The Icon Analyst

In-Depth Coverage of the Icon Programming Language and Applications

August 2000
Number 61

In this issue

The Final Year of the Analyst

With this issue, we are entering the eleventh year of publication of the Analyst. This is far longer than we could have imagined when we started.

In recent years, as we've covered most aspects of Icon and its implementation, we've moved more toward application areas for which Icon is apt.

There is no shortage of material for future Analyst articles, but it has become increasingly demanding and oppressive to develop and present material that is essentially work in progress instead of presenting material already familiar to us.

At the same time, our subscriber base has diminished, which is quite natural under the circumstances. As a result, the amount of time and effort to produce the Analyst is expended for fewer and fewer readers.

For these reasons, as well as interests in other things, we've decided that this will be the last year of the Analyst — the final issue, 66, is scheduled for June 2001.

In this final year, we expect to finish the series of articles on sequences and to cover most of the remaining material we have on topics related to weaving. We won't be able to resist starting something new that we can't finish, just as other ongoing topics such as program visualization and versum sequences certainly will be left hanging. But this would be the case however long the Analyst lasted.

We greatly appreciate the loyalty of our subscribers, many of whom have been with us since the first issue. Thank you for making this a fun adventure.

If your subscription to the Analyst expires before the last issue, you'll get a renewal notice with the subscription charge reduced according to the number of issues remaining.

Refunds will be made for unfilled portions of subscriptions.

Tricky Business

When we were preparing an article on patterns in sequences, we wanted to indicate variable terms that took on different values instead of being constant. The kind of thing we wanted was

```
[1,X,4,1,4,X,1]
```

That's easy enough to do for any particular case, but we wanted the capability in a program that could be used to produce many such patterns.

The lists module of the Icon program library has a procedure, limage(), that does almost what we wanted:

```
procedure limage(L)
   local result

   result := ""

   every result ||:= image(!L) || ","

   return ("[" || result[1:-1] || "]") | "[]"

end
```

For example, if

```
sequence := [1,2,4,6,4,2,1]
```

then limage(sequence) produces the string

```
[1,2,4,6,4,2,1]
```

However, if

```
sequence := [1,"X",4,"Y",4,"X",1]
```

then limage(sequence) produces the string

```
[1,"X",4,"Y",4,"X",1]
```

with quotation marks we didn't want.

The quotation marks come from the use of image(), which serves two purposes: (1) It makes limage() safe for values of any type (otherwise a value of a type that can't be converted to a string would result in a run-time error when concatenated) and (2) It gives the user information about the types of the values in the list.

Our first thought was to write a customized version of limage() for our purposes. But then we had a better thought: Change the value of image to 1, so that when it's applied to string values, no quotes are produced. So we wrote

```
image := 1
result := limage(sequence)
```

and the string assigned to result is what we wanted:

```
[1,X,4,Y,4,X,1]
```

Of course we didn't want to leave the value of image like that — it might be needed somewhere else. We could have saved the value of image before calling limage() and restored it after, but an easier way is to use proc() to assign the built-in function back to image:

```
image := 1
result := limage(sequence)
image := proc("image", 0)
```

Fractal Sequences

The term *fractal* is used in a variety of ways, formally and informally. It generally is understood that a fractal exhibits self similarity — that it appears the same at any scale.

This concept can be applied to integer sequences with respect to the magnitude and position of terms, various patterns, and so forth.

For example, Hofstadter's chaotic sequence [1-2], which is produced by the nested recurrence

$$q(i) = 1 \qquad i = 1, 2$$
$$q(i) = q(i - q(i-1)) + q(i - q(i-2)) \qquad i > 2$$

shows self similarity as can be seen in Figure 1.

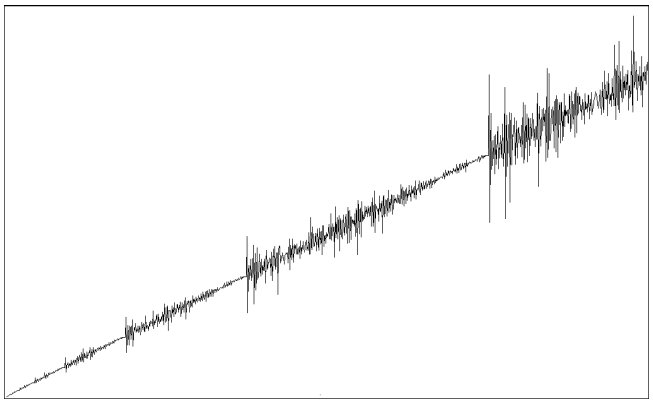


Figure 1. Hofstadter's Chaotic Sequence

In this sequence, sections of wide variations tailing off to minor variations double in length. The magnitude of the variations roughly doubles also. Within a section, you can see articulation of the preceeding section. Despite its tantalizing structure, this sequence is not strictly fractal.

Kimberling Fractal Sequences

Clark Kimberling has a precise definition of what he considers to be fractal integer sequences [3]. First, a Kimberling fractal sequence must be *infinitive*, which means that every positive integer occurs in it an infinite number of times.

An infinitive sequence $\{x_n\}$ has an *associative array*, $a(i,j)$, whose values are the jth indices for which $x_n = i$ for $i, j = 1, 2, 3, \ldots$.

$\{x_n\}$ is a fractal sequence if the following two conditions hold:

1. If $x_n = i+1$, then there is an $m < n$ such that $x_m = i$.
2. If $h < i$, then for every j there is exactly one k such that $a(i,j) < a(h,k) < a(i,j+1)$.

We think there must be a simpler (or at least clearer) way to state this, but we don't have sufficient interest to puzzle it out.

Such sequences have the property that if you strike out the first instance of every value, the resulting sequence is the same as the original (such sequences are, of course, infinite, which allows the concept of "same as" after deleting terms). An example of such a sequence is

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

Striking out the first instance of every term,

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

produces

1, 1, 1, 1, 2, 1, 2, 1, 3, 2, 1, 3, …

which is the same as the original sequence, as far as it goes.

There are two operations that when applied to fractal sequences yield fractal sequences: upper and lower trimming. Upper trimming is the "strike out" operation illustrated above. Lower trimming consists of subtracting 1 from every term and discarding 0s. Here are programmer-defined control operations for trimming sequences:

```
procedure UpperTrimPDCO(L)
   local done, i

   done := set()

   while i := @L[1] do {
      if not member(done, i) then
         insert(done, i)
      else suspend i
      }

end

procedure LowerTrimPDCO(L)
   local i

   while i := @L[1] do {
      i –:= 1
      if i ~= 0 then suspend i
      }

end
```

Signature Sequences

An interesting class of Kimberling 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 is the signature sequence for x, which we'll denote by $\mathcal{S}(x)$.

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, …

A grid plot gives a better idea of the structure of the sequence, which is typical of signature sequences. See Figure 2.

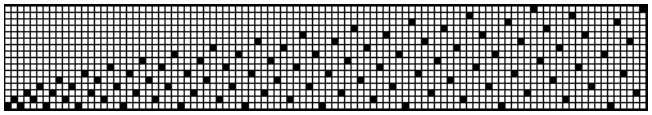


Figure 2. $\mathcal{S}(\phi)$

Upper trimming and lower trimming of a signature sequence leave the sequence unchanged.

Here's a procedure for producing signature sequences:

```
record entry(value, i, j)

procedure signaseq(x, limit)
   local result, i, j

   /limit := 100
```


The Icon Analyst

Ralph E. Griswold, Madge T. Griswold,
and Gregg M. Townsend
Editors

The Icon Analyst is published six times a year. A one-year subscription is $25 in the United States, Canada, and Mexico and $35 elsewhere. To subscribe, contact

Icon Project
Department of Computer Science
The University of Arizona
P.O. Box 210077
Tucson, Arizona 85721-0077
U.S.A.

voice: (520) 621-6613

fax: (520) 621-4246

Electronic mail may be sent to:

icon–analyst@cs.arizona.edu

THE UNIVERSITY OF
ARIZONA®
TUCSON ARIZONA
and

Bright Forest Publishers
Tucson Arizona

```
    result := []
    every j := 1 to limit do
       every i := 1 to limit do
          put(result, entry(i + j * x, i, j))
    return sortf(result, 1)
end
```

Notice the use of sortf() to sort the result by the first field of its records.

Signature Sequences in Weave Design

We started to explore fractal sequences as the result of an assignment for Complex Weavers' Mathematics and Textiles Study Group [4]. Three examples of weave patterns that we have produced so far are shown in Figures 3 through 5. All use residue sequences [5] from signature sequences.

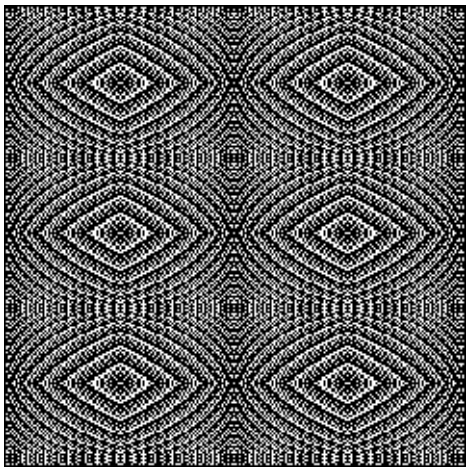


Figure 3. $\mathcal{S}(e)$Threading, $\mathcal{S}(\phi)$ Treadling

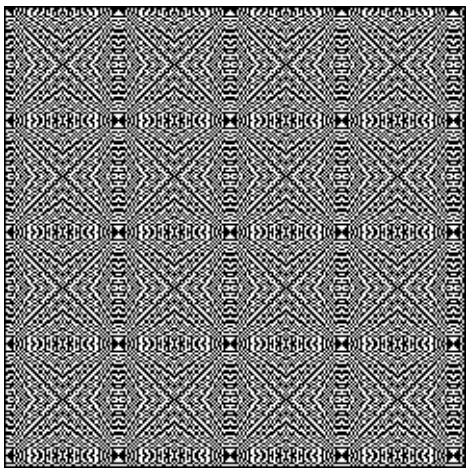


Figure 4. $\mathcal{S}(\pi)$ Threading and Treadling

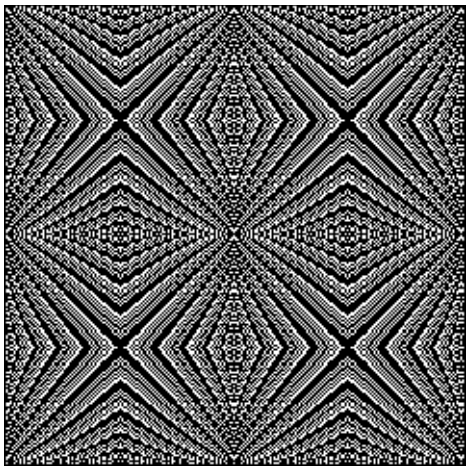


Figure 5. $\mathcal{S}(e)$ Threading, $\mathcal{S}(\pi)$ Treadling

More to Come

There are kinds fractal sequences other than Kimberling's. We'll consider one of these in a subsequent article.

References

1. *Gödel, Escher, Bach: An Eternal Golden Braid*, Douglas R. Hofstadter, Basic Books, 1979, pp. 137-138.

2. "Procedures with Memory", Icon Analyst 21, pp. 8-11.

3. *CRC Concise Encyclopedia of Mathematics*, Eric W. Weisstein, Chapman & Hall/CRC, 1999, pp. 674.

4. http://complex-weavers.org/study20.htm

5. "Residue Sequences", Icon Analyst 58, pp. 4-6.

Tie-Ups and T-Sequences

Tie-ups control the way that sheds of warp threads are formed during weaving for successive picks of weft threads [1].

Tie-ups cannot be taken in isolation. The other factors that determine the pattern of interlacement are the T-sequences — the threading sequence of warp threads through the shafts and the treadling sequence that determines which shafts are raised via the tie-up.

The tie-ups used in this article are chosen to illustrate principles without regard for float length or structural integrity. Except for the twill examples, the tie-ups used here are not appropriate for actual weaving.

The simplest T-sequences are straight draws that go though shaft and treadle numbers in succession and then repeat. There are two kinds of straight draws: upward with ascending numbers and downward with descending numbers. We'll use the symbols ↗ and ↘ for these, respectively.

Different combinations of straight draws replicate the tie-up in different orientations as shown in Figures 1 through 4. Note the arrows in the upper-left of Figure 1 that indicate increasing values for the different components of the draft. For example, "upward" for the treadling sequence is from right to left.

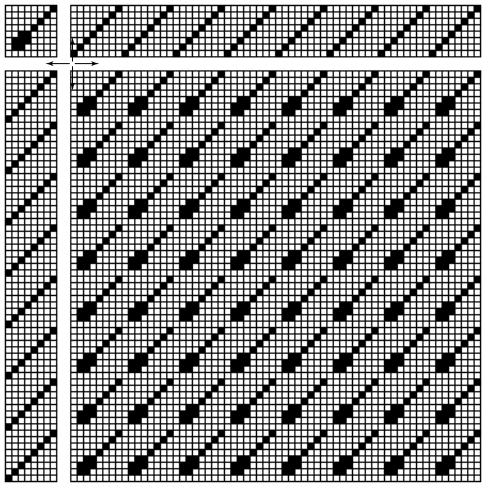


Figure 1. ↗ ↗ Draws

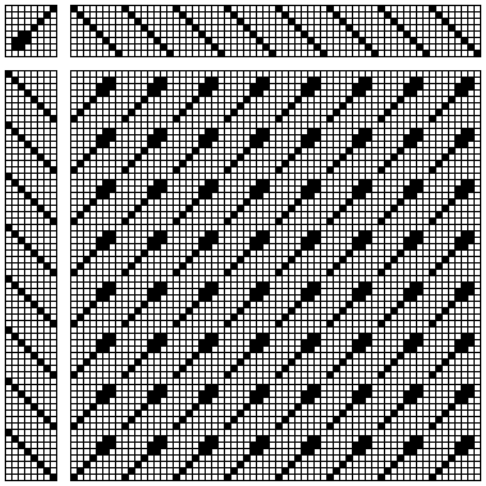


Figure 2. ↘ ↘ Draws

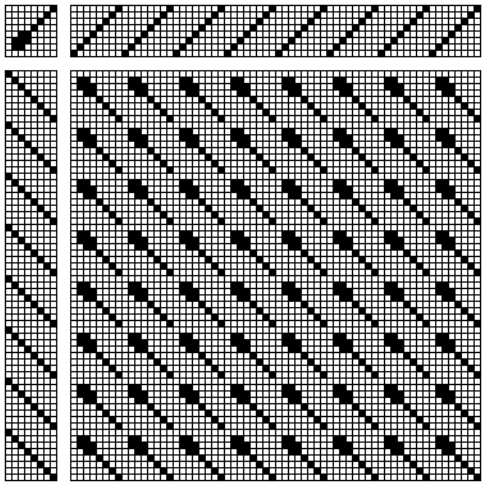


Figure 3. ↗ ↘ Draws

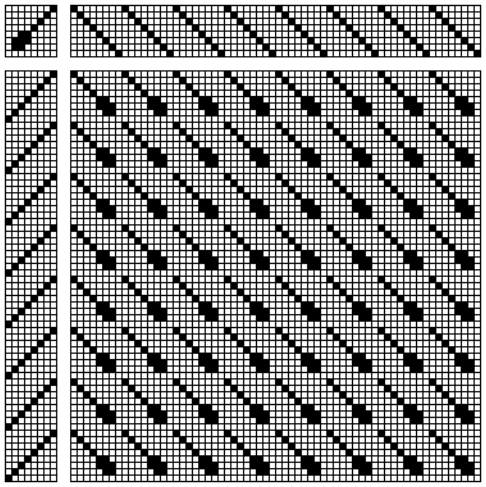


Figure 4. ↘ ↗ Draws

Other commonly used T-sequences go up and then down (↗↘) or down and then up (↘↗). These are called a wave draws.

Figures 5 and 6 show patterns resulting from wave draws.

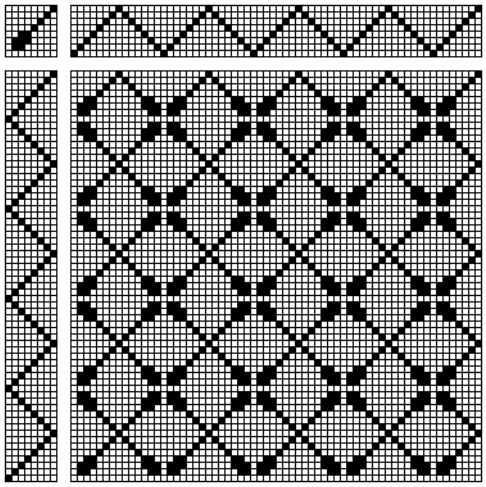


Figure 5. ↗↘ ↗↘ Draws

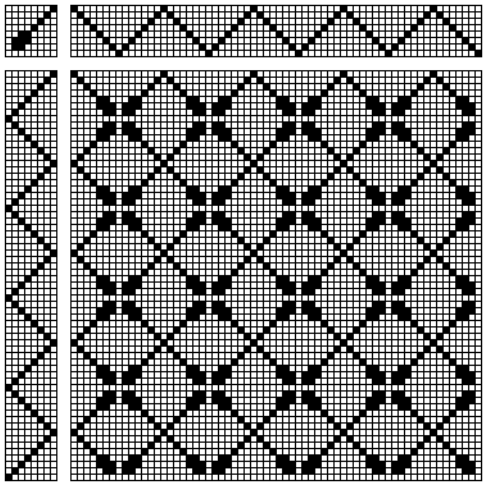


Figure 6. ↘↗ ↘↗ Draws

Notice that the drawdown pattern in Figure 6 is the same as in Figure 5, but offset horizontally and vertically.

So far, we've only shown patterns of interlacement that result from a tie-up designed to indicate orientation. Figures 7 though 9 show the patterns that result from a /2/2 twill with different combinations of draws.

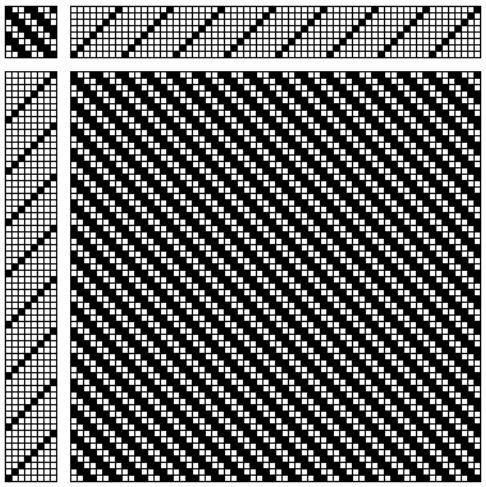


Figure 7. ↗↗ Draws with /2/2 Twill

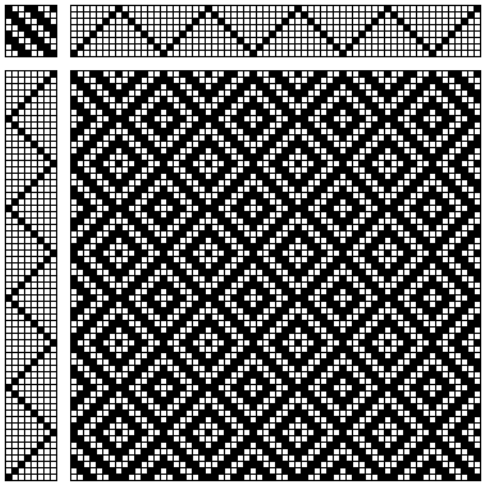


Figure 8. ↗↘ ↗↘ Draws with /2/2 Twill

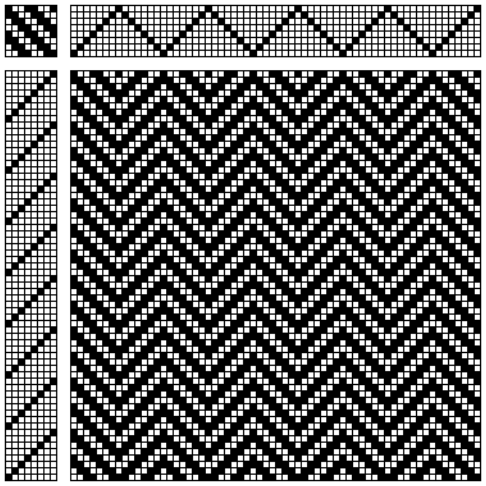


Figure 9. ↗↘ ↗ Draws with /2/2 Twill

If the tie-up is not square, the patterns depend on symmetries in the tie-up. For example, the pattern for a /2/2 twill with an 8×12 tie-up for straight draws is the same as for an 8×8 tie-up. See Figure 10.

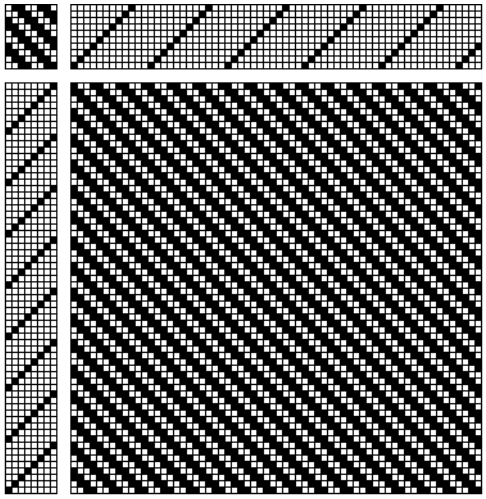


Figure 10. ↗ ↗ Draws with 8×12 /2/2 Twill

If, however, the tie-up is asymmetric, the pattern becomes more complex. See Figures 11 and 12.

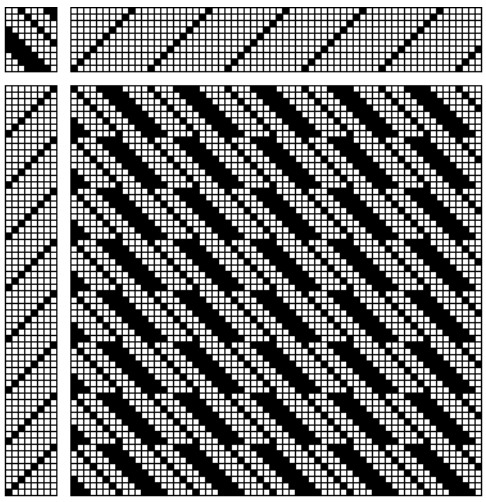


Figure 11. ↗ ↗ Draws with 8×12 /1/4/3 Twill

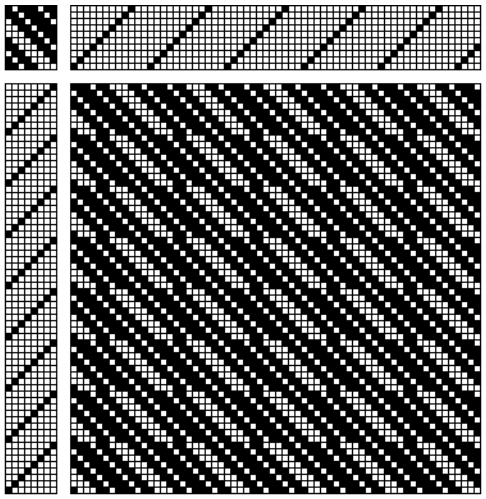


Figure 12. ↗ ↗ Draws with 8×12 /3/2/1/2 Twill

These drawdown patterns result from replicating the tie-up that does not tile seamlessly with itself. The cause of the pattern, however, may not be readily apparent to the eye.

As a final illustration of the interaction between tie-ups and T-sequences, consider the results for a motif tie-up and two combinations of draws as shown in Figures 13 and 14.

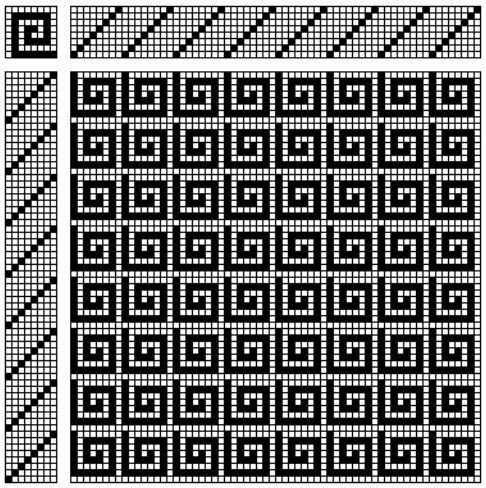


Figure 13. ↗ ↗ Draws with a Motif Tie-Up

Supplementary Material

Supplementary material for this issue of the Analyst, including images, is available on the Web. The URL is

http://www.cs.arizona.edu/icon/analyst/iasub/ia61/

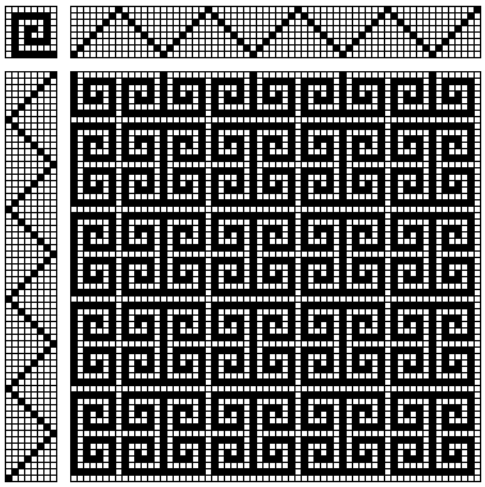


Figure 14. ↗↘↗↘ Draws with a Motif Tie-Up

Implementing T-Sequences

In future articles, we'll explore T-sequences in more detail. The ones shown in this article consist of simple segments that are repeated. Much more complex sequences are used in weaving, including ones that do not repeat. We'll need good methods of constructing such sequences.

There are two possible approaches: (1) "active" sequences whose values are produced as needed by generators, and (2) "passive" sequences that are stored in data structures.

Generating Sequences

We've described the ways that sequences can be generated in many previous Analyst articles.

There usually are several ways that the same sequence can be generated. For example, an upward straight draw can be generated by

```
|(1 to Shafts)
```

or

```
residue(seq(1), Shafts, 1)
```

where Shafts is the number of shafts.

The first expression implements a literal interpretation of an upward straight draw, using repeated alternation to repeat the segment indefinitely. The second expression uses knowledge about residue sequences [2].

Downward straight draws are only slightly more complicated:

```
|(Shafts to 1 by –1)
```

or

```
residue(–seq(0), Shafts, 1)
```

The expressions above generate unending sequences. Limitation can be used to "fit" them to a given draft, as in

```
|(Shafts to 1 by –1) \ Width
```

where Width is the number of warp threads.

Wave sequences, although simple, suggest what is involved in more complex T-sequences.

A straightforward approach to generating a wave sequence is

```
|((1 to Shafts – 1) | (Shafts to 2 by – 1))
```

This expression takes into account that the top and bottom values are not duplicated.

Of course, wave sequences are palindromic. Therefore we can use a programmer-defined control operation [3] to generate them:

```
|PatternPalinPDCO{1 to Shafts}
```

The procedure name is long and ugly, but this expression captures the idea.

Data Structures

The natural data type to use for stored sequences is the list. For example, the following list serves as the basic unit of an 8-shaft upward straight draw.

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

Using procedures in the lists module of the Icon Program library, the unit can be replicated using lrepl() or extended by repetition using lextend().

Another procedure, lreflect(), can be used to create wave sequences.

Strings provide another way of representing sequences. For example, "12345678" could be used as the basic unit for an 8-shaft upward straight draw.

If the number of shafts is greater than nine, it's necessary to encode numbers as characters, as in "123456789a" for a 10-shaft upward straight draw. (We used "a" rather than "0" to stand for 10 to avoid possible misinterpretations.)

The main advantages of representing sequences as strings are the small amount of memory they require and the ability to use string pattern matching to analyze them. The disadvantages are

the need to encode numbers by characters, the limited number of characters available, and the processing time involved when actual numbers are needed. For T-sequences, lists usually are preferable to strings.

Generators and Data Objects

Generators can be used to construct data objects that contain sequences. For example, the list for the basic unit of an upward straight draw might be constructed as follows:

```
up := []
every put(up, 1 to Shafts)
```

Conversely, the unary operator ! generates the elements of a list, as in

```
!up
```

This operator is polymorphous and works just was well for strings.

Comments

Since the interactions of tie-ups and T-sequences are so complex, weavers tend to design using variations of drafts that are known to produce attractive results and good fabric integrity.

Actual weaving is time-consuming and labor intensive. Historically, experienced and creative weavers have designed on paper. Changing a tie-up or T-sequence on a paper draft requires that the drawdown be redone to see the resulting interlacement pattern.

During weaving, planned treadling patterns may be changed as the fabric emerges. The threading, however, is done in advance and fixed during weaving. On floor looms, the tie-up is done in advance and although it can be changed during weaving, the process is difficult and time consuming. Table looms, which do not have tie-ups or treadles, use levers to select the shafts to be raised for every pick. Table looms are more versatile than floor looms in this respect.

The emergence of weaving programs has greatly expanded the design possibilities for weavers. A change can to a tie-up or T-sequence can be made easily and the results reflected in the drawdown almost immediately.

Weaving programs have many powerful features, but they are designed around conventional approaches to weave design.

In an upcoming series of articles, we'll describe a weaving program written in Icon that uses a novel approach to design and focuses on the use of sequences.

References

1. "A Weaving Language", Icon Analyst 51, pp. 5-11.

2. "Residue Sequences", Icon Analyst 58, pp. 4-6.

3. "From the Library — Programmer-Defined Control Operations", Icon Analyst 56, pp. 3-4.

Continued Fractions for Quadratic Irrationals

> *The mathematical phenomenon always develops out of simple arithmetic, so useful in everyday life, out of numbers, those weapons of the gods: the gods are there, behind the wall, at play with numbers.*
>
> — Le Corbusier

In this article, we'll start to explore continued-fraction sequences for quadratic irrationals. To put this subject in perspective, Figure 1 shows a classification of real numbers with parenthetical notes about their continued-fraction sequences.

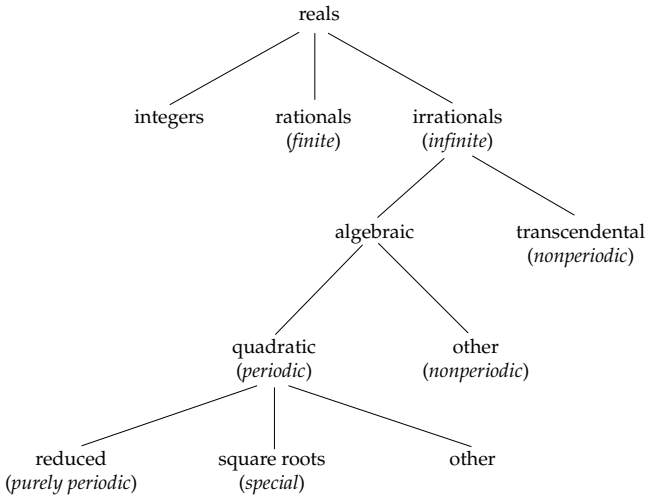


Figure 1. Classification of Reals

Integers have no continued fractions as such. A rational number is, of course, the ratio of two nonzero integers. Rationals have finite continued-fraction sequences. Real numbers that are not integers or rationals are, by exclusion, called irrational. All irrational numbers have infinite continued-

fraction sequences.

Irrationals, in turn, are divided into two categories. One category consists of the algebraic numbers, which are real roots of polynomial equations of the form

$$a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n = 0$$

Again by exclusion, all other irrationals are called transcendental. Examples of transcendental numbers are π and e. Transcendental numbers have non-periodic continued-fraction sequences, although some are known to have patterns [1].

For purposes of studying continued-fraction sequences, algebraic irrationals are divided into two categories, quadratic and "other". Quadratic irrationals have the form

$$\frac{a+\sqrt{b}}{c}$$

where a, b, and c are integers and b is positive and not a perfect square.

The division of algebraic irrationals into quadratic and "other" is important, because quadratic irrationals have periodic continued-fraction sequences and no other real numbers do.

The "other" category includes solutions of higher-order algebraic equations. Very little is known about their continued-fraction sequences except that they are not periodic. For example, it's not even known if the terms in the continued-fraction sequence for $\sqrt[3]{2}$, which begins

1, 3, 1, 5, 1, 1, 4, 1, …

are bounded. Furthermore, there are no known methods for attacking such problems.

Under quadratic irrationals, there are three categories according to the nature of their continued-fraction sequences. Reduced irrationals, which have special root properties that are important for other reasons, have purely periodic continued-fraction sequences — that is, they have no pre-periodic terms before the repeat.

Continued-fraction sequences for square roots have a particularly interesting form that we'll describe later. The "other" category contains everything else and at present we have no plans to explore it.

Rational Approximations to Square Roots

In order to compute continued-fractions sequences for quadratic irrationals, we need better rational approximations to them than provided by floating-point arithmetic [1].

Calculating Square Roots by Hand

RULE. — *Separate the number into periods of two figures each, beginning at units.*

Find the greatest square in the left-hand period, and write its root for the first figure of the required root.

Square this root, subtract the result from the left-hand period, and annex to the remainder the next period for a new dividend.

Double the root already found, with a cipher annexed, for a trial divisor, and by it divide the dividend. The quotient, or quotient diminished, will be the second figure of the root. Add to the trial divisor the figure last found, multiply this complete divisor by the figure of the root found, subtract the product from the dividend, and to the remainder annex the next period for the next dividend.

Proceed in this manner until all the periods have been used thus. The result will be the square root sought.

Editor's Note:

The quotation above is from a late 19th century algebra text book [1]. When I took algebra (which was a bit later), we were taught the method above for extracting square roots and a similar but more complicated method for extracting cube roots — and had to do seemingly endless examples in class. I could not see any sense to it; we got no understanding and only (mostly incorrect) results. Granted it gave integer results for perfect squares, but it seemed to me just like a way to fill hours that the teacher otherwise would have had to lecture.

It was depressing to learn later that a far better method of approximating square roots was known to the Babylonians.

With hand-held calculators now widely available and affordable, I suppose that taking roots long hand is no longer taught in high schools. I hope not.

However, the built-in root-extraction functions of even the best hand-held calculators are not good enough for applications like computing continued-fraction sequences.

— *reg*

Reference

1. *High School Algebra*, William J. Milne, American Book Co., NY, 1892, pp. 181-182.

The idea behind most methods of obtaining good rational approximations to real numbers is to find a way to bound the specified number above and below and then to bring the bounds closer and closer to the specified number.

For an example, consider $\sqrt{13}$. First note that if $x = \sqrt{13}$, $x^2 = 13$, and $x = 13/x$. If we have a number x' that is close to $\sqrt{13}$, then $\sqrt{13}$ will be close to half way between x' and $13/x'$.

Suppose we pick $x' = 3$ for our first approximation. The number half way between 3 and 3/13 is $(3 + 13/3)/2 = 11/3$. (It's important to reduce the result to lowest terms; otherwise the numerators and denominators get huge very quickly.) This is our second approximation. We continue with the number half way between 11/3 and 13/(11/3), which reduced to lowest terms, is 119/33. As this process continues, successive approximations get closer and closer to $\sqrt{13}$.

Picking a good first approximation is easy — integer(sqrt(x)), which is, in fact, 3 for $x = 13$. The first few rational approximations to $\sqrt{13}$ are:

```
3/1
11/3
119/33
14159/3927
200477279/55602393
40191139395243839/11147016454528647
```

Here's a procedure that generates successively better rational approximations for square roots:

```
link rational

procedure sqrtapprox(i)
   local x, half

   half := rational(1, 2, 1)

   x := rational(integer(sqrt(i)), 1, 1)

   i := rational(i, 1, 1)

   repeat {
      suspend x
      x := mpyrat(half, addrat(x, divrat(i, x, 1), 1))
      }

end
```

Of course, we can get a better first approximation by using more of the accuracy of sqrt(x). For example,

```
x := rational(integer(sqrt(i * 2 ^ 30)), 2 ^ 15, 1)
```

Quadratic Signatures

Not only are the continued-fraction sequences for quadratic irrationals periodic, but there is a one-to-one correspondence between quadratic irrationals and periodic sequences. Thus, every periodic sequence corresponds to a unique quadratic irrational.

Among other things, this means that every periodic sequence, of whatever origin, can be represented by three integers — the a, b, and c in

$$\frac{a + \sqrt{b}}{c}$$

This does not imply that a three-integer "quadratic signature" for a periodic sequence is in any sense simpler or shorter than the sequence. The integers may be very large.

We'll denote the quadratic signature of a quadratic irrational by <a, b, c>. Given a quadratic signature, we can compute its continued-fraction sequence as described in Reference 1 and find its repeat as described in Reference 2. A complete procedure is:

```
link approx
link genrfncs
link periodic
link rational

procedure sig2cf(a, b, c, limit)
   local n, m, j, rat, seq, results, line

   /limit := 5 * (log(b) * sqrt(b))

   every rat := sqrtapprox(b) do {
      seq := []
      rat := divrat(addrat(rational(a, 1, 1), rat),
         rational(c, 1, 1))
      every j := cfseq(rat.numer, rat.denom) \ limit do
         put(seq, j)
      results := repeater(seq, 1.5)
      if *results[2] ~= 0 then break
      }

   return results

end
```

The procedure sqrtapprox() is in the module approx, the procedure cfseq() is in the module genrfncs, and the procedure repeater() is in the module periodic.

There's a problem with the procedure above: the number of terms needed to get the repeat. Although it's known that there will be a repeat, it's

length can only be approximated The default approximation comes from a bound on the length of continued-fraction sequences for square roots..

Determining the quadratic signature for a periodic sequence is not so straightforward.

One approach is to work with the continued fraction for the sequence. Consider the sequence $\overline{1,2}$. Its continued fraction is

$$x = 1 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{\cdots}}}}$$

Grouping the terms at the end

$$x = 1 + \cfrac{1}{2 + \cfrac{1}{\left(1 + \cfrac{1}{2 + \cfrac{1}{\cdots}}\right)}}$$

we see the parenthesized portion is just x, so

$$x = 1 + \cfrac{1}{2 + \cfrac{1}{x}}$$

Now we have a closed form. Simplifying, we get

$$2x^2 - 2x - 1 = 0$$

By the quadratic formula, we get

$$x = \frac{1 \pm \sqrt{3}}{2}$$

So the quadratic signature for $\overline{1,2}$ is <1, 3, 2> (the positive root).

Although this method works for purely periodic sequences in general, the algebraic manipulations become tedious for long sequences.

There is an easier way using convergents, which we've already implemented [1]. If the ith convergent is denoted by

$$\frac{p_i}{q_i}$$

then for a purely periodic sequence with period n, the corresponding quadratic equation is

$$q_n x^2 - (p_n - q_{n-1})x - p_{n-1} = 0$$

so by the quadratic formula, the quadratic signature is

$$< p_n - q_{n-1},\ (p_n - q_{n-1})^2 + 4q_n p_{n-1},\ 2q_n >$$

Note that

$$b = a^2 + 4q_n p_{n-1}$$

Here's a program that computes the quadratic signature of a purely periodic sequence whose repeat is read from standard input.

```
link converge

procedure main()
   local conv, seq, convergents, p1, p2, q1, q2, a

   seq := []

   convergents := []

   while put(seq, read())

   every conv := converge(seq) do
      put(convergents, conv)

   p2 := convergents[–1][1]      # last convergent
   q2 := convergents[–1][2]
   p1 := convergents[–2][1]      # next-to-last
   q1 := convergents[–2][2]

   # Calculate values for (a + sqrt(b)) / c.

   writes("<", a := p2 – q1, ",")
   writes(x ^ 2 + 4 * q2 * p1, ",")
   write(2 * q2, ">")

end
```

Sequences for Square Roots

The continued-fraction sequences for square roots, where $a = 0$ and $c = 1$, have a particularly interesting form.

We can characterize square roots of positive integers that are not perfect squares by

$$\sqrt{n^2 + m} \qquad n = 1, 2, \ldots ; \quad 1 \le m \le 2n$$

(If $m = 2n + 1$, this expression reduces to a perfect square, $\sqrt{(n+1)^2}$.)

The continued fraction sequence for $\sqrt{n^2 + m}$ has the form

$$n, \overline{a_1, a_2, \ldots a_2, a_1, 2n}$$

where, as indicated, the part of the repeat prior to $2n$ is a palindrome. Such palindromes may or may not have a middle term. Furthermore, $1 \le a_i \le n$. If there is a middle term, it can be as large as n, but other terms in such palindromes apparently are less than n (the largest we've found is about $0.7n$).

Figure 2 on the next page lists the palin-

n	m	palindrome
1	1	
1	2	1
2	1	
2	2	2
2	3	1,1,1
2	4	1
3	1	
3	2	3
3	3	2
3	4	1,1,1,1
3	5	1,2,1
3	6	1
4	1	
4	2	4
4	3	2,1,3,1,2
4	4	2
4	5	1,1,2,1,1
4	6	1,2,4,2,1
4	7	1,3,1
4	8	1
5	1	
5	2	5
5	3	3,2,3
5	4	2,1,1,2
5	5	2
5	6	1,1,3,5,3,1,1
5	7	1,1,1
5	8	1,2,1
5	9	1,4,1
5	10	1
6	1	
6	2	6
6	3	4
6	4	3
6	5	2,2
6	6	2
6	7	1,1,3,1,5,1,3,1,1
6	8	1,1,1,2,1,1,1
6	9	1,2,2,2,1
6	10	1,3,1,1,2,6,2,1,1,3,1
6	11	1,5,1
6	12	1
7	1	
7	2	7
7	3	4,1,2,1,4
7	4	3,1,1,3
7	5	2,1,6,1,2
7	6	2,2,2
7	7	2
7	8	1,1,4,1,1
7	9	1,1,1,1,1,1
7	10	1,2,7,2,1
7	11	1,2,1
7	12	1,4,3,1,2,2,1,3,4,1
7	13	1,6,1
7	14	1
8	1	
8	2	8
8	3	5,2,1,1,7,1,1,2,5
8	4	4
8	5	3,3,1,4,1,3,3
8	6	2,1,2,1,2
8	7	2,2,1,7,1,2,2
8	8	2
8	9	1,1,5,5,1,1
8	10	1,1,1,1
8	11	1,1,1
8	12	1,2,1,1,5,4,5,1,1,2,1
8	13	1,3,2,3,1
8	14	1,4,1
8	15	1,7,1
8	16	1
9	1	
9	2	9
9	3	6
9	4	4,1,1,4
9	5	3,1,1,1,8,1,1,1,3
9	6	3
9	7	2,1,1,1,2
9	8	2,3,3,2
9	9	2
9	10	1,1,5,1,5,1,1
9	11	1,1,2,4,2,1,1
9	12	1,1,1,4,6,4,1,1,1
9	13	1,2,3,1,1,5,1,8,1,5,1,1,3,2,1
9	14	1,2,1
9	15	1,3,1
9	16	1,5,1,1,1,1,1,1,5,1
9	17	1,8,1
9	18	1
10	1	
10	2	10
10	3	6,1,2,1,1,9,1,1,2,1,6
10	4	5
10	5	4
10	6	3,2,1,1,1,1,2,3
10	7	2,1,9,1,2
10	8	2,1,1,4,1,1,2
10	9	2,3,1,2,4,1,6,6,1,4,2,1,3,2
10	10	2
10	11	1,1,6,1,1
10	12	1,1,2,1,1
10	13	1,1,1,2,2,1,1,1
10	14	1,2,10,2,1
10	15	1,2,1,1,1,1,1,2,1
10	16	1,3,2,1,4,1,2,3,1
10	17	1,4,2,4,1
10	18	1,6,3,2,10,2,3,6,1
10	19	1,9,1
10	20	1
11	1	
11	2	11
11	3	7,2,1,1,1,3,1,4,1,3,1,1,1,2,7
11	4	5,1,1,5
11	5	4,2,4
11	6	3,1,2,2,7,11,7,2,2,1,3
11	7	3,5,3
11	8	2,1,3,1,6,1,3,1,2
11	9	2,2
11	10	2,4,11,4,2
11	11	2
11	12	1,1,7,5,1,1,1,2,1,1,1,5,7,1,1
11	13	1,1,2,1,3,1,10,1,3,1,2,1,1
11	14	1,1,1,1,1,1,1
11	15	1,1,1
11	16	1,2,2,1,1,2,2,1
11	17	1,2,1
11	18	1,3,1,3,7,1,1,2,11,2,1,1,7,3,1,3,1
11	19	1,4,1
11	20	1,6,1
11	21	1,10,1
11	22	1
12	1	
12	2	12
12	3	8
12	4	6
12	5	4,1,5,3,3,5,1,4
12	6	4
12	7	3,2,7,1,3,4,1,1,1,11,1,1,1,4,3,1,7,2,3
12	8	3
12	9	2,1,2,2,2,1,2
12	10	2,2,3,1,2,1,3,2,2
12	11	2,4,2
12	12	2
12	13	1,1,7,1,5,2,1,1,1,1,2,5,1,7,1,1
12	14	1,1,3,12,3,1,1
12	15	1,1,1,1,3,1,1,1,1
12	16	1,1,1,5,1,1,1
12	17	1,2,4,1,2,1,4,2,1
12	18	1,2,1,2,12,2,1,2,1
12	19	1,3,3,2,1,1,7,1,11,1,7,1,1,2,3,3,1
12	20	1,4,6,4,1
12	21	1,5,2,5,1
12	22	1,7,1,1,1,2,4,1,3,2,12,2,3,1,4,2,1,1,1,7,1
12	23	1,11,1
12	24	1
13	1	
13	2	13
13	3	8,1,2,2,1,1,3,6,3,1,1,2,2,1,8
13	4	6,1,1,6
13	5	5,4,5
13	6	4,2,1,2,4
13	7	3,1,3
13	8	3,3,2,8,2,3,3
13	9	2,1,12,1,2
13	10	2,1,1,1,3,5,13,5,3,1,1,1,2
13	11	2,2,2
13	12	2,4,1,8,6,1,1,1,1,2,2,1,1,1,1,6,8,1,4,2
13	13	2
13	14	1,1,8,1,1
13	15	1,1,3,2,1,2,1,2,3,1,1
13	16	1,1,1,1
13	17	1,1,1,3,4,3,1,1,1
13	18	1,2,13,2,1
13	19	1,2,2,6,2,2,1
13	20	1,2,1
13	21	1,3,1,1,1,2,2,2,1,1,1,3,1
13	22	1,4,1,1,3,2,2,13,2,2,3,1,1,4,1
13	23	1,5,1
13	24	1,8,3,2,1,3,3,1,2,3,8,1
13	25	1,12,1
13	26	1
14	1	
14	2	14
14	3	9,2,1,2,2,5,4,1,1,13,1,1,4,5,2,2,1,2,9
14	4	7
14	5	5,1,1,1,2,1,8,1,2,1,1,1,5
14	6	4,1,2,2,1,4
14	7	4
14	8	3,1,1,6,1,1,3
14	9	3,6,1,4,1,6,3
14	10	2,1,5,14,5,1,2
14	11	2,1,1,2,1,1,2
14	12	2,2,1,2,2
14	13	2,5,3,2,3,5,2
14	14	2
14	15	1,1,9,5,1,2,2,1,1,4,3,1,13,1,3,4,1,1,2,2,1,5,9,1,1
14	16	1,1,3,1,1,1,6,1,1,1,3,1,1
14	17	1,1,2,6,1,8,1,6,2,1,1
14	18	1,1,1,2,3,1,4,9,1,1,5,3,14,3,5,1,1,9,4,1,3,2,1,1,1
14	19	1,1,1
14	20	1,2,3,2,1
14	21	1,2,1,2,1,1,9,4,9,1,1,2,1,2,1
14	22	1,3,3,1
14	23	1,3,1
14	24	1,4,1
14	25	1,6,2,6,1
14	26	1,8,1
14	27	1,13,1
14	28	1
15	1	
15	2	15
15	3	10
15	4	7,1,1,7
15	5	6
15	6	5
15	7	4,3,7,3,4
15	8	3,1,3,1,1,1,1,3,1,3
15	9	3,2,1,2,1,2,3
15	10	3
15	11	2,1,3,5,1,6,1,5,3,1,2
15	12	2,1,1,7,10,7,1,1,2
15	13	2,2,1,14,1,2,2
15	14	2,5,1,2,4,15,4,2,1,5,2
15	15	2
15	16	1,1,9,1,5,3,3,1,1,3,3,5,1,9,1,1
15	17	1,1,3,1,14,1,3,1,1
15	18	1,1,2,3,15,3,2,1,1
15	19	1,1,1,1,1,2,1,5,1,1,9,1,6,1,9,1,1,5,1,2,1,1,1,1,1
15	20	1,1,1,7,6,7,1,1,1
15	21	1,2,5,1,14,1,5,2,1
15	22	1,2,1,1,9,1,9,1,1,2,1
15	23	1,2,1
15	24	1,3,1,1,5,1,3,10,3,1,5,1,1,3,1
15	25	1,4,3,3,4,1
15	26	1,5,2,1,2,2,15,2,2,1,2,5,1
15	27	1,6,1
15	28	1,9,1,1,1,2,1,7,4,2,2,2,4,7,1,2,1,1,1,9,1
15	29	1,14,1
15	30	1

Figure 2. Palindromes for $\sqrt{n^2+m}$

dromes though $n = 15$.

One natural question is whether there are any interesting patterns in the palindromic parts of such sequences. Indeed there are. Here are some that depend on m:

m	*palindrome for* $\sqrt{n^2+m}$	
1	*empty*	
2	n	
$n/2$	4	n even
n	2	
$2n-1$	$1, n-1, 1$	
$2n$	1	

These and other relationships can be proved. The idea is to convert the continued fraction to a quadratic equation and find its positive root.

Consider, for example, the case for $m = 1$, that is $\sqrt{n^2+1}$. The palindromes in Figure 2 strongly suggest that the palindromic part is empty and that the continued fraction sequence is $n, \overline{2n}$. This continued-fraction sequence corresponds to the continued fraction

$$x = n + \cfrac{1}{2n + \cfrac{1}{2n + \cfrac{1}{\cdots}}}$$

Now we use some insight (a "trick"). We'd like to make the right-hand side of the continued fraction uniform so that it can be interpreted as a recursive structure. To get this, we add n to both sides, giving

$$x + n = 2n + \cfrac{1}{2n + \cfrac{1}{2n + \cfrac{1}{\cdots}}}$$

Now, grouping terms

$$x + n = 2n + \cfrac{1}{\left(2n + \cfrac{1}{2n + \cfrac{1}{\cdots}}\right)}$$

we see the expression in parentheses is the same as the whole left-hand side. So we can substitute the right-hand side for it:

$$x + n = 2n + \cfrac{1}{x + n}$$

This simplifies to

$$x^2 - n^2 - 1 = 0$$

or

$$x = \sqrt{n^2+1}$$

which proves our conjecture.

This approach works in general, although the algebra quickly gets out of hand as the lengths of continued-fraction sequences increase. We'll have more to say about this from a programming standpoint later.

What's to Come?

The palindromic parts of continued-fraction sequences for square roots provide a veritable garden of patterns. There are all kinds of questions one might ask, such as

- Are there palindromes that never occur?
- Some palindromes appear infinitely often for different values on n. Do all?
- What kinds of patterns appear in the palindromes and how to they depend on n and m?
- How do we find such relationships?
- Are there any patterns to the lengths of the palindromes?
- Are there any patterns to the central terms in odd-length palindromes?
- Is there anything special about first terms? Second terms? Terms in other positions?
- How large can the terms be?
- Do factors and primes enter into the patterns in the palindromes?

We recall writing the first article on versum numbers with no expectation that there would be more than one or two subsequent articles. Then we discovered a wealth of material, which led to 15 articles in all and could have gone on indefinitely if we'd not decided it was time to get on to other things.

The patterns in the palindromic parts of the continued-fraction sequences for square roots appear to offer more promise than those related to versum numbers. But with only five issues of the Analyst to go, we can't be led too far down the (pattern) garden path.

References

1. "Continued Fractions", Icon Analyst 60, pp. 1-5.

2. "Finding Repeats", Icon Analyst 57, pp. 7-11.

Creating Weavable Color Patterns

In previous articles on weavable color patterns [1-3], we described how to determine if a color pattern is weavable and, if so, how to create a draft for it.

In this article, we'll look at weavable color patterns from a different perspective: how to create color patterns that are guaranteed to be weavable. This article is a precursor to an article on an interactive application for creating weavable color patterns.

Much of the material that follows is basic and in some cases obvious. We're presenting it here to provide a foundation for what follows.

We'll treat color patterns as $i \times j$ arrays of rectangular colored cells. An array in which every column and row is labeled with a different color is called *nonredundant*.

One question is how many cells are needed to create a weavable 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. This case, however, is degenerate and uninteresting.

In general, to have k colors in a nonredundant array, there must be $i + j = k$ columns and rows with different colors assigned to each one. If $i + j > k$, more complex patterns are possible, but we'll stick to the minimum size to begin with and look at the case of redundant arrays with duplicate color assignments later.

For nonredundant arrays, k is partitioned into two parts. There are $k - 2$ non-degenerate size combinations, given by

$$i = k - j \qquad 2 \le j \le k - 2$$

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

ftp.cs.arizona.edu (cd /icon)

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 we have eight colors and a 4×4 array as shown in Figure 1.

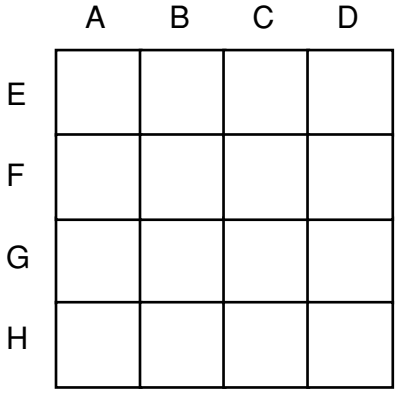


Figure 1. A 4×4 Array

It is obvious that it's possible to have all k colors in such an array. Figure 2 shows one such pattern.

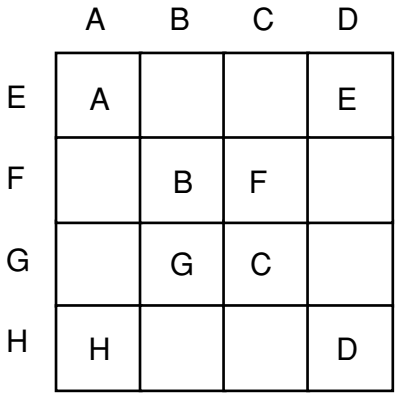


Figure 2. An 8-Color 4×4 Pattern

The remaining cells in Figure 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 color and the row colors.
- For each even-numbered row, assign alternate cells the row and column colors.

Figure 3 on the next page shows the result for a 4×4 array.

	A	B	C	D
E	A	E	C	E
F	F	B	F	D
G	A	G	C	G
H	H	B	H	D

Figure 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 4 for an example, but also view the color image on the Web page for this issue of the Analyst.

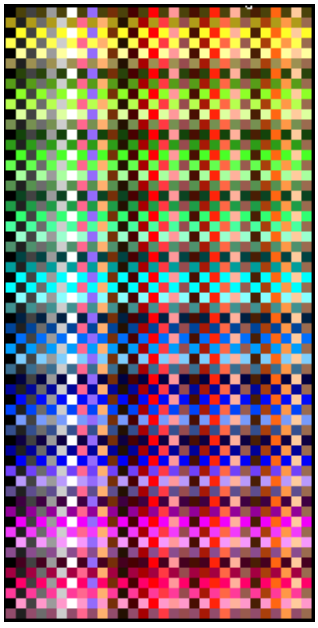


Figure 4. An Algorithmic Pattern

Implementing the algorithm to produce, say, image strings for such patterns is straightforward. Here's a procedure that produces an image string for a *k*-colored pattern using the column and row keys from a palette given as arguments:

```
procedure kcolor(cols, rows, palette)
   local i, j, ims

   ims := *cols || "," || palette || ","

   every j := 1 to *rows do
      every i := 1 to *cols by 2 do
         ims ||:= if j % 2 = 1 then cols[i] || rows[j]
            else rows[j] || cols[i + 1]

   return ims

end
```

This procedure is straightforward, if a bit uninteresting. There is a more interesting (aka obscure) approach that is based on the pattern of characters produced by the algorithm.

Odd- and even-numbered rows have different but complementary patterns. In Figure 3, the strings for the odd-numbered rows are AECE and AGCG, respectively, while for the even-numbered rows, they are FBFD and HBHD. These strings are collations (interleavings) of characters from strings of column and row labelings. The odd-numbered rows can be produced by

```
collate("AC", "EE")
```

and

```
collate("AC", "GG")
```

respectively, while the even-numbered rows can be produced by

```
collate("FF", "BD")
```

and

```
collate("HH", "BD")
```

It's clear that this pattern of construction generalizes for larger patterns.

The arguments of collate() are of two forms: replications and alternate characters (decollations) of the column and row strings.

Here's a procedure that produces image strings using this approach:

```
link strings

procedure kcolor(cols, rows, palette)
   local i, j, ims, width, height

   ims := *cols || "," || palette || ","

   width := *cols
   height := *rows

   every i := 1 to *rows by 2 do
      ims ||:= collate(decollate(cols, i),
         repl(rows[i], *cols / 2)) ||
            collate(repl(rows[i + 1],  *cols / 2),
               decollate(cols, i + 1))

   return ims

end
```

The procedures collate() and decollate() are from the strings module in the Icon program library. The result produced by decollate() is the odd- or even-numbered characters of its first argument depending on the parity of its second argument.

There's a catch. The procedure above only works for patterns in which the number of columns, *i*, is even. We can fix this by adding an extra, dummy column if *i* is odd and then trimming off the unwanted portions during the construction process. (If the number of rows is odd, the loop bound assures the correct number of rows.) Figure 5 shows how a 3×4 pattern can be carved out of a 4×4 pattern.

	A	B	C	D
E	A	E	C	E
F	F	B	F	D
G	A	G	C	G
H	H	B	H	D

Figure 5. A 3×4 Pattern Within a 4×4 Pattern

The label of the added column is, of course, irrelevant.

Here's a procedure that implements the approach described above.

```
procedure kcolor(cols, rows, palette)
   local i, j, ims, width

   ims := *cols || "," || palette || ","

   width := *col

   if *cols % 2 ~= 0 then cols ||:= "~"     # dummy

   every i := 1 to *rows by 2 do
      ims ||:= left(collate(decollate(cols, i),
         repl(rows[i], *cols / 2)), width) ||
            left(collate(repl(rows[i + 1], *cols / 2),
               decollate(cols, i + 1)), width)

   return ims

end
```

Suppose you encountered this procedure when, say, strolling in a park. Would you have any clue as to what it's for?

Transformations that Preserve Weavability

Given a weavable color pattern, there are several kinds of changes that can be made to it that preserve weavability:

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 diagonally
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

One of the consequences of 1 and 3 is that the mirror symmetry [4] of a weavable pattern is weavable.

A Program for Experimenting

We put together a simple program for experimenting with various transformations that preserve weavability.

The program allows the user to specify transformations interactively, but user input is taken from the command line instead of from a visual interface. This design is somewhat crude, but it's easier than building an application with a visual interface. A visual interface can be added later if desired.

Although user input comes from the command line, dialogs are used to request information from the user. This avoids command-line input with complicated syntax.

User Input

Commands are one-character strings that are chosen to have mnemonic value where possible. For example, `s` means save, `q` means quit, and `>` means rotate 90° clockwise. (The use of one-character commands also facilitates conversion to a program with a visual interface, where the characters

are keyboard shortcuts.)

Data Representation

Patterns are represented by image strings and changes are made to a pattern by making changes to its image string. To facilitate manipulation of image strings, image records are used [5].

A window displaying the pattern is drawn from the current image string.

Program Structure

The main procedure contains a loop in which user commands are read and processed. A case expression selects a procedure that carries out the command.

Program Listing

Here's a listing of selected portions of the program. The complete program is on the Web site for this issue of the Analyst.

```
link graphics
link imrutils
link imsutils
link imxform
link interact
link lists

global imr              # image string record
global pattern          # image window
global stack            # saved image string records

procedure main()
   local command

   stack := []

   while command := read() do {
      "!"      : shuffle_cols()
      ":"      : shuffle_rows()
      "$"      : swap_cols()
      "#"      : swap_rows()
      "="      : colscaleimr()
      "*"      : rowscaleimr()
      "+"      : zoom_in()
      "-"      : zoom_out()
      "h"      : flip_horizontal()
      "v"      : flip_vertical()
      "/"      : flip_left()
      "\\"     : flip_right()
      "|"      : r180()
      "<"      : rccw()
      ">"      : rcw()
      "i"      : info()
      "l"      : load_pattern()
      "m"      : mirror()
      "q"      : exit()
      "r"      : read_ims()
      "s"      : snapshot()
      "u"      : undo()
      "w"      : write_ims()
                    …
      default  : Notice("Invalid command.")
      }
   }

end

procedure ScaleDialog(length)
   local slist

   if OpenDialog("Scaling list:") == "Cancel" then fail

   slist := []

   dialog_value ? {
      while tab(upto(&digits)) do
         put(slist, tab(many(&digits)))
      }

   if *slist = 0 then slist := list(length, 2) else
      slist := lextend(slist, length)

   return slist

end

procedure check_imr()

   push(stack, \imr) | {
      Notice("No image.")
      fail
      }

   return

end

procedure colscaleimr()
   local row, pixels, i, width, slist

   check_imr() | fail

   slist := ScaleDialog(imr.width) | fail

   pixels := ""

   width := 0

   every width +:= !slist

   imr.pixels ? {
      while row := move(imr.width) do
         every i := 1 to imr.width do
            pixels ||:= repl(row[i], slist[i])
      }

   imr.pixels := pixels
   imr.width := width
```

```
   redraw()
   return
end
procedure flip_horizontal()
   check_imr() | fail
   imr := imrfliph(imr)
   redraw()
   return
end
procedure info()
   check_imr() | fail
   Notice("Size=" || imr.width || "x" ||
      *imr.pixels / imr.width, "Colors=" ||
         *cset(imr.pixels))
   return
end
procedure load_pattern()
   repeat {
      if OpenDialog("Load image:") == "Cancel"
         then fail
      WClose(\pattern)
      pattern := WOpen("image=" || dialog_value) | {
         Notice("Cannot open image: ", dialog_value)
         next
         }
      break
      }
   imr := imstoimr(Capture(pattern))
   return
end
procedure mirror()
   local pixels, row
   pixels := ""
   imr.pixels ? {
      while row := move(imr.width) do
         pixels ||:= row || reverse(row)
      }
   imr.pixels := ""
   pixels ? {
      while imr.pixels := move(imr.width) ||
         imr.pixels
      }
   imr.pixels := pixels || imr.pixels
   imr.width *:= 2
   redraw()
   return
end
procedure read_ims()
   local input
   repeat {
      if OpenDialog("Read image string:") == "Cancel"
         then fail
      input := open(dialog_value) | {
         Notice("Cannot read image string file.")
         next
         }
      imr := imstoimr(input) | {
         Notice("Invalid image string.")
         next
         }
      break
      }
   close(input)
   redraw()
end
procedure redraw()
   WAttrib(pattern, "width=" || imr.width)
   WAttrib(pattern, "height=" ||
      (*imr.pixels / imr.width))
   imrdraw(pattern, 0, 0, imr)
   return
end
```

...

Comments

The structure of the program makes it easy to add features experimentally, although as you can see, the limitation to one-character commands eventually leads to unintuitive labeling of operations.

One of the most interesting operations is "scaling" (= and *), in which the user can specify duplication of each column or row by values given in a dialog box. Figure 6 shows a pattern created using scaling and mirroring. Again, see the color version on the Web site for this issue of the Analyst.

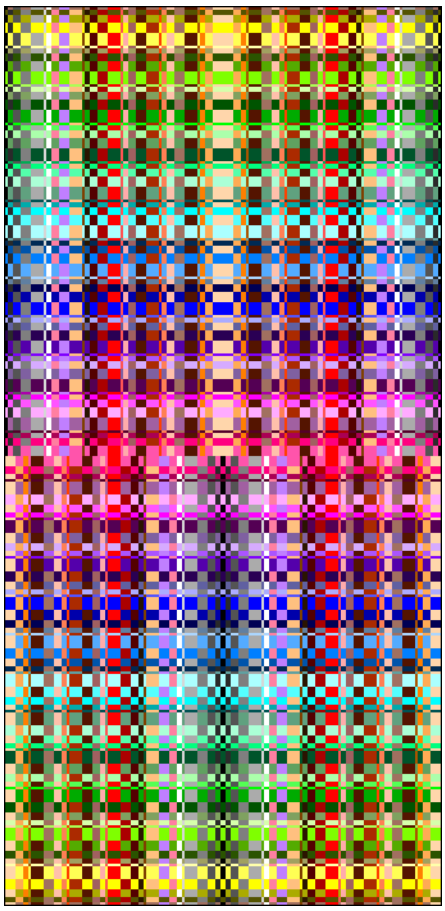


Figure 6. A Weavable Color Pattern

Next Time

In the next article on weavable color patterns, we'll describe an application that allows the user to construct weavable color patterns interactively, but inways different from those described in this article.

References

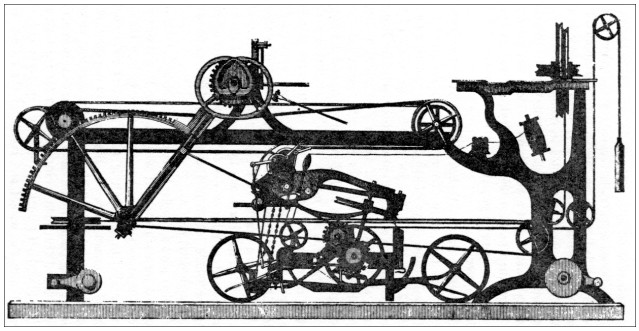
1. "Weavable Color Patterns", Icon Analyst 58, pp. 7-10.

2. "Weavable Color Patterns", Icon Analyst 59, pp. 10-15.

3. "Drafting Weavable Color Patterns", Icon Analyst 60, pp. 6-9.

4. "Graphics Corner — Seamless Tiling", Icon Analyst 45, pp. 10-12.

5. "Graphics Corner — Fun with Image Strings", Icon Analyst 50, pp. 10-13.



Perhaps if we wrote programs from childhood on, as adults we'd be able to read them. However, reading a program is not like reading a book, it is more like being a psychiatrist to a recumbent patient.

— Alan Perlis

What's Coming Up

With only five more issues of the Analyst remaining, we've revised our plans for future articles. We don't have everything thought out yet, but there are some articles we plan to include that may cause the exclusion of articles we'd planned earlier.

We didn't finish the planned article on derived tie-ups in time for this issue, but we expect to have it in a future issue, along with articles on geometric and motif tie-ups. The article on adaptive name drafting is problematical at this time.

We ran out of room before completing the article on polygram substitution, but it's scheduled for the next issue and we expect to conclude the series on classical cryptography in subsequent issues.

In the series of articles on sequences, we'll continue with continued fractions and fractal sequences. We also expect to have articles on base expansions of fractions, as well as articles on packet sequences and template sequences.

We have one more article on creating weavable color patterns that could turn into two articles depending on how our work in that area goes.

We also hope to have one final article on versum sequences that puts the problem in perspective. We'll also try to work in another article on digit patterns in primes.

We have some new subjects staged, including one called "Sigma Quest".