The Icon Analyst

In-Depth Coverage of the Icon Programming Language and Applications

October 2000
Number 62

In this issue

Square Root Palindromes

In the previous article on continued fractions for quadratic irrationals [1], we started to explore the continued fraction sequences for square roots. These sequences have the form

$$n, \overline{p, 2n}$$

where p is a palindromic sequence.

In this article, we'll explore the properties of p and answer some of the questions posed in the previous article. We'll refer to p as a square-root palindrome.

Exploring the Square-Root Palindromes

Patterns and relationships in square-root palindromes can be found by inspecting examples. To get very far, a large table is needed, since many of the patterns are subtle and occur only for square roots with special properties.

Programming help is needed for such an enterprise. Our first idea was a program the operated on a database of previously computed square-root palindromes and looked for instances of patterns specified by the user.

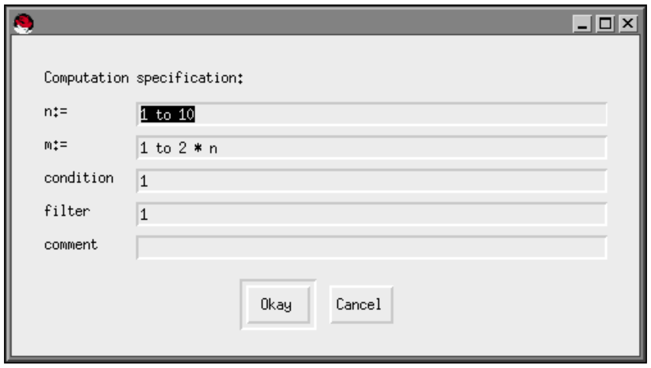
This approach turned out to be impractical. A database that contains all the palindromes for $1 \leq k \leq 100$ has 10,100 entries. Yet to find many patterns, it is necessary to go to values of k much larger than 100.

On the other hand, many such searches, while deep in terms of k, are narrow, exploring only a tiny portion of the space of square-root palindromes for k.

We found it more practical to compute the palindromes for searches rather than to look them up.

Since the application is specialized and not of general interest, we took the approach we used for exploring weavable color patterns [2]. Instead of a full interface, we designed an application to take instructions from the command line but used dialogs for requesting user input and displaying results.

Searches take the form of specifications for n and m with optional constraints. Figure 1 shows the specification dialog with the default values.

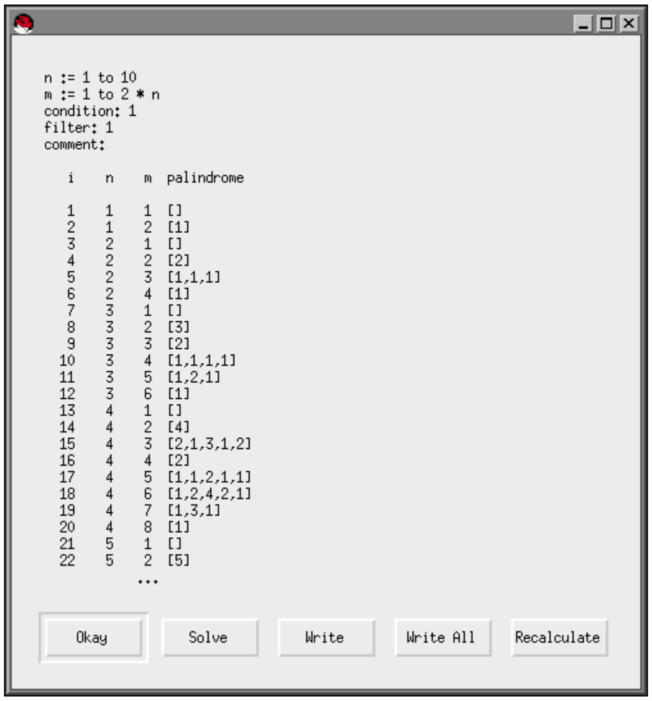

Computation specification:

n:=	1 to 10
m:=	1 to 2 * n
condition	1
filter	1
comment	

Figure 1. Specification Dialog

The condition field allows the user to specify an Icon expression to limit the values of n and m for which palindromes are computed. The filter field provides for an expression that accepts or rejects the results to be returned. The values of 1 in Figure 1 simply impose no constraints and allow all results to be accepted.

Figure 2 shows the results for the specification in Figure 1.



```
n := 1 to 10
m := 1 to 2 * n
condition: 1
filter: 1
comment:

  i    n    m  palindrome

  1    1    1  []
  2    1    2  [1]
  3    2    1  []
  4    2    2  [2]
  5    2    3  [1,1,1]
  6    2    4  [1]
  7    3    1  []
  8    3    2  [3]
  9    3    3  [2]
 10    3    4  [1,1,1,1]
 11    3    5  [1,2,1]
 12    3    6  [1]
 13    4    1  []
 14    4    2  [4]
 15    4    3  [2,1,3,1,2]
 16    4    4  [2]
 17    4    5  [1,1,2,1,1]
 18    4    6  [1,2,4,2,1]
 19    4    7  [1,3,1]
 20    4    8  [1]
 21    5    1  []
 22    5    2  [5]
           ...
```

Figure 2. Result Dialog

If there are more results than shown in the window, ellipses are given at the end as shown.

The Write and Write All buttons allow the results to be written to a file, either just the results shown or all results, respectively.

The Recalculate button brings up the search dialog again for possible modification. It is typical for a search to produce results that are suggestive but not quite what are wanted. Cycling between search and result dialogs allows refinements that may lead to a desired result.

The Solve button uses the method of differences [3] to attempt to derive formulas for *n* and *m*.

Conditions and Filters

Conditions generally relate to the values of *n* and *m*. For example, the condition

```
n % m = 0
```

limits the values of *n* and *m* to those for which *m* is a factor of *n*.

Filtering expressions require a knowledge of the variables and procedures that are available. For example, the sequence to be filtered is the value of the variable pal and eq() is a procedure that compares a sequence to pal. For example,

```
*pal = 3
```

accepts only sequences of length 3 and

```
eq([1,2,1])
```

accepts only sequences of the form 1, 2, 1. Such a filter might be used to determine what values of *n* and *m* produce the sequence 1, 2, 1.

If a term in the sequence argument to eq() is omitted, it is considered to be a "don't care" and matches any term in the corresponding position of the other argument. For example,

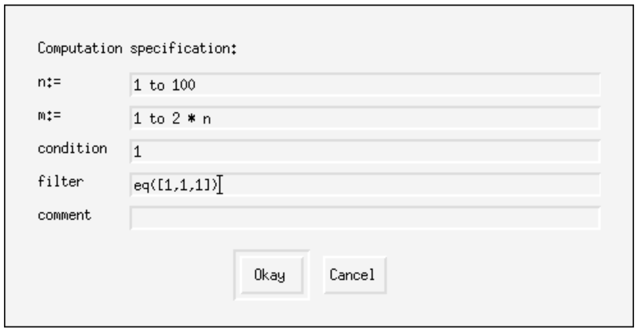
```
eq([1, , 1])
```

accepts any 3-term sequence that begins and ends with a 1.

Finding Formulas

Deriving formulas for *n* and *m* by hand is tedious and error-prone, yet such formulas are essential to discovering patterns in square-root palindromes.

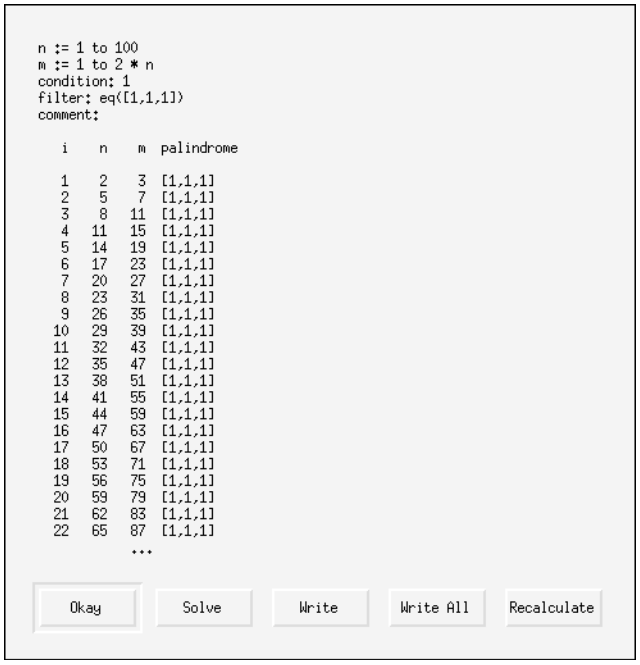
As an example, to find formulas for *n* and *m* that produce the palindrome 1, 1, 1, a specification such as shown in Figure 3 could be used. The results are shown in Figure 4.



Computation specification:

n:=	1 to 100
m:=	1 to 2 * n
condition	1
filter	eq([1,1,1])
comment	

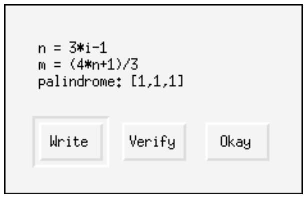
Figure 3. A Specification Dialog for 1,1,1



```
n := 1 to 100
m := 1 to 2 * n
condition: 1
filter: eq([1,1,1])
comment:

  i    n    m  palindrome

  1    2    3  [1,1,1]
  2    5    7  [1,1,1]
  3    8   11  [1,1,1]
  4   11   15  [1,1,1]
  5   14   19  [1,1,1]
  6   17   23  [1,1,1]
  7   20   27  [1,1,1]
  8   23   31  [1,1,1]
  9   26   35  [1,1,1]
 10   29   39  [1,1,1]
 11   32   43  [1,1,1]
 12   35   47  [1,1,1]
 13   38   51  [1,1,1]
 14   41   55  [1,1,1]
 15   44   59  [1,1,1]
 16   47   63  [1,1,1]
 17   50   67  [1,1,1]
 18   53   71  [1,1,1]
 19   56   75  [1,1,1]
 20   59   79  [1,1,1]
 21   62   83  [1,1,1]
 22   65   87  [1,1,1]
           ...
```

Figure 4. The Results

Clicking on the Solve button leads to the results shown in Figure 5.



```
n = 3*i-1
m = (4*n+1)/3
palindrome: [1,1,1]
```

Figure 5. Formulas for 1, 1, 1

Clicking on the Verify button uses these formulas without a condition or filter for a new search to assure that the formulas produce the expected results.

Results

One of the questions we posed in the previous article [1], rephrased in the current terminology, was "Are there any palindromic sequences that are not square-root palindromes?"

A little exploration suggests that 1,1 is not a square-root palindrome. Taking this as a conjecture, it's straightforward, if tedious, to solve the continued fraction

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

The result is

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

The numerator in the fraction is odd for all n and consequently there are no values for which the fraction is an integer. Hence 1,1 is not a square-root palindrome.

We know from the results shown in the previous article that some square-root palindromes occur infinitely often. Two simple examples are the "unit" palindromes 2 for $m = n$ and 1 for $m = 2n$.

Some exploration shows that there are many others. For example, 1, 1, 1 occurs for

$n = 3i - 1$ and $m = (4n + 1)/3$ i = 1, 2, 3, …

Again, the proof is straightforward.

Unit Palindromes

Intelligent guidance is the key to the successful use of the application like the one described here. What to look for?

We decided to explore unit palindromes, which appear scattered throughout square-root palindromes.

We were intrigued to discover that for a particular n, the unit palindromes occur for values of $m > 1$ that evenly divide $2n$. For example, for $n = 15$, the unit palindromes are:

m	*unit palindrome*	*product*
2	15	30
3	10	30
5	6	30
6	5	30
10	3	30
15	2	30
30	1	30

Again, the proof is straightforward. For unit palindrome k, the continued fraction is

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

for which the solution is

$$x = \pm\sqrt{\frac{2n}{k} + n^2}$$

Thus,

$$\frac{2n}{k}$$

must be an integer; in other words, k must divide $2n$ evenly.

It follows from the result above that all positive integers occur infinitely often as square-root unit palindromes.

Of course, a mathematician probably would see this relationship immediately.

It might appear that the unit palindrome 30 should appear in the list above; after all $k = 2n$ produces an integer for the fraction. But terms in square-root palindromes cannot be larger than n. The source of this apparent contradiction is the violation of an implicit assumption made in converting an infinite continued fraction to closed form [1]. Substituting $2n$ for k in the continued fraction above gives

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

which should be treated as

$$x = n + \frac{1}{2n + \ldots}$$

in the conversion — that is, as the empty palindrome.

Other Results

Here are some other results found using the application described above. In all of these, $i = 1, 2, 3, \ldots$.

Palindromes with Constant Terms

palindrome	*n*	*m*
empty	*i*	1
1	*i*	2*n*
2	*i*	*n*
4	2(*i*+1)	*n*/2
6	3(*i*+1)	*n*/3
8	4(*i*+1)	*n*/4
2,2	5*i*+1	(4*n*+1)/5
4,4	17*i*+2	(8*n*+1)/17
6,6	37*i*+3	(12*n*+1)/37
8,8	65*i*+4	(16*n*+1)/65
1,1,1	3*i*–1	(4*n*+1)/3
1,2,1	2*i*+1	(3*n*+1)/2
1,3,1	5*i*–1	(8*n*+3)/5
1,4,1	3*i*+2	(5*n*+2)/3
1,5,1	7*i*–1	(12*n*+5)/7
1,6,1	4*i*+3	(7*n*+3)/4
1,7,1	9*i*–1	(16*n*+7)/9
1,8,1	5*i*+4	(9*n*+4)/5
1,9,1	11*i*–1	(20*n*+9)/11
1,10,1	6*i*+5	(11*n*+5)/6
1,11,1	13*i*–1	(24*n*+11)/13
1,12,1	7*i*+6	(13*n*+6)/7
1,37,1	39*i*–1	(76*n*+37)/39
1,38,1	20*i*+19	(39*n*+19)/20
3,2,3	12*i*–7	(7*n*+1)/12
3,3,3	3(11*i*—5)	(20*n*+3)/33
4,4,4	2(18*i*+1)	(17*n*+2)/36
1,1,1,1	5*i*–2	2(3*n*+1)/5
1,3,3,1	17*i*–3	2(13*n*+5)/17
1,5,5,1	37*i*–4	2(31*n*+13)/37
1,2,1,2,1	15*i*+1	2(11*n*+4)/15
1,2,3,2,1	33*i*–19	2(23*n*+8)/33
1,1,1,1,1,1	13*i*–6	(16*n*+5)/13
1,1,1,1,1,1,1	21*i*–10	2(13*n*+4)/21
1,1,1,1,1,1,1,1,1	55*i*–27	(68*n*+21)/55

Back Issues

Back issues of The Icon Analyst are available for $5 each. This price includes shipping in the United States, Canada, and Mexico. Add $2 per order for airmail postage to other countries.

Palindromes with One Variable Term

palindrome	*n*	*m*	*x*
x	*ki*	*k*	*n*/*k*
x	*i*+1	2	*n*
1,*x*,1	*i*+1	(2*n*–1)	(*n*–1)
2,*x*,2	5*i*+2	(*n*–1)	2(*n*–2)/5
2,*x*,2	9*i*+4	(*n*–2)	2(*n*–4)/9
2,*x*,2	13*i*+6	(*n*–3)	2(*n*–6)/13
4,*x*,4	18*i*–7	(*n*–1)/2	2(*n*–2)/9
6,*x*,6	39*i*–11	(*n*–1)/3	2(*n*–2)/13
6,*x*,6	75*i*–46	(*n*–2)/3	2(*n*–4)/25
8,*x*,8	68*i*–15	(*n*–1)/4	2(*n*–2)/17
1,1,*x*,1,1	3*i*+1	(*n*+1)	2(*n*–1)/3
1,1,*x*,1,1	7*i*+3	(*n*+2)	2(*n*–3)/7
2,*x*,6,*x*,2	9*i*–2	(*n*–2)	(2*n*–5)/9
x,1,1,*x*,1,1,*x*	6(2*i*–1)	8	(*n*–2)/4
x,1,2,*x*,2,1,*x*	12(3*i*–2)	9	2(*n*–3)/9
4,*x*,1,10,1,*x*,4	18*i*+5	(*n*–1)/2	2(*n*–5)/9
6,*x*,1,28,1,*x*,6	75*i*+14	(*n*–2)/3	2(*n*–14)/25
4,*x*,1,1,2,1,1,*x*,4	18*i*–1	(*n*–1)/2	(2*n*–7)/9

Palindromes with Two Variable Terms

palindrome	*n*	*m*	*x*	*y*
x,1,1,*y*,1,1,*x*	6(2*i*–1)	8	(*n*+6)/4–2	(*n*–2)/2
x,1,1,*y*,1,1,*x*	12(*i*–1)	16	(*n*+12)/8–2	(*n*+12)/2–7

Many of these results suggest the possibility of other, more general patterns. Before exploring this topic, there are a few aspects of the implementation that deserve mention.

The Implementation

Much of the implementation follows the lines of other interactive applications that have been described in the Analyst, most notably the one for experimenting with color weavability [2].

Square-root palindromes are represented by records:

```
record palindrome(n, m, seq)
```

where n and m correspond to values in $\sqrt{n^2 + m}$ and seq is the palindromic sequence.

The current list of palindromes is a list, named seqlist, of such records.

Since the specification dialog involves Icon expressions, the computation of palindromes is done by calling a program that incorporates the expressions. This is accomplished by writing definitions for the expressions to an include file in /tmp. Here is the include file for Figure 3:

```
$define NEXP (1 to 100)
$define MEXP (1 to 2 * n)
$define CEXP (1)
$define FEXP (eq([1,1,1]))
```

The program for computing palindromes includes the definition file and uses the definitions as shown below.

```
$include "/tmp/comqir.inc"
      …

  every n := NEXP do {
    every m := MEXP do {
      if not (1 <= m <= 2 * n) then next
      if not CEXP then next
               # … compute palindrome
      if (FEXP) then
               # ... put palindrome on list

      }
    }
```

The output of the program is a list of palindromic sequences, which are records of the type described above. The output is communicated to the initiating program by depositing an xencoded file in /tmp [4]. The use of xencode() and xdecode() allows the complicated data structures to be transferred intact.

The most interesting, as well as the most useful, part of the application involves the solution of sequences of *n* and *m* to give general formulas that are cast in a useful way. It uses the method of differences but only for one level (which explains the form of the results shown previously). In other words, for it to work, the sequences for *n* and *m* must have constant differences, as in

```
nseq := []
every put(nseq, (!seqlist).n)
n := constant(delta(nseq)) |   # fail
```

where the procedures delta() and constant() are

```
procedure delta(seq)
  local deltaseq, i

  deltaseq := []

  every i := 2 to *seq do
     put(deltaseq, seq[i] – seq[i – 1])

  return deltaseq

end

procedure constant(seq)
  local c

  if *set(seq) = 1 then return seq[1]
  else fail

end
```

The remaining challenge is to format the formulas. For *n*, the procedure is

```
procedure eformat(nd, n1)
  local neqn

  if nd = 0 then neqn := n1
  else if n > 1 then neqn := nd || "*i"
  else neqn := "i"

  if n1 > 0 then neqn ||:= "+" || n1
  else if n1 < 0 then neqn ||:= "–" || –n1

  return neqn

end
```

Here nd is the constant difference for *n* and n1 is the difference between the first value of *n* and nd.

As usual, the program, although still under development, is on the Web site for this issue of the Analyst.

More to Come

In the next article on square-root palindromes, we'll continue our exploration of patterns, concentrating on more general ones that characterize classes of palindromes.

After that, we'll look into the distribution of terms in square-root palindromes and the lengths of square-root palindromes.

References

1. "Continued Fractions for Quadratic Irrationals", Icon Analyst 61, pp. 9-15.

2. "Creating Weavable Color Patterns", Icon Analyst 61, pp. 15-20.

3. "Recurrence Relations", Icon Analyst 59, pp. 18-20.

4. "From the Library", Icon Analyst 39, pp. 15-16.

Sigma Quest

Go back in time and let the free spirit in you enter. Talk to it, play, ask the strangest questions.

— Cecil Balmond [1]

A few years ago we came across a small, enigmatic book with the title *Number 9: The Search for the Sigma Code* [1].

The Library of Congress classification for the book is number theory, but the book is a strange combination of fantasy, numerology, and mathematics — which is beyond the reach of the Library of Congress classification system.

As the title indicates, the book focuses on the number 9 and its “magical properties”. These properties mostly are due to the fact that the arithmetic in the book is done in the usual base 10; for arithmetic in base b, $b-1$ figures significantly. The author, an architect, mentions this only briefly near the end of the book. Mentioning it at the beginning would have spoiled all the fun.

Sigma Codes

One of the themes in the book has to do with what happens when you sum the digits of a number, and particularly, when you repeat the process until you get a single digit. For example, 987 yields 9+8+7=24 and 2+4=6, so the “sigma code” (for the Greek symbol used to indicate summation) for 987 is 6. We’ll write this as Σ987=6.

The mathematical term for the result of repeated digit summation is the *additive digital root* [2,3]. We’ll stay with “sigma code” in keeping with the nature of the book that inspired this article (Are you there, Harry Potter?).

The straightforward way of computing the sigma code for the integer n is

```
procedure digsum(n)
  local j

  repeat {
    j := 0
    every j +:= !n
    if *j > 1 then n := j else return j
    }

end
```

It is, however, not necessary to sum all the digits repeatedly to get the sigma code. It suffices to use modular arithmetic:

```
procedure sigma(n)

  if n = 0 then return 0

  n %:= 9

  return if n = 0 then 9 else n

end
```

Ah, the number 9; but in base b it would be $b-1$.

Another aspect of repeated digit summation is how many iterations are needed to get to one digit. This is called the *additive digital persistence* of a number. For 987, the additive digital persistence is 2, as illustrated above.

Again, this is easy to program, although as far as we know, it’s necessary to perform all the digit summations:

```
procedure adp(n)
  local j, k

  k := 0

  until *n = 1 do {
    j := 0
    every j +:= !n
    n := j
    k +:= 1
    }

  return k

end
```

These concepts also apply to multiplication. The multiplicative digital root of 987 is $9 \times 8 \times 7 = 504$, $5 \times 0 \times 4 = 0$. The multiplicative digital persistence of 987 is 2.

Here are procedures to compute the multiplicative digital root and multiplicative digital persistence:

```
procedure mdr(n)
  local i

  until *n = 1 do {
    i := 1
    every i *:= !n
    n := i
    }

  return n

end

procedure mdp(n)
  local i, j

  i := 0

  until *n = 1 do {
    j := 1
    every j *:= !n
    n := j
    i +:= 1
    }

  return i

end
```

There are many number-theoretic matters related to digital roots. See References 2 and 3 and the material they cross reference.

Sigma Code Sequences

We thought it would be interesting to compute the sigma codes for the terms in various sequences. Consider versum sequences produced by repeated digital reversal and addition [4]. The versum sequence starting at 1 is

```
1
2
4
8
16
77
154
605
1111
2222
 …
```

The corresponding sigma sequence is periodic: $\overline{1,2,4,8,7,5}$. This is neither mysterious nor a property of versum sequences. Repeated addition without digit reversal has the same sigma sequence. The reason is clear: A number and its digit reversal — or any digit permutation — have the same sigma code. In addition, the sigma code of the sum of two integers is the sigma code of their individual sigma codes:

$$\Sigma(i+j) = \Sigma(\Sigma(i) + \Sigma(j))$$

Starting with 1, it goes like this:

addition	*sigma*
1	1
1+1=2	2
2+2=4	4
4+4=8	8
8+8=16	7
16+16=32	5
32+32=64	1
…	…

The repeat begins as shown.

There are just three distinct sigma sequences for repeated addition, depending on the starting number:

$$\overline{1,2,4,8,7,5}$$

$$\overline{3,6}$$

$$\overline{9}$$

Figure 1 on the next page shows grid plots for some sigma sequences. See Reference 5 and the recent article on fractal sequences [6] for a description of the generators used.

References

1. *Number 9: The Search for the Sigma Code*, Cecil Balmond, Prestel, 1998.

2. *CRC Concise Encyclopedia of Mathematics*, Eric W. Weisstein, Chapman & Hall/CRC, 1999, pp. 433-434.

3. http://mathworld.wolfram.com/DigitalRoot.html

4. "The Versum Problem", Icon Analyst 30, pp. 1-4.

5. "From the Library — Generators", Icon Analyst 55, pp. 6-7.

6. "Fractal Sequences", Icon Analyst 60, pp. 2-4.

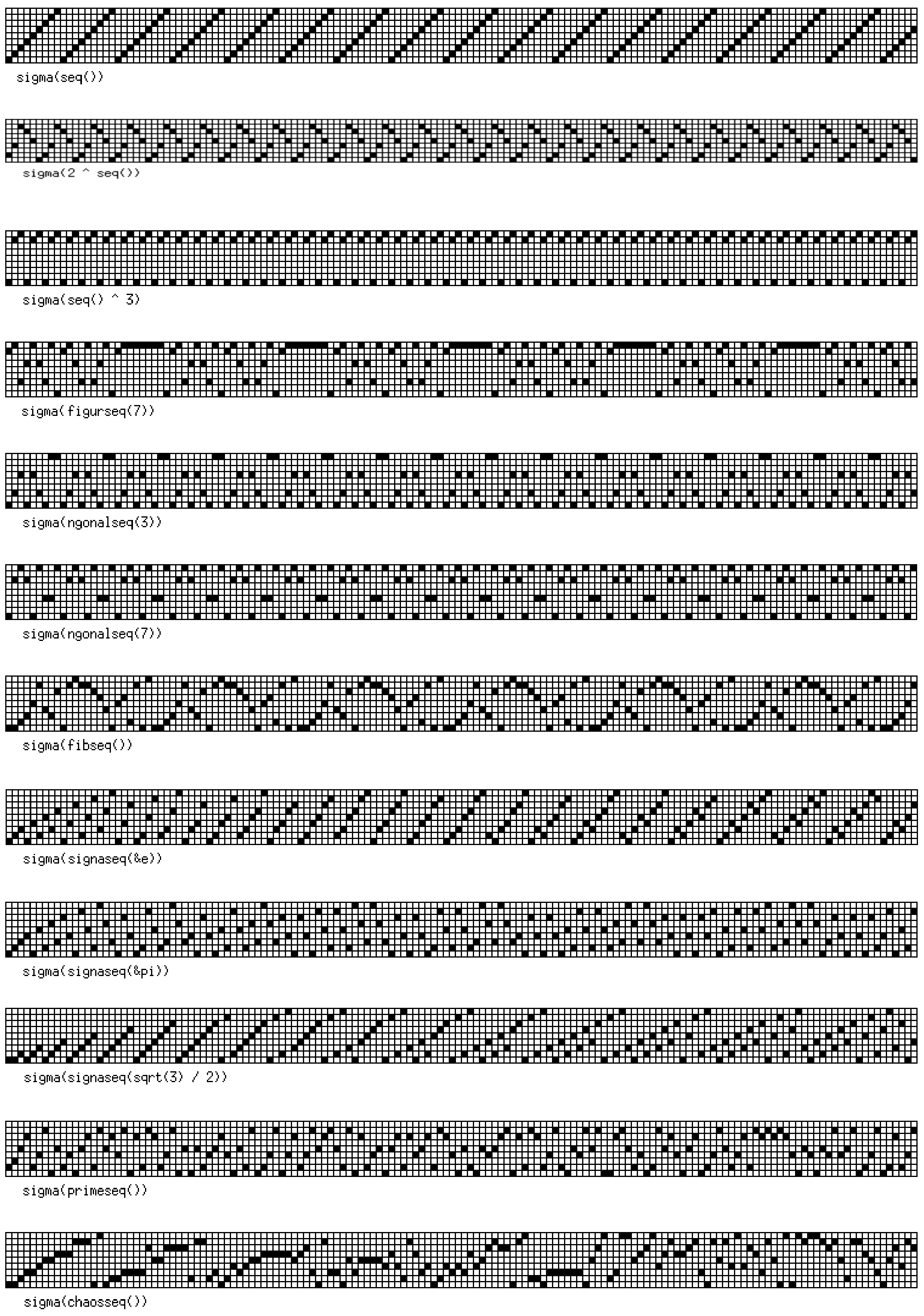


Figure 1. Sigma Sequences

Decimal Fractions

This is the last article on the primary sources of periodic sequences.

We are all familiar with representing fractions by their decimal expansions, as in 1/4 = 0.25, 1/3 = 0.333…, and 1/6 = 0.1666… .

These examples illustrate the three kinds of results that occur in decimal fractions:

1. A finite sequence of digits, as in 1/4 = 0.25.

2. A purely periodic sequence, as in 1/3 = 0.333… .

3. A periodic sequence with a preperiodic part, as in 1/6 = 0.1666… .

The nature of the sequence depends only on the denominator. For example, 3/4 = 0.75 is finite, 2/3 = 0.666… is purely periodic, and 5/6 = 0.8333… has a preperiodic part. Incidentally, any periodic sequence of decimal digits represents a rational number.

In what follows, we'll concentrate on the reciprocals of the natural numbers: 1/2, 1/3, 1/4, … and mention the effects of different numerators later. We'll also discard the whole part and focus on the mantissa [1].

Reciprocals of the natural numbers are called Egyptian fractions because of their use in Egyptian computation. See the side-bar.

To understand decimal fractions, consider long division. For 1/4:

```
   0.25
4 |1.00
    8
    20
    20
     0
```

Since the last remainder is 0, the process terminates and the sequence is finite.

On the other hand, for 1/7, long division goes like this:

Downloading Icon Material

Implementations of Icon are available for downloading via FTP:

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

Egyptian Fractions

The Egyptians used the reciprocals of natural numbers, also called unit fractions, in computation. For example, they represented a fractional quantity as the sum of progressively smaller unit fractions, as in

3/8 = 1/4 + 1/8

and

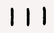
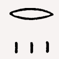
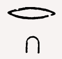
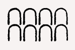
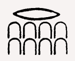
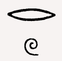
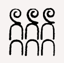
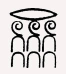
2/5 = 1/3 + 1/15

Some fractions can be represented in more than one way in this fashion, as in

3/7 = 1/3 + 1/11 + 1/231 = 1/4 + 1/7 + 1/28

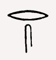
This system of computation was used for centuries by subsequent civilizations.

It remains a mystery why the Egyptians did not use fractions with other numerators. The most likely explanation is that they had a linguistic notation for unit fractions in which the glyph over the glyph for a number represented the reciprocal of the number [1]. Here are some examples:

number	*glyph*	*reciprocal*
3		
10		
80		
100		
360		

This ready notation for converting any number to its corresponding unit fraction probably was a linguistic trap. There was no way to create a glyph for, say, 3/7, or perhaps even to conceive of it.

It's worth noting a single exception: The glyph



represented 2/3 and was used freely among

continued on next page

unit fractions, as in

$7/9 = 2/3 + 1/6$

Interest in Egyptian fractions persists to the present time in the form of number-theoretic problems. For example, Erdös conjectured that $4/n$ and $5/n$ for all $n > 0$ can be expressed as the sum of three Egyptian fractions.

Reference

1. *An Egyptian Hieroglyphic Dictionary in Two Volumes*, E. A. Wallace Budge, Dover, 1978.

```
    0.142857
 7 ⎡1.000000
    7
    30
    28
      20
      14
        60
        56
          40
          35
            50
            49
              1 ...
```

When we get to the remainder 1, we're back where we started, and the sequence is infinite and periodic: $\overline{1,4,2,8,5,7}$.

Since the remainders on division by n are less than n, there are only n possible remainders: 0, 1, … $n - 1$. If a remainder is 0, the division terminates at that point. If a remainder repeats, it can have at most $n - 1$ terms, as in 1/7, but it may have less, as in 1/3, which has a period length of 1.

Here is a procedure for producing the mantissa sequence for an Egyptian fraction:

```
record perseq(pre, rep)

procedure egypt_mantissa(n)
  local  quotients, numer, quotient, seq, count

  quotients := table()

  seq := list(*n – 1, 0)     # possible leading zeros
  numer := 1 || repl("0", *n)

  count := 1

  while numer > 0 do {
    quotient := numer / n
    quotients[quotient] :=  count
    put(seq, quotient)
    numer –:= quotient * n
    numer *:= 10
    count +:= 1
    if count > 3 * n then return repeater(seq)
    }

  return perseq(seq, [])

end
```

The value returned is a record whose first element is a list with the preperiodic part and whose second element is a list with the periodic part [2]. Either list can be empty.

The limit on the number of quotients computed is designed to assure that there are enough terms in the quotient sequence to be able to detect a repeat. This limit is based on knowledge about the lengths of mantissa sequences that is not given here.

We have been looking at quotients, but the remainders also are interesting. For example, the sequence of remainders for 1/3 is $\bar{1}$ and the sequence of remainders for 1/7 is $\overline{3,2,6,4,5,1}$. The remainder sequence of course has the same periodic structure as the quotient sequence. It's easy to modify the procedure given above to produce the remainder sequence.

The nature of the mantissa for the Egyptian fraction $1/n$ depends on the divisors of n. If $n = 2^i \times 5^j$, then n divides 10^k where $k = \max(i, j)$ and the (finite) mantissa sequence has length k. For example, $4 = 2^2 \times 5^0$, max(2,0) = 2, 4 divides 10^2, and the length of the mantissa sequence for 1/4 is 2.

If n does not have a factor of 2 or 5, the mantissa sequence is purely periodic, while if it does but has another prime factor, the mantissa sequence has a preperiodic part.

Note that this information could be used to improve the procedure given earlier.

Period Lengths

One of the most interesting aspects of decimal fractions is the length of their periods. As we mentioned above, the maximum length of the period for the mantissa sequence for $1/n$ is $n - 1$, but this is achieved only when n is a prime. Even then, not all prime n have mantissa sequences of the maximum possible length. Here are the period lengths for the first few primes other than 2 and 5:

prime	*length*
3	1
7	6
11	2
13	6
17	16
19	18

For p prime, the period length depends on the divisors of p –1. While there are methods for determining the length of the mantissa sequence for 1/p, there is no formula.

Divisors

The divisors of a number play an important role in many number-theoretic problems. A naive approach for generating the (proper) divisors of a number is:

```
procedure divisors(n)
  local  j

  every j := 2 to n – 1 do
    if n % j = 0 then suspend j

end
```

We can improve this in two ways: (1) noting that the quotient of a number that divides n also divides n and (2) by limiting the range to $\sqrt{n}$, since $n / \sqrt{n} = \sqrt{n}$:

```
procedure divisors(n)
  local j

  every j := 2 to sqrt(n) do
    if n % j = 0 then {
      suspend j
      suspend n / j
      }

end
```

This produces the divisors but no longer in increasing order. To produce the divisors in increasing order, they can be saved in a list that is sorted at the end:

```
procedure divisors(i)
  local divs, j

  divs := set()

  every j := 2 to sqrt(i) do
    if i % j = 0 then {
      insert(divs, j)
      insert(divs, i / j)
      }

  suspend !sort(divs)

end
```

Of course, instead of generating the values, the list could be returned:

```
return sort(divs)
```

Deciding between generating a (finite) sequence of values and a data structure containing all the values is a frequent design problem.

Note that the penalty for the approaches that use lists is that all the divisors must be computed before any is produced.

The Role of Numerators

Numerators affect the values in the mantissa sequences for fractions, but, as mentioned above, they do not affect the nature of the sequence.

The most interesting situation occurs for *cyclic numbers*, m, for which multiplication by $i = 1, \ldots$ length(m) – 1 simply rotates the digits of m [3]. The mantissa of 1/7, considered as the integer 142857, has this property:

i	*product*
1	142857
2	285714
3	428571
4	571428
5	714285
6	857142

If a cyclic number is multiplied by its length, the result is all 9s, as in $7 \times 142857 = 999999$. As an infinite decimal fraction with repeats, this represents 1, as it should.

Now the prize conversation piece: A number is cyclic if and only if it is the mantissa of a decimal fraction with period $p – 1$ for a prime denominator p. Here are the first few cyclic numbers:

p	*cyclic number*
7	142857
17	0588235294117647
19	052631578947368421
23	0434782608695652173913
29	0344827586206896551724137931
47	0212765957446808510638297872340425531914893617

59 0169491525423728813559322033898305084745762711864406779661

It is generally believed that there are infinitely many cyclic numbers, but this has not been proved.

Figure 1 shows grid plots for these cyclic numbers.

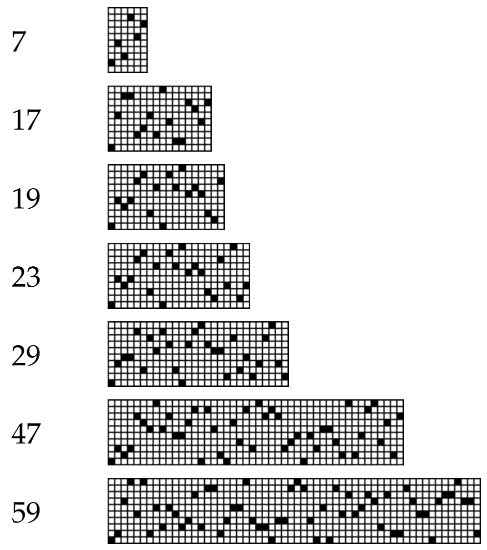


Figure 1. Grid Plots for Cyclic Numbers

Do you see any similarities in these plots?

Other Bases

There is nothing special about the base 10 with respect to fraction expansions. The nature of such fractions, however, depends on the divisors of the base.

For base 12, the (proper) divisors are 2, 3, 4, and 6. Therefore if n base 12 has the form $2^i \times 3^j \times 4^k \times 6^m$, its "duodecimal fraction" is finite.

On the other hand, for a prime base, such as 11, all fractions except reciprocals of powers of 11 are purely periodic.

We'll leave the computation of such fractions in arbitrary bases as "an exercise".

Learning More About Decimal Fractions

Decimal fractions are the subject of both recreational and research mathematics. The literature on the subject is extensive. References 4-9 are accessible to readers with a modest background in mathematics. Reference 10 is the "classic" reference but more difficult.

References

1. "Periodic Sequences", Icon Analyst 57, pp. 5-7.

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

3. *Mathematical Circus*, Martin Gardner, Knopf, 1979, pp. 111-122.

4. *Recreations in the Theory of Numbers: The Queen of Mathematics Entertains*, Albert H. Beiler, Dover, 1964, pp. 73-82.

5. *Number Theory and Its History*, Oystein Ore, Dover, 1988, pp. 311-325.

6. *The Enjoyment of Mathematics*, Hans Rademacher and Otto Toeplitz, Dover, 1990, pp. 147-160.

7. *Mathematical Recreations and Essays*, W. W. Rouse Ball and H. S. M. Coxeter, Dover, 1987, pp. 54-57.

8. *What is Mathematics?*, 2nd. ed., Richard Courant and Herbert Robbins, revised by Ian Stewart, Oxford, 1996, pp. 61-63.

9. *CRC Concise Encyclopedia of Mathematics*, Eric W. Weisstein, Chapman & Hall/CRC, 1999, pp. 382-383, 405-407.

10. *An Introduction to the Theory of Numbers*, 5th Ed., G. H. Hardy and E. M. Wright, Oxford, 1979, pp. 107-128.

Designing Weavable Color Patterns

Most weavers design using drafts — threading sequences, treadling sequences, tie-ups, and the resulting drawdown. The warp and weft threads may be assigned colors, producing a color pattern — a so-called color drawdown.

This process assures a weavable pattern. There is no way to produce one that is not weavable.

Most weavers do not design color patterns for weaving independently of the drafting process. They may have a color pattern in mind, but they work it out within the constraints imposed by drafts.

In the last issue of the Analyst, we described how to ensure weavability in algorithmically constructed patterns and showed transformations on weavable patterns that preserve weavability [1].

We have experimented with a different approach to constructing weavable color patterns; one in which a designer constructs color patterns "from scratch" but not in the context of drafting.

Instead, the designer uses an application that prevents anything that would result in an unweavable pattern.

The Application

The application displays several windows.

The Interface

The application interface, shown in Figure 1, displays three colors associated with the left, middle, and right mouse buttons, respectively.

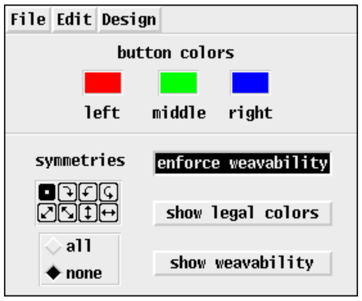


Figure 1. The Interface Window

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 2.

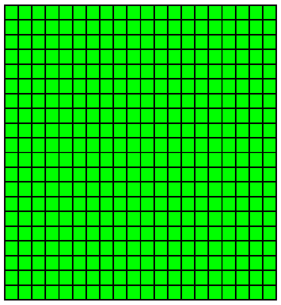


Figure 2. The Design Window

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 weavable. If the result would not be weavable, 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 weavability is not made until the mouse button is released. If the result would not be weavable, the application backtracks, removing the most recently colored cells until there is a weavable 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 is `c1`, but the palette can be changed as described later. See Figure 3.

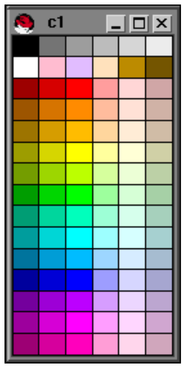


Figure 3. The Palette Window

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 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 4 shows a design produced by using symmetries.

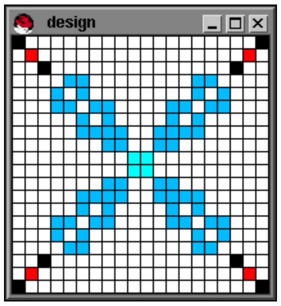
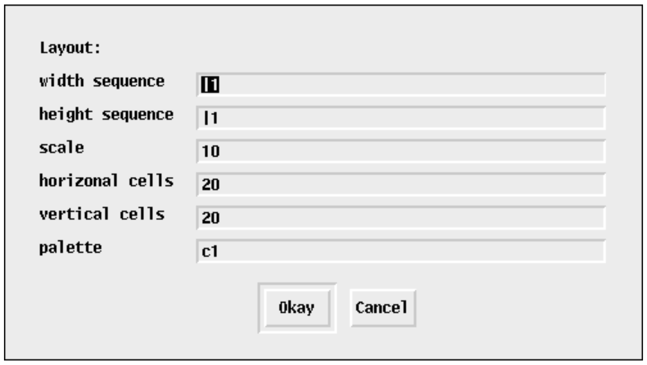


Figure 4. A Symmetrical Design

Layout

The default layout for the design window is a 20 × 20 array of 10-pixel square cells. See Figure 5.



Layout:

width sequence	|1
height sequence	|1
scale	10
horizonal cells	20
vertical cells	20
palette	c1

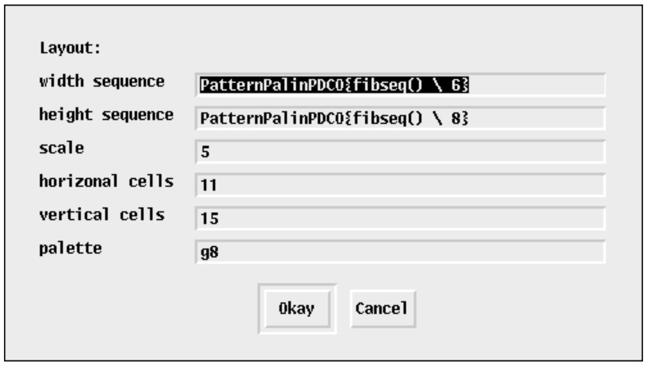
Figure 5. The Default Layout

The cells need not be square. Their widths and heights are determined by the values in sequences produced by Icon expressions. A scaling factor is applied to these values.

The number of cells in the horizontal and vertical directions are specified separately. The width and height sequences can, of course, be limited; the independent specification of the number of cells is a convenience and a safeguard.

Finally, a palette can be specified. 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 6 shows a layout based on the Fibonacci sequence, which is reflected to produce a symmetric result. The resulting design window is shown in Figure 7 and the new palette window is shown in Figure 8.



Layout:

width sequence	PatternPalinPDCO{fibseq() \ 6}
height sequence	PatternPalinPDCO{fibseq() \ 8}
scale	5
horizonal cells	11
vertical cells	15
palette	g8

Figure 6. A Fibonacci Layout

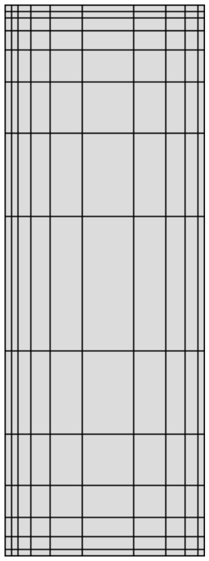


Figure 7. A Fibonacci Design Window

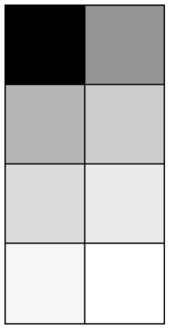


Figure 8. New Palette Window

Enforcing Weavability

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 weavable pattern to another by changing the colors of cells one at a time. For example, it’s always possible to preserve weavability by changing all the cells in a row or column to the same color, but it may not be possible to accomplish this one cell at a time. For example, in the design shown in Figure 4, it is not

Supplementary Material

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

http://www.cs.arizona.edu/icon/analyst/iasub/ia62/

possible to change any single cell to a color not already used.

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 9.

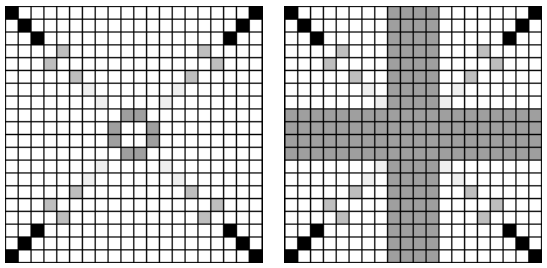


Figure 9. Design and Legal Colors Windows

The legal color window only shows individual cells that can be colored; it does not show all combination 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 [3]. This result shows the row and column colors determined by the test. See Figure 10.

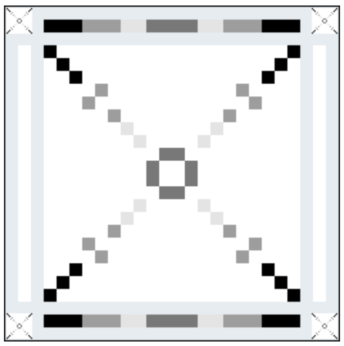


Figure 10. Weavability Solution

Menus and Shortcuts

The File menu, shown in Figure 11, has items for saving an image of the design window, loading a custom palette database, and quitting the application.

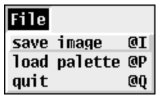


Figure 11. The File Menu

The Edit menu, shown in Figure 12, provides items for undoing and redoing the last change to the design window. 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.

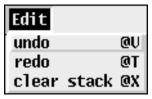


Figure 12. The Edit Menu

The Design menu, shown in Figure 13, provides items related to the design. 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.

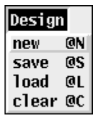


Figure 13. The Design Menu

Since the design window is the focus of attention for most of the activity, the keyboard shortcuts shown in the menus above work for both the interface window and the design window.

The Implementation

Most of the implementation is routine in nature and follows the lines described in previous Analyst articles on interactive applications with visual interfaces.

The event loop is straightforward but complicated by the fact that four windows accept user events: the interface window, the design window, the palette window, and the legal colors window. See Reference 4 for a description of handling events from more than one window.

The layout of the design window is represented by a two-dimensional array of cells. The cells are lists that have the form

```
cell := [win, x, y, width, height]
```

The cell information is used in several ways, including coloring, as in

```
Fg(win, color)
FillRectangle ! cell
```

The colors for the layout are represented by an image string in which each pixel corresponds to a cell. This representation is very compact and contains all that is needed to test the design for weavability, regardless of the sizes of the cells.

A customized version of the program for testing weavability [3] is built into the application to minimize the testing overhead.

The application is in an experimental stage, with new features being added, tried, and saved or discarded depending on the results. Although the present implementation is far from polished, it is usable and the program can be found on the Web site for this issue of the Analyst.

Experience with the Application

Our experience with the application is limited and the only users so far have been familiar with the concepts of weaving and color weavability.

If you don't think in terms of row and color assignments but consider the application as a game, the results can be more frustrating than interesting.

Starting with a "blank" (solid-colored) design window, you can color cells with a second color 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.

You learn things about reserving 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).

Figure 14 shows examples of weavable color patterns created using the application described here. These patterns are best viewed in color; see the Web page for this issue of the Analyst.

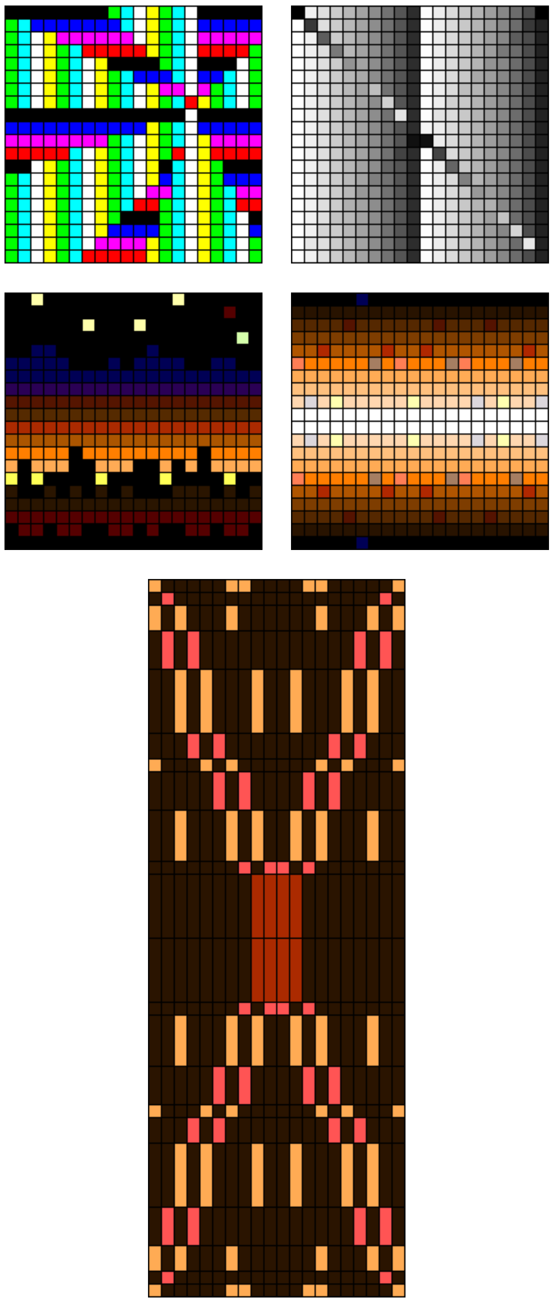


Figure 14. Examples of Weavable Patterns

References

1. "Creating Weavable Color Patterns", Icon Analyst 61, pp. 15-20.

2. *Penelope — A Pattern Tile Editor*, Ralph E. Griswold, IPD234, Dpartment of Computer Science, The University of Arizona, 1996.

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

4. "Multiple VIB Interfaces", Icon Analyst 42, pp. 1-4.

Polygraphic Substitution

The two kinds of substitution ciphers we've considered so far — monoalphabetic substitution [1] and polyalphabetic substitution [2] — replace single plain text characters by single characters. These are forms of monographic substitution.

The devices of polyalphabetic substitution make ciphers more secure, but they still are vulnerable to cryptanalytic techniques [3].

Substitution ciphers that are based on more than one character offer better security. There are many forms of such polygraphic substitution. Digraph substitution, in which pairs of characters are replaced by other pairs of characters, illustrates the concepts.

The simplest digraph ciphers encode pairs of characters using an array in which the individual characters label rows and columns and the replacement is given at the intersection. The replacements must, of course, all be different. And, for good security, the replacements should not have a simple pattern.

The earliest recorded digraph cipher, the Porta cipher, replaced letter pairs by symbols [4]. See the side-bar.

More practical digraph ciphers replace pairs of characters by other pairs of characters. Figure 1 on the next page shows an example.

Note that all possible pairs of plain text characters must be represented. In classical cryptography, where ciphering and deciphering was done by hand, characters like blanks and punctuation were simply deleted from the plain text on the questionable assumption the plain text could be reconstructed from deciphered text unambiguously.

Another problem with polygraphic substitution is that the length of the plain text must be an even multiple of the polygraph size. The usual way to handle this problem is to pad the end of the plain text with innocuous characters that are not likely to confuse the meaning of the message.

Digraph ciphers are easily implemented using tables. The replacement digraphs can be constructed by interleaving (collating) two cipher alphabets that are permutations of the plain alphabet [2].

Suppose cipher_alpha1 and cipher_alpha2 are two such cipher alphabets, and plain_alpha is the

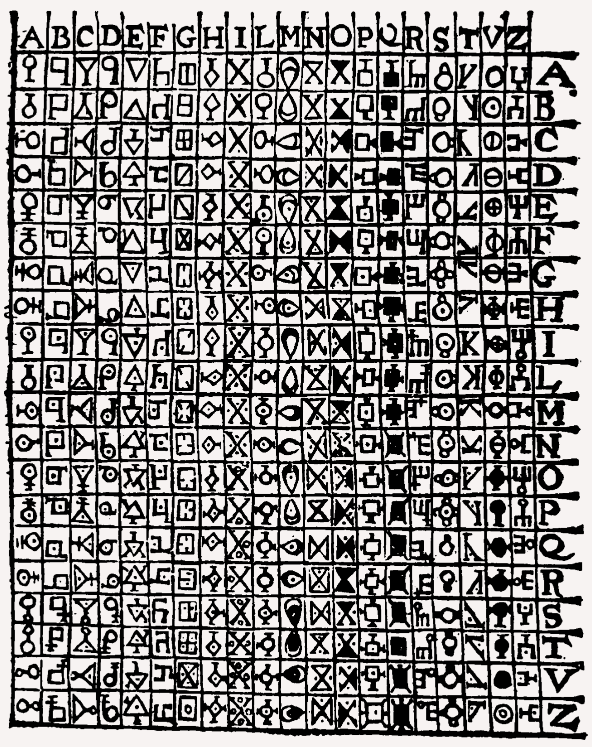
The Porta Cipher

Giovanni Battista Porta
1535-1615

Giovanni Battista Porta was an Italian magician, scientist, alchemist, inventor, and prolific writer.

When he was 28, he published *De Fustivis Literatum Notis*. Its four volumes covered the range of cryptography from ancient times through the knowledge of his time.

Much of the work was original with him, including the first digraphic substitution cipher, which is shown below.


A B C D E F G H I L M N O P Q R S T V Z
A
B
C
D
E
F
G
H
I
L
M
N
O
P
Q
R
S
T
V
Z

In this cipher, every pair of letters is replaced by a symbol. It's hard to imagine using this system, especially deciphering a string of the complicated and strange symbols he used.

Among his other contributions is the cipher disk shown on the next page.

When we saw this, it bought back fond memories of a Captain Midnight Coding Badge that we got for saving the tops of cereal boxes. The Captain Midnight version was, not surprisingly, for Caesar ciphers. Those were the days.

plain alphabet. Here is a procedure to construct the digraph table and encipher a message:

```
procedure encipher(plain_txt, plain_alpha,
   cipher_alpha1, cipher_alpha2)
      local keys, values, digraphs, cipher_txt

   keys := create !plain_alpha || !plain_alpha
   values := create !cipher_alpha1 || !cipher_alpha2

   digraphs := table()

   while digraphs[@keys] := @values

   cipher_txt := ""

   plain_txt ? {
      while cipher_txt ||:= digraphs[move(2)]
      }

   return cipher_txt

end
```

It is assumed here that plain_txt has been padded if necessary so that its length is a multiple of two. What happens if it is not? What happens if the plain

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	zn	zo	zp	zq	zr	zs	zt	zu	zv	zw	zx	zy	zz	za	zb	zc	zd	ze	zf	zg	zh	zi	zj	zk	zl	zm
b	yn	yo	yp	yq	yr	ys	yt	yu	yv	yw	yx	yy	yz	ya	yb	yc	yd	ye	yf	yg	yh	yi	yj	yk	yl	ym
c	xn	xo	xp	xq	xr	xs	xt	xu	xv	xw	xx	xy	xz	xa	xb	xc	xd	xe	xf	xg	xh	xi	xj	xk	xl	xm
d	wn	wo	wp	wq	wr	ws	wt	wu	wv	ww	wx	wy	wz	wa	wb	wc	wd	we	wf	wg	wh	wi	wj	wk	wl	wm
e	vn	vo	vp	vq	vr	vs	vt	vu	vv	vw	vx	vy	vz	va	vb	vc	vd	ve	vf	vg	vh	vi	vj	vk	vl	vm
f	un	uo	up	uq	ur	us	ut	uu	uv	uw	ux	uy	uz	ua	ub	uc	ud	ue	uf	ug	uh	ui	uj	uk	ul	um
g	tn	to	tp	tq	tr	ts	tt	tu	tv	tw	tx	ty	tz	ta	tb	tc	td	te	tf	tg	th	ti	tj	tk	tl	tm
h	sn	so	sp	sq	sr	ss	st	su	sv	sw	sx	sy	sz	sa	sb	sc	sd	se	sf	sg	sh	si	sj	sk	sl	sm
i	rn	ro	rp	rq	rr	rs	rt	ru	rv	rw	rx	ry	rz	ra	rb	rc	rd	re	rf	rg	rh	ri	rj	rk	rl	rm
j	qn	qo	qp	qq	qr	qs	qt	qu	qv	qw	qx	qy	qz	qa	qb	qc	qd	qe	qf	qg	qh	qi	qj	qk	ql	qm
k	pn	po	pp	pq	pr	ps	pt	pu	pv	pw	px	py	pz	pa	pb	pc	pd	pe	pf	pg	ph	pi	pj	pk	pl	pm
l	on	oo	op	oq	or	os	ot	ou	ov	ow	ox	oy	oz	oa	ob	oc	od	oe	of	og	oh	oi	oj	ok	ol	om
m	nn	no	np	nq	nr	ns	nt	nu	nv	nw	nx	ny	nz	na	nb	nc	nd	ne	nf	ng	nh	ni	nj	nk	nl	nm
n	mn	mo	mp	mq	mr	ms	mt	mu	mv	mw	mx	my	mz	ma	mb	mc	md	me	mf	mg	mh	mi	mj	mk	ml	mm
o	ln	lo	lp	lq	lr	ls	lt	lu	lv	lw	lx	ly	lz	la	lb	lc	ld	le	lf	lg	lh	li	lj	lk	ll	lm
p	kn	ko	kp	kq	kr	ks	kt	ku	kv	kw	kx	ky	kz	ka	kb	kc	kd	ke	kf	kg	kh	ki	kj	kk	kl	km
q	jn	jo	jp	jq	jr	js	jt	ju	jv	jw	jx	jy	jz	ja	jb	jc	jd	je	jf	jg	jh	ji	jj	jk	jl	jm
r	in	io	ip	iq	ir	is	it	iu	iv	iw	ix	iy	iz	ia	ib	ic	id	ie	if	ig	ih	ii	ij	ik	il	im
s	hn	ho	hp	hq	hr	hs	ht	hu	hv	hw	hx	hy	hz	ha	hb	hc	hd	he	hf	hg	hh	hi	hj	hk	hl	hm
t	gn	go	gp	gq	gr	gs	gt	gu	gv	gw	gx	gy	gz	ga	gb	gc	gd	ge	gf	gg	gh	gi	gj	gk	gl	gm
u	fn	fo	fp	fq	fr	fs	ft	fu	fv	fw	fx	fy	fz	fa	fb	fc	fd	fe	ff	fg	fh	fi	fj	fk	fl	fm
v	en	eo	ep	eq	er	es	et	eu	ev	ew	ex	ey	ez	ea	eb	ec	ed	ee	ef	eg	eh	ei	ej	ek	el	em
w	dn	do	dp	dq	dr	ds	dt	du	dv	dw	dx	dy	dz	da	db	dc	dd	de	df	dg	dh	di	dj	dk	dl	dm
x	cn	co	cp	cq	cr	cs	ct	cu	cv	cw	cx	cy	cz	ca	cb	cc	cd	ce	cf	cg	ch	ci	cj	ck	cl	cm
y	bn	bo	bp	bq	br	bs	bt	bu	bv	bw	bx	by	bz	ba	bb	bc	bd	be	bf	bg	bh	bi	bj	bk	bl	bm
z	an	ao	ap	aq	ar	as	at	au	av	aw	ax	ay	az	aa	ab	ac	ad	ae	af	ag	ah	ai	aj	ak	al	am

Figure 1. A Digraph Cipher Table

text contains characters that are not in the plain alphabet?

This procedure shows an example of a situation in which co-expressions provide a convenient way to perform a parallel computation concisely without the need for indexes and subscripting.

As usual, the deciphering process is the inverse of the ciphering process, which the keys and values swapped in digraphs:

```
procedure decipher(cipher_txt, plain_alpha,
    cipher_alpha1, cipher_alpha2)
      local keys, values, digraphs, plain_txt

   values := create !plain_alpha || !plain_alpha
   keys := create !cipher_alpha1 || !cipher_alpha2

   digraphs := table()

   while digraphs[@keys] := @values

   plain_txt := ""

   cipher_txt ? {
      while plain_txt ||:= digraphs[move(2)]
      }

   return plain_txt

end
```

It is, of course, possible to use larger polygraphs. The problem is the size of the tables. When done by hand, anything larger than a digraph is impractical and a large alphabet can be overwhelming. Even with just the 26 letters with upper- and lowercase letters considered to be the same, a digraph table has $26^2 = 676$ entries. Differentiating upper- and lowercase letters, adding digits and the blank brings this to 3,969 entries. For trigraph substitution, these numbers swell to 17,575 and 250,047 respectively.

Next Time

In the next article on cryptography, we'll take up the other major classification of ciphers: transposition ciphers in which the characters of the plain next are rearranged.

References

1. "Classical Cryptography", Icon Analyst 59, pp. 7-9.

2. "Polyalphabetic Substitution", Icon Analyst 60, pp. 9-12.

3. *Elementary Cryptanalysis: A Mathematical Approach*, Abraham Sinkov, Random House, 1968, pp. 58-112.

4. *The Codebreakers*, David Kahn, revised edition, Scribner, 1996, pp. 137-143.

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

Programming Tips

In programming, there often are several ways to accomplish the same thing. This is particularly true of Icon, which has a rich repertoire of operations and data structures.

Consider square-free numbers — those that do not have a square factor. These numbers have interesting properties, but there is no known formula for them. Instead, a number must be tested to determine if it's square free.

This involves determining the factors of the number. By the fundamental theorem of arithmetic, any positive integer *i* has a unique decomposition into prime factors of the form

$$i = 2^{n_1} \times 3^{n_2} \times 5^{n_3} \times 7^{n_4} \ldots \times p^{n_m}$$

If none of n_1, n_2, … n_m is greater than 1, then *i* is square free.

The factors module in the Icon program library contains a procedure factors() that returns a list of the prime factors of a number. The list is ordered by increasing magnitude of the factors and if the same factor occurs more than once, it is listed multiple times.

We can use factors() in a procedure to test numbers for square factors:

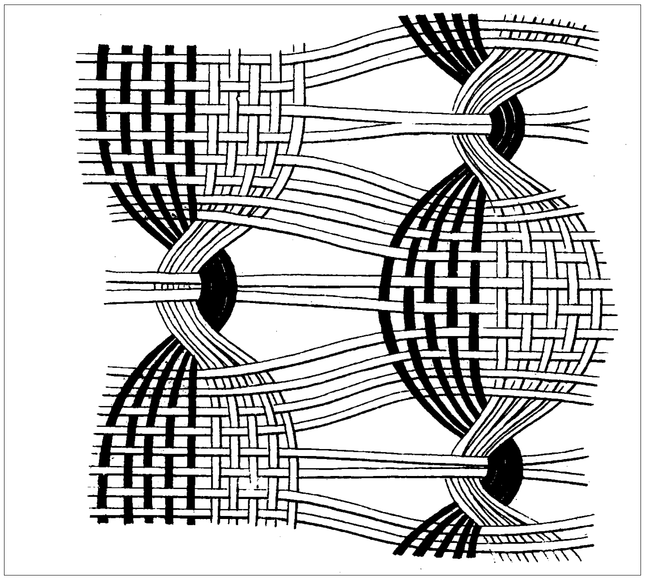
```
link factors

procedure squarefree(i)
   local facts, i

   facts := factors(i)

   every i := 1 to *facts – 1 do
      if facts[i] = facts[i + 1] then fail

   return i

end
```

This is straightforward and typical of the kind of code that might be written in many programming languages. In Icon, there is an easier way:

```
procedure squarefree(i)
   local facts

   facts := factors(i)

   if *facts = *set(facts) then return i
   else fail

end
```

This method also is more general: It works if duplicate terms in a list are not in order. So we can write

```
procedure dupl_elem(L)

   if *L = *set(L) then fail
   else return L

end

procedure squarefree(i)

   if  dupl_elem(factors(i)) then fail
   else return i

end
```



What's Coming Up

On our agenda for the next issue of the Analyst are transposition ciphers and another article in the series on continued fractions for square roots.

In our series on sequences, we plan to have an article on packet sequences, in which the terms of a sequence can be sequences.

We've been exploring the analysis and synthesis of T-sequences. If that work progresses as expected, we'll have the first in a series of articles on that topic.

We also are working on the use of Lindenmeyer Systems for the design of T-sequences.

But mostly we are juggling things for the last four issues of the Analyst, trying to figure out what will get done in time, what will fit, and what won't.