without much structure and applying an asymmetric rule.

Size matters also. 19 × 19 patterns are used in this article for presentation purposes. Large patterns usually lead to longer pattern sequences and allow more interesting results, as illustrated by this large 1-of-8 pattern.



A T-Sequence Language

Introduction

[This section now assumes t-sequences have been defined and discussed in the early parts of the book. The section has been renamed to distinguish t-sequences from a language that describes them]

This section describes a language that can be used for constructing and manipulating t-sequences. The formalism that is introduced here allows t-sequences to be described precisely and compactly and provides conceptual focus. There is no mathematics, *per se*, just as these is no mathematics in the notation used for weaving drafts. Like draft notation, the t-sequence notation must be understood to be useful. The t-sequence language uses many "special" characters to describe operations concisely. Because of this, the t-sequence language may appear to be daunting. But the ideas are simple.

Terminology and Notational Conventions

The term *sequence* implies linear order. The *terms* in a sequence come one after another. There is a first term, a second term, and so on.

T-sequence terms may be explicit, as in 1, 4, 6, and so on, or they may be given as *variables* that take on different values in different contexts. Variables are indicated by lowercase italic letters, such as i, j, and k. Subscripts may be used to distinguish different term variables, such as i_1, i_2, j_5, and so on.

Sequences may be given explicitly by enclosing their terms in square brackets, as in

[1, 2, 3, 4, 3, 2, 3, 4, 5, 6, 7, 8, 7, 6, 5, 4]

$[i_1, i_2, i_3, i_4, i_5, i_6, i_7, i_8]$

It is also possible to have an empty sequence with no terms. Although an empty sequence is not useful in weaving, it may arise in operations used to create other t-sequences. The empty sequence is denoted explicitly by [] and is represented by the symbol Θ.

Ellipses are used to indicate one or more terms in a sequence that are not given explicitly, as in

[1, 2, 3, 4, 5, … 15, 16, 15, … 1]

Variables are used to name sequences so that they can be referred to without specifying their terms. Sequence variables are indicated by uppercase italic letters, such as S, T, and U. Sequence variables also may have subscripts to distinguish different sequences in a common context. Examples are S_1, S_2, and T_5.

A specific sequence can be given a name. This is called *assignment* and is indicated by a colon followed by an equal sign, as in

S := [1, 2, 3, 4, 3, 2, 3, 4, 5, 6, 7, 8]

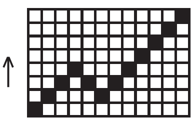
Then S can be used to refer to this sequence without giving all the terms.

Two t-sequences are identical, denoted by S = T, if they are the same, term by term.

Graphic Representation

Patterns in t-sequences usually are easier to detect in graphical representations than by examining sequences of integers.

In this section, the values in grid plots increase upward:



In the grid plots used in this section, the axes usually are not marked, since such markings tend to distract the human visual system and interfere with pattern recognition.

The bottom row corresponds to the value 1 and the left column corresponds to the first value in the sequence.

T-Sets and T-Numbers

Sometimes it is useful to specify the particular shafts/treadles used in a t-sequence. This is called the *t-set* of the t-sequence. Braces are used to denote t-sets, as in {1, 2}.

In many cases, all the shafts and treadles are used, as illustrated in the example above. For example, the t-set for the plot above is

{1, 2, 3, 4, 5, 6, 7, 8}.

Finally, to avoid having to say shaft/treadle numbers repeatedly, *t-numbers* is used to cover both.

Sequence Metrics

There are three important properties associated with a sequence: its length, its minimum value (usually 1), and its maximum value, called its *bound*. These are given by functions whose names are lowercase Greek letters:

$\lambda(S)$	length
$\mu(S)$	minimum value
$\gamma(S)$	maximum value (bound)

For the sequence S in the preceding section, $\lambda(S) = 12$, $\mu(S) = 1$, and $\gamma(S) = 8$.

Extension

Concatenation

The most fundamental operation on t-sequences is appending one to another to form a longer one. This is called *concatenation*.

Concatenation of t-sequences is denoted by

$S \mid T$

in which the result is a new sequence consisting of the terms of S followed by the terms of T.

For example if

$S = [1, 2, 3, 4, 5, 6, 5, 4, 3, 2]$

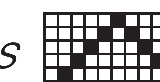
and

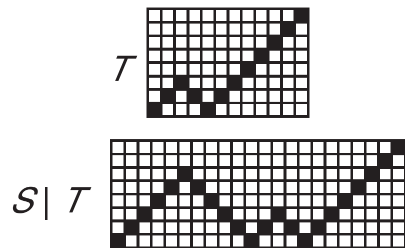
$T = [1, 2, 3, 2, 1, 2, 3, 4, 5, 6, 7, 8]$

then

$S \mid T = [1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 2, 3, 2,$
$1, 2, 3, 4, 5, 6, 7, 8]$

Here is the graphic representation:





The empty sequence Θ is the identity with respect to concatenation. That is,

$(S \mid \Theta) = (\Theta \mid S) = S$

for all S.

Often many t-sequences are concatenated, one after the other. To handle such cases conveniently, the notation

$\mid (S_1, S_2, \ldots, S_n)$

denotes the concatenation of $S_1, S_2, \ldots, S_n$.

Repetition

Repetition is one of the most common operations on t-sequences. Repetition consists of concatenating a sequence with itself, perhaps several times.

Repetition is denoted by

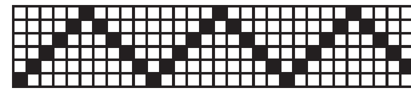
$S \times i$

where i, an integer ≥ 0, specifies the number of repetitions.

For example, if S is as given in the preceding section, then

$(S \times 3) = [1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 2, 3,$
$4, 5, 6, 5, 4, 3, 2, 1, 2, 3, 4, 5, 6, 5, 4, 3, 2]$

Here is what it looks like as a grid plot:



$(S \times 1) = S$ and $(S \times 0) = \Theta$, the empty sequence for all S.

Extension

It sometimes is desirable to repeat a sequence to a specific length that is not

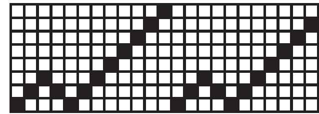
an even multiple of the length of the sequence.

This operation is called *extension* and is denoted by $S \Rightarrow i$, where $i \geq 0$ is the length of the new sequence.

For example, if T is as given previously,

$$(T \Rightarrow 23) = [1, 2, 3, 2, 1, 2, 3, 4, 5, 6, 7, 8, 1, 2, 3, 2, 1, 2, 3, 4, 5, 6, 7]$$

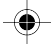
Here is what it looks like as a grid plot:



The extension length i may be less than $\lambda(S)$, in which case truncation at the right occurs. For example,

$$(T \Rightarrow 9) = [1, 2, 3, 4, 5, 6, 5, 4, 3]$$

Of course, $(S \Rightarrow 0) = \Theta$ for all S.



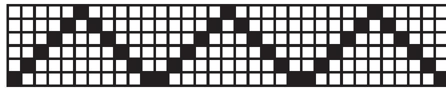
Duplicate Terms

Although concatenation and its two specialized forms, repetition and extension, are simple and fundamental operations, problems may arise if the last term in a sequence is the same as the first term in the sequence appended to it. Such duplicate terms may appear as undesirable artifacts of the concatenation and in some weaving contexts may cause structural problems.

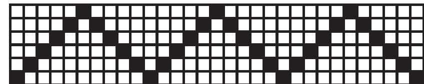
For example, if

$$S = [1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1]$$

and duplicates at the boundaries of concatenation are not removed, $S \times 3$ would be as shown as:



If duplicate terms at the boundaries of concatenation are removed, however, the result is as shown here:



30 Jul 2006 J7: T-Sequences

Whether or not duplicates that result from concatenation should be removed is a matter of context and not a property of the sequences involved. More often than not, duplicate removal is desired, so the operations of concatenation, repetition, and extension do that.

There are alternative versions of these operations that do not remove duplicates. These are denoted by $S \mid_+ T$, $S \times_+ i$, and $S \Rightarrow_+ i$. For example, for the sequence S given above,

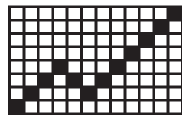
$$(S \times_+ 3) = [1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1, 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1]$$

Note: Any duplicates *within* a sequence are unaffected by any of the concatenation operations.

Runs

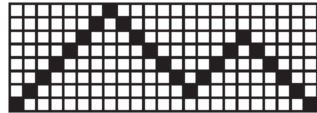
Runs — integers in numerical sequence — occur very frequently in t-sequences.

A *simple run* consists of integers in order from a starting value to an ending value. If the starting value is less than the ending value, the run is up, else it is down. Here is an up run followed by a down run.

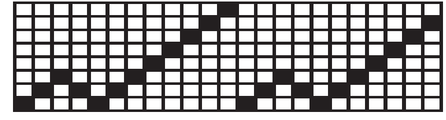


Simple Runs

There are two different kinds of runs that are composed of simple runs: *connected runs* and *disconnected runs*.



Connected Run



Disconnected Runs

Simple Runs

A simple run is denoted by

$i \rightarrow j$

For example,

$(2 \rightarrow 8) = [2, 3, 4, 5, 6, 7, 8]$

and

$(5 \rightarrow 1) = [5, 4, 3, 2, 1]$

Connected Runs

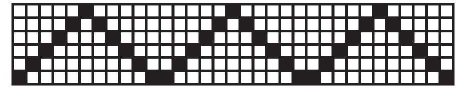
In a connected run, runs go up and down (or down and up) between a *beginning point*, *inflection points*, and an *ending point* with no gaps. Beginning points, inflection points, and ending points collectively are called *anchor points*.

Connected runs can be constructed using simple runs and concatenation (with duplicate removal [1]). For example, the connected run shown in Figure 2 can be constructed by

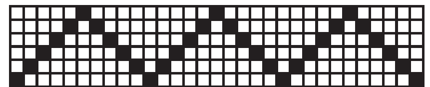
$(1 \rightarrow 8) \mid (8 \rightarrow 2) \mid (2 \rightarrow 6) \mid (6 \rightarrow 1)$

where parentheses are used to make the grouping of operations unambiguous.

Constructing a connected run using concatenation is unnecessarily cumbersome, however, since the run is completely described by its anchor points: the sequence [1, 8, 2, 6, 1]. This is emphasized in the figure below, where the anchor points are set off by color. The sequence of anchor points follows.



Highlighted Anchor Points



Anchor-Point Sequence

The operation for constructing a connected run from an anchor-point sequence *S* is denoted by → *S*. Note that the operator symbol is in prefix position before its operand as opposed to the same symbol used to denote simple runs, which is in infix position between its operands.

For example,

→ [1, 8, 2, 6, 1]

produces the same connected run as the concatenation of simple runs shown earlier.

Disconnected Runs

In a disconnected run, there are breaks in the numerical sequence of values.

Disconnected runs can, of course, be constructed by concatenating simple runs. For example, the disconnected run shown in Figure 3 can be constructed by

(1 → 5) | (2 → 6) | (3 → 7) | (4 → 8) |
(6 → 3) | (5 → 2) | (4 → 1)

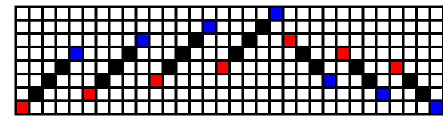
This also is unnecessarily cumbersome, since the runs are completely characterized by pairs of beginning and ending points: the sequences

[1, 2, 3, 4, 6, 5, 4]

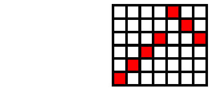
and

[5, 6, 7, 8, 3, 2, 1]

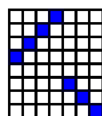
This picture shows the disconnected runs with the end points highlighted. Following two pictures show the sequences of beginning and end points.



Highlighted End Points



Beginning-Point Sequence



End-Point Sequence

The operation for constructing disconnected runs from sequences of end points is denoted by S / T, where S is the sequence of beginning points and T is the sequence of ending points.

For example, the disconnected runs shown in Figure 6 can be constructed by

$$[1, 2, 3, 4, 6, 5, 4] \;/\; [5, 6, 7, 8, 3, 2, 1]$$

It is worth noting that the sequences of beginning and ending points are concatenations of simple runs, so the same result can be obtained by

$$((1 \to 4) \mid (6 \to 4)) \;/\; ((5 \to 8) \mid (3 \to 1))$$

Although this form is more complicated than the one using explicit sequences, it reveals underlying structure in this disconnected run. Another form is perhaps more revealing:

$$((1 \to 4) \;/\; (5 \to 8)) \mid ((6 \to 4) \;/\; (3 \to 1))$$

This is the concatenation of a sequence of upward disconnected runs with a sequence of downward connected runs. This is, of course, evident in the grid plot.

Symmetries

Symmetry is one of the most powerful tools for producing aesthetically pleasing patterns. In t-sequences, the main use of symmetry is in concatenating a sequence and its reversal to produce a palindrome. Geometrically, reversal is horizontal reflection.

Horizontal Reflection

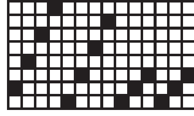
Horizontal reflection reverses the order of the terms in a sequence left to right. Horizontal reflection is denoted by $\leftrightarrow S$. For example, if

30 Jul 2006 J7: T-Sequences

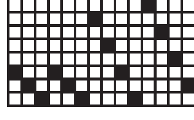
S = [2, 4, 6, 8, 1, 3, 5, 7, 1, 2, 3, 1, 2, 3]

then

$\leftrightarrow S$ = [3, 2, 1, 3, 2, 1, 7, 5, 3, 1, 8, 6, 4, 2]



S



$\leftrightarrow S$

Vertical Reflection

It is also possible to reflect a sequence vertically by reversing the *values*, so that the largest becomes 1, the next-to-largest becomes 2, and so on. If this operation is denoted by $\nu(i)$, then

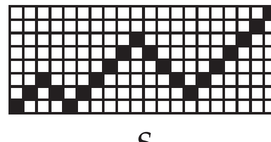
$$\nu(i) = \gamma(i) - i + 1$$

The operation of vertical reflection is denoted by $\updownarrow S$. For example, if

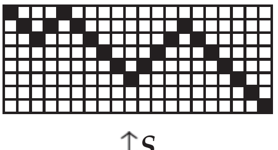
$S = \rightarrow$[1, 3, 1, 6, 2, 8]

then

$\updownarrow S = \rightarrow$[8, 6, 8, 3, 7, 1]



S



↕S

Palindromes

A palindrome is a sequence that is the same forwards and backwards. A palindrome is created by concatenating a sequence with its horizontal reflection (reversal):

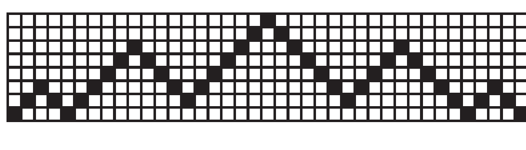
S | ↔S

This operation is so important that it has its own notation: ∩S. For example, if

S = →[1, 3, 1, 6, 2, 8]

then

∩S = →[1, 3, 1, 6, 2, 8, 2, 6, 1, 3, 1]



∩S

Note that the duplicate value at the middle is removed, as it is with concatenation. In the case that duplicate removal is not desired,

S $|_+$ ↔S

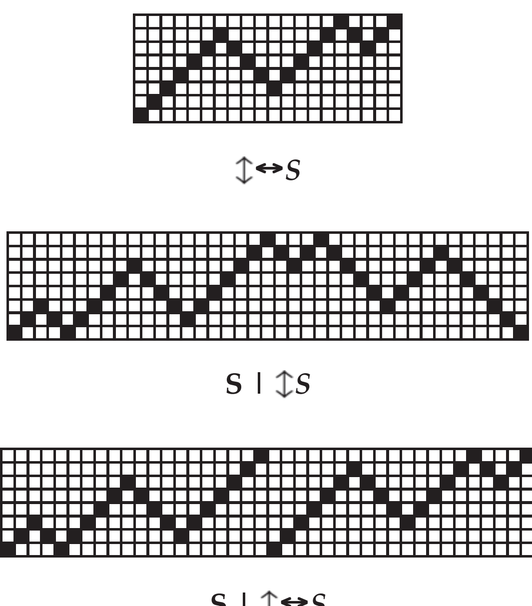
can be used.

"Palinforms"

The coined the word "palinforms" refers to concatenations of a sequence with one of its reflections other than the horizontal one.

There are two reflections other than horizontal that can be used to create palinforms: vertical and combined horizontal and vertical. Consider

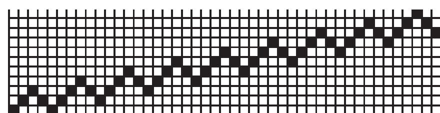
S = →[1, 3, 1, 6, 2, 8]



Although these palinforms do not have the obvious symmetry of palindromes, the inherent relationships produce visual interest, if of a more subtle form.

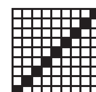
Motifs along Paths

Some of the most interesting patterns in t-sequences come from placing a (usually) short sequence, called a *motif*, at successive points along a path. Here is an example:



The motif is

and the path is a straight draw:



.

The operation of placing a motif M along a path P is denoted by

$M @ P$

Placing a motif along a path is concatenation with an offset. Adjacent duplicates may arise, and as for other forms of concatenation [1], duplicate values at boundaries are removed by default. The operation

$M @_+ P$

does not remove duplicates that arise at boundaries.

It is worth noting that if the path is a constant sequence (all terms the same), a motif along the path simply is a repeat . In other words, the concept of a

Summary

$S \mid T$	concatenation
$\mid(S_1, S_2, \ldots, S_n)$	concatenation
$S \times i$	repetition
$S \Rightarrow i$	extension
$i \rightarrow j$	simple run
$\rightarrow S$	connected run
$S \,/\, T$	disconnected runs
$\leftrightarrow S$	horizontal reflection
$\updownarrow S$	vertical reflection
$\cap S$	palindrome formation
$M @ P$	motif along a path

Concatenation operations without duplicate removal:

$S \mid_+ T$	concatenation
$\mid_+(S_1, S_2, \ldots, S_n)$	concatenation
$S \times_+ i$	repetition
$S \Rightarrow_+ i$	extension
$M @_+ P$	motif along a path

Introduction

There are numerous freeware, shareware, and commercial weaving programs. While the vary somewhat in the features they offer, they all are based on a common model in which the user can create and modify drafts in conventional ways.

This chapter describes some programs that use ideas contained in this book.

The first program, **Painter's Weaving Language**, is hidden inside a commercial software package designed for artists. It is was the initial inspiration that led to the explorations in this book.

The second program uses Boolean algebra as the basis of design. The third program focuses on designing draftable color patterns.

The final program is one based on the ability to evaluate functions dynamically; it is a "programmer's program", not one for the general user — but the one used by the author of this book for developing and testing most of the ideas in it.

Look at these programs for ideas — and for unconventional approaches to weave design.

The Painter Weaving Language

Corel's Painter application provides facilities for creating images using tools that mimic natural media.

One of Painter's facilities is a "weaving engine" for an eight-shaft, eight treadle loom. On the surface, the weaving engine offers a variety of built-in weaves that can be displayed in various ways — basically a way to produce patterns. Behind the scenes is an "advanced weaving language" [1]. The user can compose and edit expressions that describe the threading, treadling, and warp and weft color sequences. The tie-up is handled in the conventional manner. Figure Ω.1 shows an example of the weaving dialog for a shadow weave and Figure Ω.2 shows part of the corresponding image.

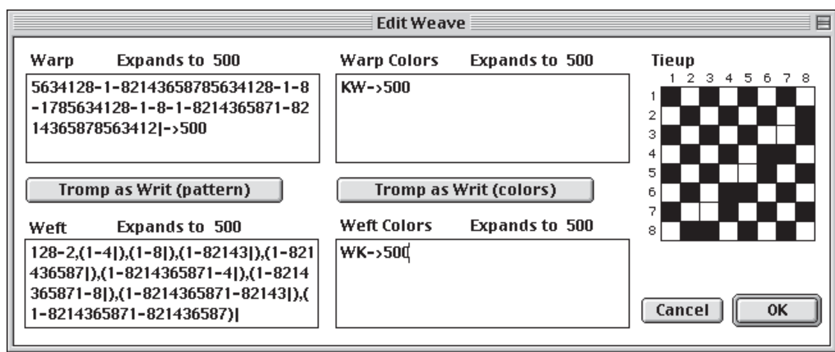


Figure Ω.1. Dialog for a Shadow Weave

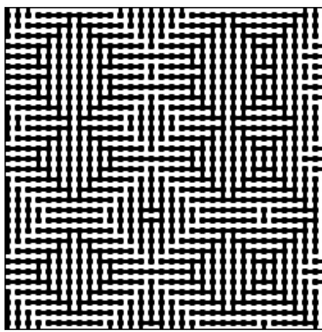


Figure Ω.2. Image of a Shadow Weave

The weaving language consists of expressions that produce sequences of characters. In the case of the threading and treadling, sequences are composed

of digits 1 though 8, which identify the shafts and treadles, respectively.

For example, in threading the shafts, the sequence 1346 means the first warp thread goes through shaft 1, the second through shaft 3, the third through shaft 4 and the fourth through shaft 6. In treadling, 1346 means treadle 1 is pressed for the first weft thread, treadle 3 for the second weft thread, 4 for the third, and 6 for the fourth. Full sequences are, of course, much longer than this.

Colors are represented by letters, with the actual color values being assigned elsewhere in the application.

The power of the weaving language lies in its repertoire of expressions, which can used to describe sequence structure — patterns. There are 15 expressions in all, ranging from simple to complex in terms of their descriptive power.

The expressions are essentially the same for colors as for the threading and treadling, but some operators do not apply to colors. In our examples, we'll use the threading and treadling expressions.

The Domain

The concept of *domain* plays an important role in the weaving language. The domain consists of the digits that label the shafts and treadles. The sequence 12345678 is called the *domain run*. Sequences "wrap around" on the domain. That is, 1 follows 8 in situations involving domain runs. This is just arithmetic modulo 8 with the numbering starting at 1 instead of 0 to accommodate the convention used in weaving drafts.

Expression Syntax

Expressions are composed of *operators* and *operands* on which they operate. In ordinary arithmetic, 3 + 5 is an expression in which the operator + (addition) operates on its operands, 3 and 5 to produce 8. The operands of weaving operators are sequences and integers. The operators produce sequences. For example, in the weaving expression

```
16243 # 2
```

is the rotation operator, its left operand is a sequence and its right operand is the number of places to rotate to the left. The result is

```
24316
```

Most operators can be written in terms of names, short names, and symbols. For example, rotate, rot, and # are equivalent. We'll use symbols where they are available.

Spaces between operators and operands are optional in most situations and can be used to improve readability. Parentheses can be used to group operators

and their operands and also to improve readability.

The Operators

Concatenation

Concatenation (concat or ,) appends one sequence to another to produce a longer sequence. For example,

(16243#2),432

appends 432 to the result of rotating 16242 by two places and produces

24316432

Repetition

Repetition (repeat, rep, or *) repeats its left-operand sequence the number of times given by its right operand. For example,

1346*7

produces

1346134613461346134613461346

Blocks

In creating blocks (block or [ ]), the left operand is a pattern and the right operand is a sequence of integers. Each character in the left operand is repeated individually by the corresponding integer in the right operand. For example,

1346[ ]9231

expands to

111111111334441

Integers greater than 9 can be specified by enclosing them in braces. For example,

1345[ ]12{10}3

produces

1334444444444666

There also is an interleaved form. For example,

[1 1 3 2 4 10 5 3]

produces the same result as the example above.

02 Jul 2006 K2: PWL

Extension

The concept of *extension* permeates the weaving language, as it does weaving itself. Extension (extend, ext, or –>) replicates its left operand to produce a result whose length is given by its right operand. For example,

1346 –> 16

produces

1346134613461346

If the replication does not come out even, it is truncated on the right. For example,

1346 –> 15

produces

134613461346134

Reversal

Reversal (reverse, rev, or `) reverses the order of a sequence. In this case, there is only one operand the operator follows it in suffix position. For example,

12365238`

produces

83256321

Rotation

Rotation was described earlier but is included here for completeness. The left operand of rotation (rotate, rot, or #) is a sequence and its right operand an integer, which may be negative. It moves the specified number of characters from the beginning of the sequence and places them at the end. If the number is negative, the characters are moved from the end to the beginning. For example,

16243#–2

produces

43162

Palindromes

Palindromes are common in weaving and other design contexts. The palindrome operator (palindrome, pal, |), like rotation, is in suffix position. For example

1346|

produces

134643

Note that this is not a true palindrome — it does not read the same forward and backward. The center value, 6, is not duplicated and the value, 1, is omitted. It is what is called a *pattern palindrome*.

The reason pattern palindromes are not true palindromes has to do with their use in repeats. Consider, for example, the (true) palindrome 1346431, derived from 1346. If this is repeated, the result is

1346431134643113464311346431 …

Note the duplication of 1s at the boundaries of the repeats. This produces unavoidable and generally undesirable artifacts. On the other hand, if the pattern palindrome 134643 is repeated, there is no such artifact:

134643134643134643134643 …

When creating a true palindrome from a sequence, the last character of the sequence could be repeated, as in 13466431. This also produces artifacts:

1346613466431431134664311346643 1…

This also is not done in pattern palindromes.

Interleaving

Interleaving (interleave, int, ~) interleaves the characters of two sequences. For example,

1234 ~ 5678

produces

15263748

If one operand is shorter than the other, it is extended to the length of the other.

Domain Runs

There are four operators related to domain runs.

"Upto" (upto, <, or –) concatenates its left and right operands but inserts between them the portion of the domain between the last character of the left operand the first character of the right operand. For example,

1346 < 8

produces

02 Jul 2006 K2: PWL

134678

If, however, the last character in the left operand is greater than the first character in the right operand, the intervening portion “cycles” through the domain, so that

1346 < 3

produces

134678123

“Tick marks”, indicated by a single quotes, can be placed in front of the right operand. When this is done, the number of tick marks specifies the number of complete domain runs to be added in. For example,

1346 <'' 8

adds two domain runs and produces

1346781234567812345678

The operator – can be used in place of < if the last character of the left operand is less than the first character of the right operand, as in

123 – 765

There is a corresponding “downto” operator (downto, >, or –). For example,

8 > 3

produces

876543

The operator – can be used in place of > if the last character of the left operand is greater than the first character of the right operand.

The ”updown” operator, (updown or <>) produces alternating ascending and descending domain runs. The first, ascending, run starts at the first character of the left operand and goes to the first character of the second operand. The second, descending, run starts from there and goes to the second character of the right operand, and so on, alternating between ascending and descending runs. For example,

1234<>5678

produces

12345432345654345676545678

As in “upto”, tick marks can be used to indicate domain cycles between

02 Jul 2006 K2: PWL

runs.

The “downup” operator, (downup or ><), is like “updown” except that the order is descending, ascending, … .

Permutation

The permutation operator (permute or perm) applies a permutation vector (right operand) to a pattern (left operand). The pattern is permuted in sections whose lengths are the lengths of the permutation vector. The permutation vector specifies the positions of the elements in a section. For example, 4123 puts the fourth character of the section first, the first second, the second third and the third fourth. Thus,

```
1346 perm 4123
```

produces 6134.

In the case that the pattern is not the same length as the permutation vector, the pattern is extended to an *integer multiple* of the length of the vector.

Pattern Boxes

The pattern box operator (pbox), like perm, has a left operand pattern and a right operand permutation vector. The permutation vector is extended to the length of the pattern and the permutation is applied. For example,

```
123456787654321 pbox 21436587
```

produces

```
214365878563412
```

Templates

The template operator (template, temp, or :) provides for “sub-articulation” of a pattern (left operand) by a “texture pattern” (right operand). The first character (digit) in the template pattern is taken as the root. The remaining digits in the template pattern are taken with respect from their distance from the root. For example, in the template pattern 342 the root, r, is 3 and the template is r, $r + 1$, $r - 1$. The template is applied to each character in the pattern with the character replacing the root. For example,

```
12345678:121
```

has the template r, $r + 1$, r and produces

```
121232343565676787818
```

Note that values wrap around on the domain, so that for the last character of the pattern, 8, $r + 1$ produces 1.

Comments

It is interesting to note that the weaving language, as rich as it is, does not have operators for some patterns that occur frequently in weaving. Two missing ones are true palindromes and the interleaving of more than two sequences.

Another problem relates to domains. Although some of Painter's built-in weaves use only four shafts and four treadles, and only the labels 1, 2, 3, and 4, there is no way to restrict domain runs to this subset. For example, 4<2 produces 4567812, not 412 as it would if the domain could be restricted. Of course, 5678 does nothing.

The restriction to 8 shafts and 8 treadles is more fundamental, since it limits the kinds of things that can be woven. More shafts and treadles could be handled by extending the domain to include more labeling characters. If the number of shafts and treadles is not the same, the situation becomes more complicated, especially for domain operators.

The weaving language is difficult to use in the context of Painter. Expressions are limited to 256 characters. The fields for entering expressions are small and there is no way to find out what sequence an expression produces except by trying to puzzle it out in a resulting image.

It also is much harder to produce desired results in an expression-based language than it is to do so with a graphic representation.

The Painter weaving language is, nonetheless, very powerful. It allows complicated patterns to be expressed concisely and built up from simpler ones. It could well serve as a basis for a more powerful language that has variables, data structures, conditionals, control structures, and recursion. Such a language might not be suitable for designing weaves directly, but it could serve as a foundation for a powerful weaving program with graphical capabilities. [Mention article on an extension to PWL.]

02 Jul 2006 K2: PWL

A Boolean Design Program

Previous sections described the use of Boolean operations for designing drawdowns. The techniques described can be done by hand, at least for small patterns, but the process is so time consuming and error prone that it is not likely to be used, despite its potential for creating attractive and novel patterns.

This section describes an interactive program that makes it easy to design drawdowns using Boolean operations.

The Program

The program has three components:

- a design laboratory
- a drawdown editor
- a Boolean operator array editor

The editors are invoked from the design laboratory and are subordinate to it.

The Design Laboratory

Figure Ω.1 shows the Design Laboratory interface.

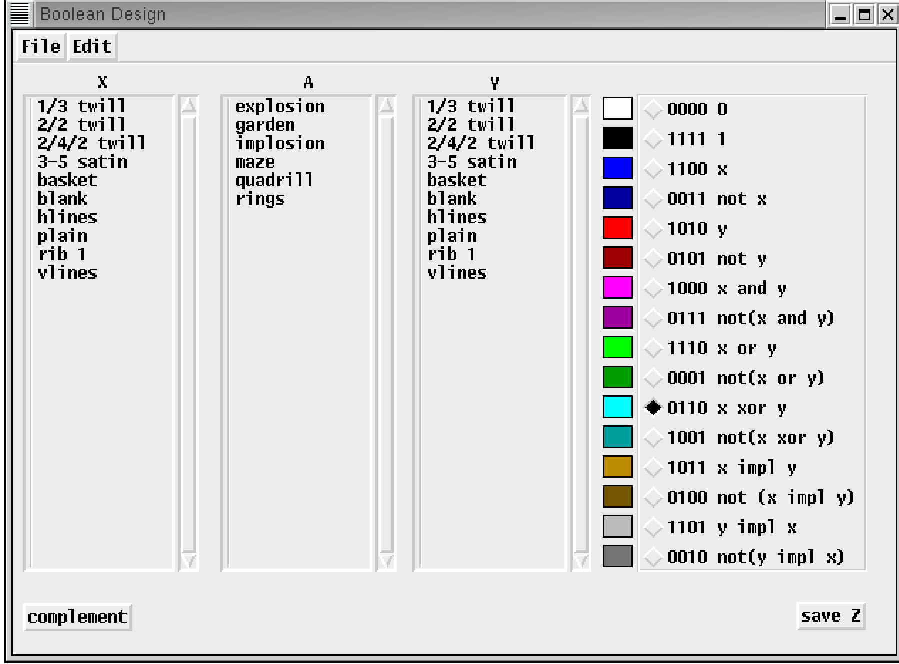


Figure Ω.1 Program Interface

The three lists at the left correspond to the components of a Boolean operation:

$$X\ A\ Y \rightarrow Z$$

where X and Y are Boolean arrays representing drawdowns, *A* is an array of Boolean operations [2], and Z is the drawdown resulting from applying *A* to X and Y.

The X and Y lists are the same, and contain the names of available drawdowns. The *A* list contains the names of the available Boolean operator arrays.

Some drawdowns and arrays are built in; others can be added when the program is run.

Clicking on a name in one of the lists makes the corresponding drawdown or operator array current. Windows show the current drawdowns and operator array. Selecting plain for X, rings for *A*, and 2/2 twill for Y produces the result shown in Figure Ω.2.

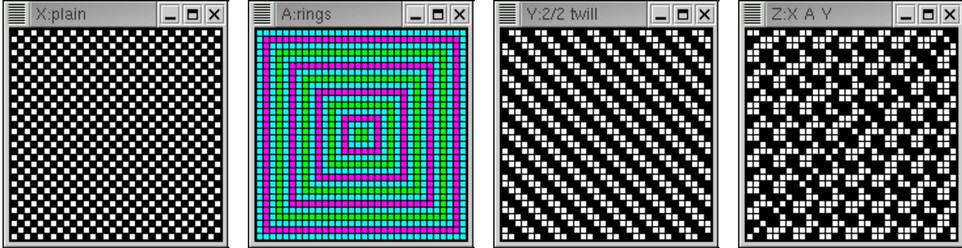


Figure Ω.2. Drawdown and Operator Array Windows

The panel on the right side of the design laboratory interface shown in Figure Ω.1 contains buttons for selecting a Boolean operator that applies to all cells in the X and Y drawdowns. Three forms of identification are provided for the operators: the associated color [2], the value pattern, and a symbolic form.

Selecting an operator replaces the current array by one in which all operators are the selected one. The Z drawdown is updated accordingly. Figure Ω.3 shows the result of selecting *exclusive or* operation.

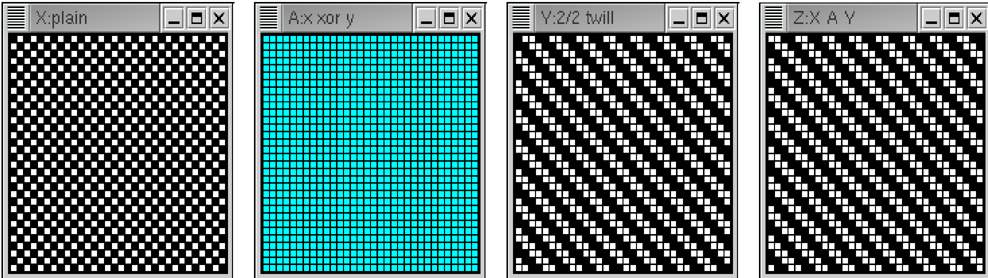


Figure Ω.3. Drawdown and Operator Array Windows After Selecting *exclusive or*

The complement button at the bottom left of the design laboratory interface brings up a dialog in which X, Y, or Z can be complemented. See Figure Ω.4.

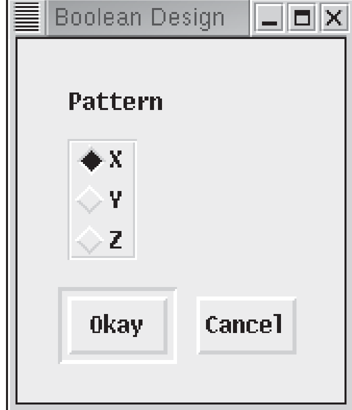


Figure Ω.4. Complement Dialog

The Save Z button at the bottom right of the design laboratory interface brings up a dialog in which a name can be specified for saving the Z pattern. See Figure Ω.5.

Figure Ω.5. Save Z Dialog

The Z pattern is saved and the name is added to the X and Y lists.

The File menu, shown in Figure Ω.6, provides items for saving the current design-laboratory configuration in a database, loading a saved database, importing and exporting patterns and arrays, and quitting the application.

Figure Ω.6. The File Menu

The notations at the right provide reminders of keyboard shortcuts that can be used in place of the menu. The symbol @ indicates that a keyboard shortcut is invoked by depressing the meta modifier key and pressing the letter that follows. For example, pressing q while the meta modifier key is depressed quits the application.

The Edit menu, shown in Figure Ω.7, provides items for invoking the drawdown and operator array editors.

Figure Ω.7. The Edit menu

If drawdown is selected, a dialog listing the available patterns is presented. See Figure Ω.8.

Figure Ω.8. Drawdown Editor Dialog

The drawdown editor is launched with the selected drawdown.

If array is selected, a dialog listing the available operator arrays is presented. See Figure Ω.9.

Figure Ω.9. Operator Array Editor Dialog

The operator array editor is launched with the selected array.

The Drawdown Editor

Figure Ω.10 shows the drawdown editor.

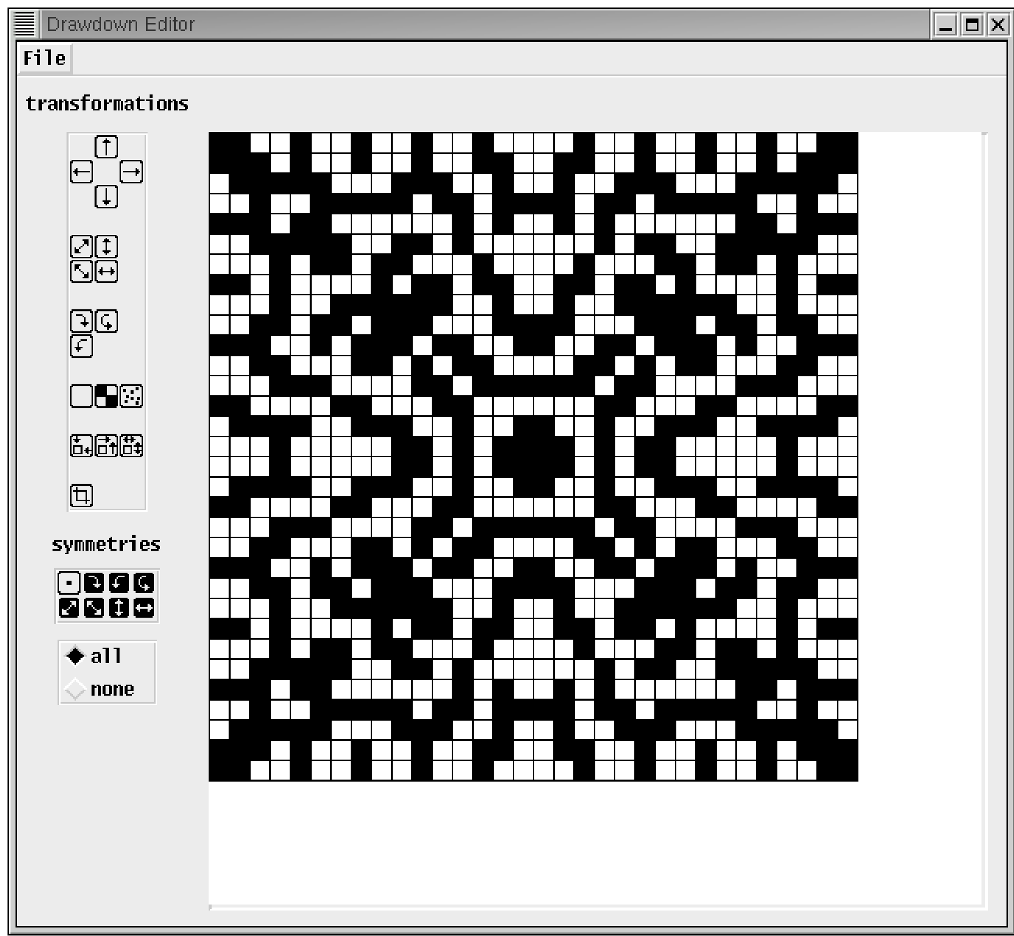
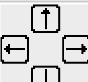


Figure Ω.10. The Drawdown Editor

The drawdown to be edited is displayed on the right side of the editor window. Various editing tools are available through the panel on the left side of the window.



These buttons allow the drawdown to be rotated circularly in

each of the four directions, one cell at a time.

These buttons allow the drawdown to be flipped around the axes indicated.

These buttons allow the drawdown to be rotated in increments of 90° as indicated.

These buttons affect the contents of cells. The left button clears all cells to white. The middle button inverts the drawdown, changing white cells to black and vice versa. The right button randomizes the pattern.

The Operator Array Editor

The operator array editor operates in a fashion similar to the application for creating weaveable patterns [3]. See that article for a basic description.

The operator array editor window is shown in Figure Ω.11 .

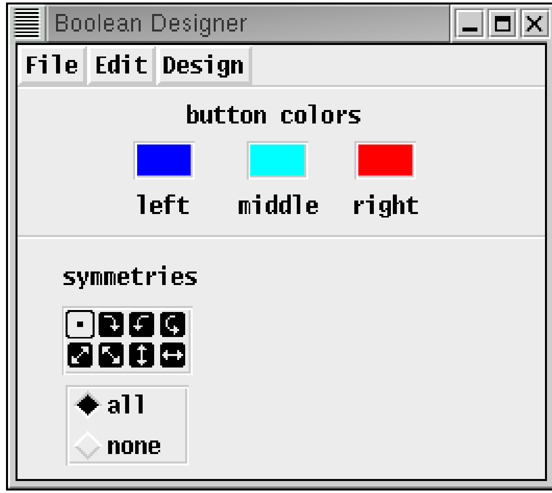


Figure Ω.11. Operator Array Editor Window

The palette of colors used to identify Boolean operators is displayed in a separate window. See Figure 12.

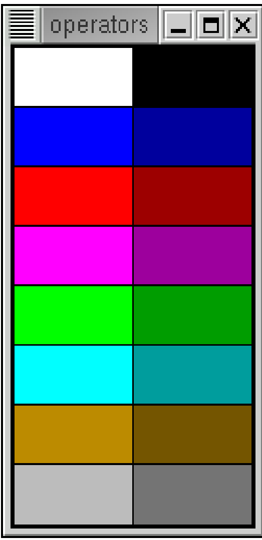


Figure Ω.12. Boolean Operator Palette

A third window displays the current operator array. See Figure 13.

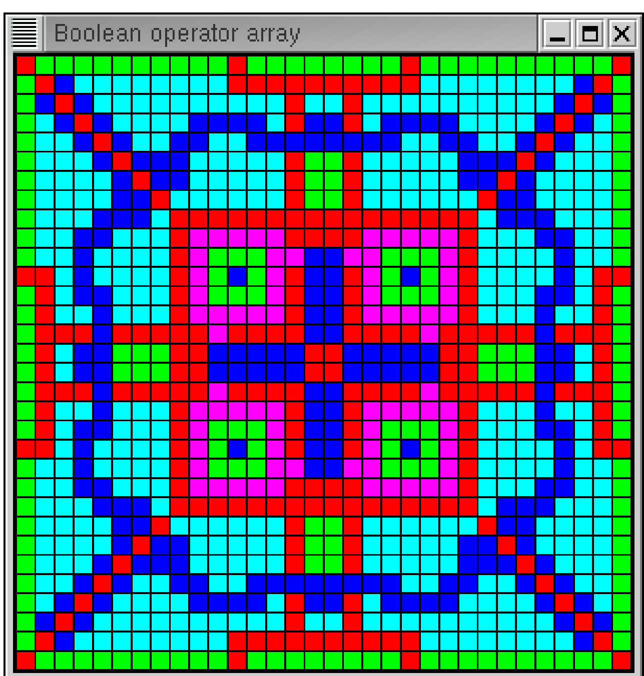


Figure Ω.13. Boolean Operator Array

The File menu provides the usual items for importing and exporting patterns and dismissing the editor. When the array operator editor is dismissed, a dialog is presented to name the array. The array then is added to the *A* list.

A Color Design Application

Most handweavers design using drafts — threading sequences, treadling sequences, tie-ups, and the resulting drawdown. The warp and weft threads may be assigned colors, and the resulting pattern viewed in a "color drawdown". This process assures a weaveable pattern. There is no way to produce one that is not weaveable.

A previous section described how to ensure weaveability in algorithmically constructed patterns and showed transformations on weaveable patterns that preserve weaveability [1].

The program designed here uses a different approach to constructing weaveable color patterns; one in which a designer constructs color patterns "from scratch" but not in the context of drafting. Instead, the designer uses an interactive computer application that prevents anything that would result in an unweavable pattern.

The Application

The application displays several windows: an interface window that provides controls for the user, a design window in which the user creates designs, and a palette window that displays the colors available for design.

The Interface

The application interface, shown in Figure Ω.1, displays three colors associated with the left, middle, and right mouse buttons, respectively.

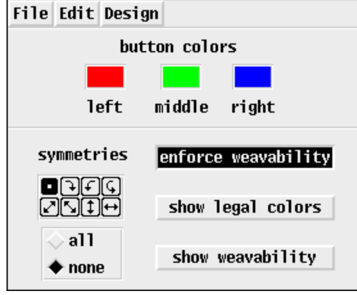


Figure Ω.1. The Interface Window

The initial colors are red, green, and blue. These colors can be changed as described later.

The Design Window

The design window consists of a rectangular array of cells. Initially all cells are colored with the middle mouse button color. See Figure Ω.2.

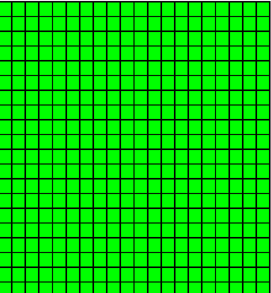


Figure Ω.2. The Design Window

If the user clicks on a cell in the design window, the cell is colored with the color associated with the button used — provided the result would be weaveable. If the result would not be weaveable, the change is not made and there is an audible alert.

The user also can click and drag to color several cells at one time. The test for weaveability is not made until the mouse button is released. If the result would not be weaveable, the application backtracks, removing the most recently colored cells until there is a weaveable result.

The Palette Window

The color associated with a mouse button can be changed by clicking with that button on a cell in the palette window. The initial palette provides a range of colors as shown in Figure Ω.3, but other palettes are available and can be selected as described later.

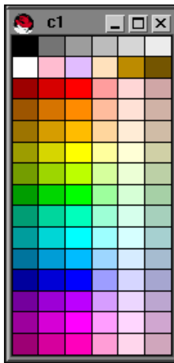


Figure Ω.3. The Palette Window

Symmetrical Designing

The application supports symmetrical designing in which cells in symmetric positions are colored. Symmetries can be selected from the symmetry panel

on the application interface. See Figure Ω.1.

The default is no symmetry, so only the color of the cell under the mouse pointer is changed. This is indicated by the highlighted button in the upper-left corner of the symmetry panel. Various combinations of symmetries can be selected by clicking on the icons for individual symmetries. See Reference 2 for a detailed explanation.

All symmetries can be enabled by choosing the all radio button below the symmetry panel. Figure Ω.4 shows a design produced by using symmetries.

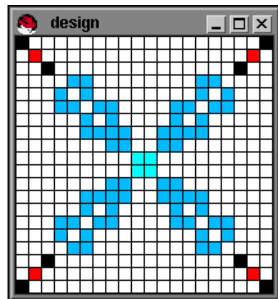


Figure Ω.4. A Symmetrical Design

Layout

The default layout for the design window is a 20 × 20 array of 10-pixel square cells. See Figure Ω.5.

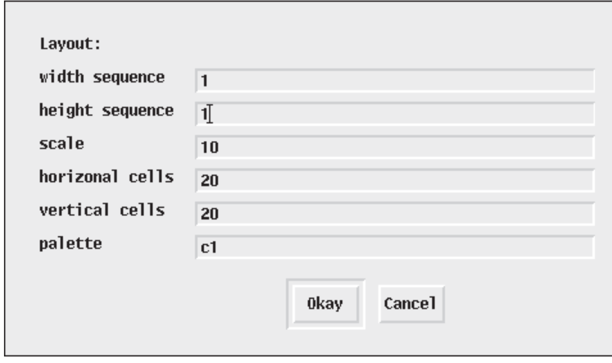


Figure Ω.5. The Default Layout

The cells need not be square. Their widths and heights are determined by the values in sequences in the first two fields of the layout window. These sequences are repeated as necessary to fill out the specified number of cells. A scaling factor is applied to these values.

Finally, a palette can be specified. The default palette is named `c1`, which is shown in Figure Ω.3. See Reference 3 for information about the other palettes that are available.

The initial colors for buttons for a new design are colors in the given palette that are close to red, green, and blue, respectively.

Figure Ω.6 shows a layout based on the Fibonacci sequence, which is reflected to produce a symmetric result. The palette `g8` provides eight equally spaced shades of gray. The resulting design window is shown in Figure Ω.7 and the new interface and palette windows are shown in Figures Ω.8 and Ω.9.

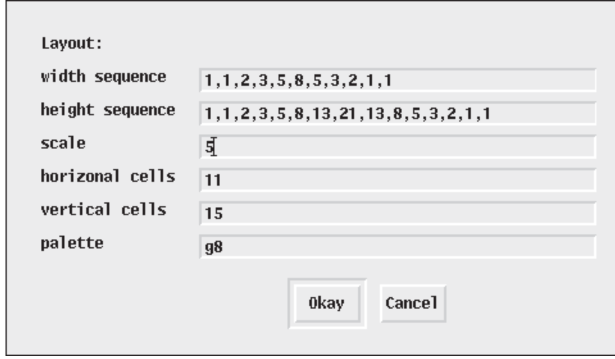


Figure Ω.6. A Fibonacci Layout

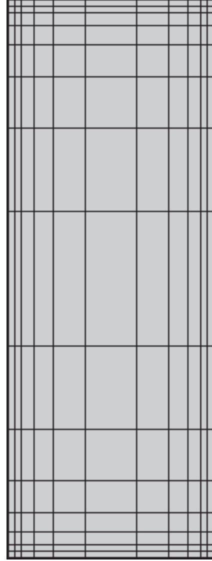


Figure Ω.7. A Fibonacci Design Window

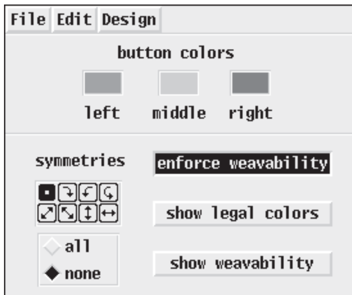


Figure 8. New Interface Window

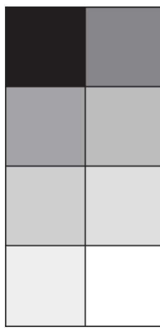


Figure 9. New Palette Window

Enforcing Weaveability

The enforce weavability button on the interface is a toggle, which initially is on. If it is off, changes in the design window are not checked for weavability.

One reason for not enforcing weavability is that it often is not possible to get from one weaveable pattern to another by changing the colors of cells one at a time. One way to accomplish changes that preserve weavability but cannot be done piecewise is to disable weavability testing, make the changes, and then enable weavability testing. (Should the result not be weavable, the previous changes are undone as necessary the next time a change is made.)

Legal Colors

The show legal colors button toggles the visibility of a window that mimics the design window. Clicking on a button color region on the interface window (see Figure 1) overlays on the legal color window all the cells in the design that could be made that color while preserving weavability. See Figure 10.

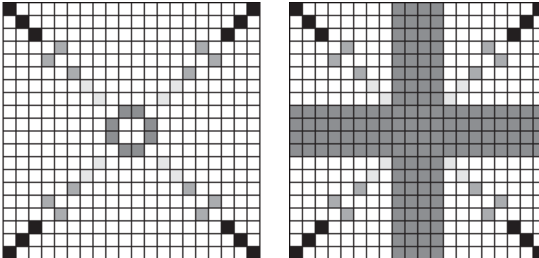


Figure Ω.10. Design and Legal Colors Windows

The legal color window only shows individual cells that can be colored; it does not show all combinations of cells that together would preserve weavability. Indeed, all cells in the design window can be made one color with a result that is trivially weavable.

Showing Weavability Testing

The show weavability button toggles the visibility of a window that shows the result of the last weavability test [4]. This result shows the row and column colors determined by the test. See Figure Ω.11.

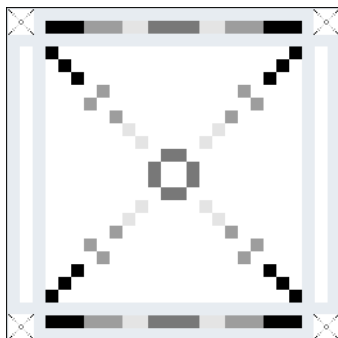


Figure Ω.11. Weavability Solution

Menus

The File menu, shown in Figure Ω.12, has items for saving an image of the design window, loading a custom palette database, and quitting the application.

Figure Ω.12. The File Menu

The Edit menu, shown in Figure Ω.13, provides items for undoing and redoing the last change to the design window.

Figure Ω.13. The Edit Menu

There is no limit to the number of changes that can be undone or redone; it is possible to move backward and forward through the entire history of a design. As indicated, the stack for saved designs can be cleared, which frees the memory they occupy.

The Design menu, shown in Figure Ω.14, provides items related to the design.

Figure Ω.14. The Design Menu

The new item brings up the layout dialog described previously. Entire designs can be saved and loaded as indicated. The clear item clears the design to a single color. It brings up a dialog in which the user can chose between the left, middle, and right button colors.

Experience with the Application

Experience with the application is limited and the only users so far have been familiar with the concepts of weaving and color weavability but not experienced weavers.

If the user doesn't think in terms of row and color assignments (doesn't know the rules) but considers the application as a game, the results can be more frustrating than interesting.

Starting with a "blank" (solid-colored) design window, cells with a second color can be drawn in any fashion, since all two-colored designs are weavable. However, it may be impossible to change any cell to a third color unless the second color was used judiciously.

The user learns to reserve areas of cells so that they can be colored with other colors, the most notable being stripes (it's always possible to change all the cells in a row or column to a new color).

Figures Ω.15-Ω.19 show examples of weavable color patterns created using the application described here.

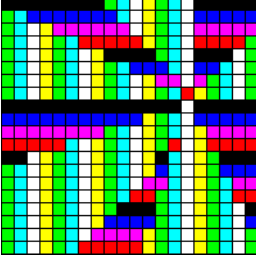


Figure Ω.15. Vivid Geometry

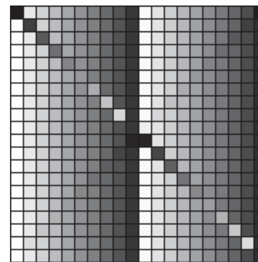


Figure Ω.16. A Study in Grays

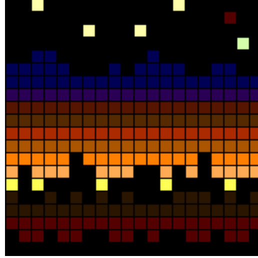


Figure Ω.17. Skyline at Dawn

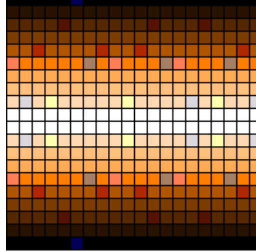


Figure Ω.18. Shimmer

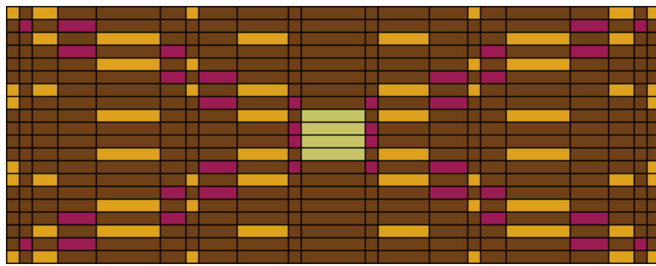


Figure Ω.19. Parquet