Whizbang!

—by Roy A. Sykes, Jr.
Los Angeles, California

Evolution

SOBERING IS THE EXPERIENCE of examining one's ancient code. We often tend to think we have matured in our code design and programming skills, and looking back at old code reinforces this tendency. But the fact that it keeps happening—yesterday's code looks like trash compared to today's gems—really belies the truth that we are always improving, and our code is never quite so extraordinary as we might wish to think.

I recently was confronted with this disconcerting thought while reading Eugene McDonnell's column "At Play with J" in the October, 1996 (Vol.13, No.2) issue of *Vector*, the excellent sister publication of *APL Quote Quad* from the British APL Association. This particular column was entitled "Volutes," which are spirals of numbers as illustrated below (don't worry about the arguments yet, although the right should be plain):

```
      ⎕IO←1 ⋄ L←0 8 ⋄ R←4 5
      'Involutes' 'Evolutes',V←L∘.VOLUTE R
 Involutes     1  2  3 4     1  2  3  4 5
              12 13 14 5    16 17 18 19 6
              11 16 15 6    15 24 25 20 7
              10  9  8 7    14 23 22 21 8
                            13 12 11 10 9

 Evolutes     16 15 14 13   25 24 23 22 21
               5  4  3 12   10  9  8  7 20
               6  1  2 11   11  2  1  6 19
               7  8  9 10   12  3  4  5 18
                            13 14 15 16 17
```

Involutes spiral in (increase) from a corner, and evolutes spiral out from the center. One may also view evolutes as spiraling in (decreasing) from a corner, and in fact subtracting either from `(R*2)-¯1*⎕IO` (here `17` or `26`) produces the other.

```
      V≡(2 2⍴17 26)-(⌽L)∘.VOLUTE R
1
```

Notice that only odd volutes have a distinct center item. Because of this characteristic, let us describe involutes by where they start (`⎕IO`) and evolutes by where they end (`¯1+⎕IO+R*2`). Thus we will call all of these top-left clockwise volutes.

Recalling that I had solved this problem some time ago (no doubt tastefully) I rummaged about and found (to my chagrin) the following, which I've edited only slightly for publication:

```
    ∇ R←A SPIRAL B;C;D;⎕IO
[1]   ⍝ NONNEGATIVE INTEGER SCALAR <B> GIVES THE LENGTH
[2]   ⍝ OF THE SPIRAL.  CHARACTER VECTOR <A> SPECIFIES
[3]   ⍝ A SEQUENCE OF DIRECTIONS (1=↑=NORTH, 2=→=EAST,
[4]   ⍝ 3=↓=SOUTH, 4=←=WEST).  5 JUN 82 /ROY
[5]    R←(B⌊1 1)⍴⎕IO←0 ⋄ →⍳B≤1 ⍝ ESCAPE IF TRIVIAL
[6]   ⍝ DECODE DIRECTIONS:
[7]    D←(12 2⍴¯1 0 0 1 1 0 0 ¯1)['NESW↑→↓←1234'⍳A←,A;]
[8]   ⍝ COMPUTE THE MOVES:
[9]    C←⌈(4×R←B-1)*0.5 ⋄ D←0⍪+⍀D[R↑(|-\1+⍳C)/C⍴⍳⍴A;]
[10]  ⍝ FILL IN THE NUMBERS:
[11]   R←(×/C←1+⌈⌿D←D-(⍴D)⍴⌊⌿D)⍴0 ⋄ R[C⊥⍉D]←⍳B ⋄ R←C⍴R
    ∇

      'WSEN' SPIRAL 25  ⍝ top-left clockwise evolute
 24 23 22 21 20
  9  8  7  6 19
 10  1  0  5 18
 11  2  3  4 17
 12 13 14 15 16
```

Well, at least the comments were spelled correctly. Actually, the code wasn't too bad, although my design for the arguments was clearly misguided. The left argument does not provide for involutes, and has arbitrary synonyms (which did not include lowercase because this was written on a mainframe) for directions (whatever happened to left, down, right, and up?). Instead of specifying the side length, the right argument is its square (why I don't know), and the documentation is not explicit about this. Also, the function only provides for origin-0 results (Eugene and other J advocates would no doubt approve), again with the comments silent. Furthermore, one can call `SPIRAL` with arguments for it is clearly neither prepared nor demanding enough to reject, generating spurious results:

```
      'W→2' SPIRAL 23
 1 6 7 16 17 18 19 20 21 22
```

I now prefer that computational functions such as this have more succinct arguments, relying on application cover functions to decode whatever oddball method of specification the user may choose. Also, an argument should be as restrictive as the code for which it is intended. We'll look at `VOLUTE`'s arguments later, but first let's talk about the algorithm.

Gene's article described six different J solutions, culminating with a clever algorithm originally written in APL by Joey Tuttle, which we present for J aficionados (spaces have been introduced for clarity):

```
   evJKT =. ,~ $ /: @ (+\/) @ evJKT2
   evJKT2=. _1&|. @ (evJKT0 # evJKT1)
   evJKT1=. <:@+: $ _1: , ] , 1: , -
   evJKT0=. }:@(2: # >:@i.)
```

Here is the code recast in terms of APL (it assumes `⎕IO←0`):

```
    ∇ Z←evJKT R
[1]   Z←(2⍴R)⍴⍋+\evJKT2 R
    ∇

    ∇ Z←evJKT2 R
[1]   Z←¯1⌽(evJKT0 R)/evJKT1 R
    ∇
```

```
    ∇ Z←evJKT1 R
[1]   Z←(¯1+2×R)ρ¯1,R,1,-R
    ∇

    ∇ Z←evJKT0 R
[1]   Z←¯1↓2/1+⍳R
    ∇

      ⎕IO←0 ⋄ evJKT 5
24 23 22 21 20
 9  8  7  6 19
10  1  0  5 18
11  2  3  4 17
12 13 14 15 16
```

Our taste in programming is to avoid a proliferation of trivial little functions, many of which being once-used subroutines. Rather, we prefer to have somewhat heftier programs, so let's bulk up our translation of Gene's code into a one-liner, after localizing *⎕IO* to reproduce J's fixed origin-0 behavior:

```
    ∇ Z←evJKTio0 R;⎕IO
[1]   ⎕IO←0
[2]   Z←(2ρR)ρ⍋+\¯1⌽(¯1↓2/1+⍳R)/(¯1+2×R)ρ¯1,R,1,-R
    ∇
```

That's not too bad, is it? Better yet, we can incorporate *⎕IO* into the code to make it origin-sensitive; notice that only the right argument of replicate (2/) changes:

```
    ∇ Z←EVJKT R
[1]   Z←(2ρR)ρ⍋+\¯1⌽(¯1↓2/(~⎕IO)+⍳R)/(¯1+2×R)ρ¯1,R,1,-R
    ∇
      ⎕IO←1 ⋄ E1←EVJKT 5 ⋄ ⎕IO←0 ⋄ E0←EVJKT 5 ⋄ E0 E1
 24 23 22 21 20   25 24 23 22 21
  9  8  7  6 19   10  9  8  7 20
 10  1  0  5 18   11  2  1  6 19
 11  2  3  4 17   12  3  4  5 18
 12 13 14 15 16   13 14 15 16 17
```

It turns out that my *SPIRAL*, sans its arcane arguments, was similar to the first approach Gene described in his article, which he found in the book *Concrete Mathematics* by Graham, Knuth, and Patashnik as Exercise 3.40. Although the book took a scalar approach whereas *SPIRAL* naturally uses arrays, both methods are basically geometrical. Two aspects piqued my curiosity:

- *SPIRAL* has a lot of argument decoding nonsense which masks its essence. Stripped to its essentials, how close would it be to *EVJKT*? Could one be derived from the other?
- *EVJKT* has a grade up (⍋) but *SPIRAL* does not. I thought the big-O nature of the problem to be N-squared (the size of the result) rather than N-squared times log N-squared (the nature of sorting). Could a stripped down *SPIRAL* improve upon *EVJKT* by avoiding a costly grade up?

I also wondered if I could succinctly describe and efficiently code one function which would produce all square involutes and involutes.

Starting with *SPIRAL*, I eliminated the left argument and all its ghastly code, essentially hardwiring the *'WSEN'* value which produces the same result as Gene's verbs. I also changed the right argument to be the side length, the choice everyone else had apparently settled on well before me, and I changed the local names to those I now prefer.

```
    ∇ Z←EVO R;A;B;C;D;⎕IO
[1]   Z←(R⌊1 1)ρ⎕IO←0 ⋄ →⍳R≤1
[2]   B←4 2ρ0 ¯1 1 0 0 1 ¯1 0
[3]   C←0⍪+⍀B[(¯1+R×R)↑(|-\1+⍳2×R)/(2×R)ρ⍳4;]
[4]   Z←(×/A←1+⌈⌿D←C-(ρC)ρ⌊⌿C)ρ0
[5]   Z[A⊥⍉D]←⍳R×R ⋄ Z←AρZ
    ∇

      EVO R←4
 9  8  7  6
10  1  0  5
11  2  3  4
12 13 14 15
```

After these essentially bookkeeping chores, I delved into the algorithm. The rare subtraction scan (-\) on line [3] was probably an old idiom held over from the days before replication (compression extended to positive integers above 1), which appeared only in the early 1980's in most versions of APL. Incongruously, this vector formed the left argument to … replication.

```
      1+⍳2×R            ⍝ remember we're using origin 0
1 2 3 4 5 6 7 8
      -\1+⍳2×R
1 ¯1 2 ¯2 3 ¯3 4 ¯4
      |-\1+⍳2×R         ⍝ old habits
1 1 2 2 3 3 4 4
      2/1+⍳R            ⍝ new tricks
1 1 2 2 3 3 4 4

      (2×R)ρ⍳4
0 1 2 3 0 1 2 3
      (2/1+⍳R)/(2×R)ρ⍳4
0 1 2 2 3 3 0 0 0 1 1 1 2 2 2 2 3 3 3 3
```

The truncation of the resulting row indices of small matrix *B* could also be simplified from (¯1+R×R)↑… to (¯1-R)↓…:

```
      ⎕←B←4 2ρ0 ¯1 1 0 0 1 ¯1 0
 0 ¯1
 1  0
 0  1
¯1  0
      ρC←0⍪+⍀B[(¯1+R×R)↑(|-\1+⍳2×R)/(2×R)ρ⍳4;]   ⍝ old
16 2
      C≡0⍪+⍀B[(¯1-R)↓(2/1+⍳R)/(2×R)ρ⍳4;]         ⍝ new
1
      ⍉C
0  0  1 1 1 0 ¯1 ¯1 ¯1 ¯1  0  1  2  2 2 2
0 ¯1 ¯1 0 1 1  1  0 ¯1 ¯2 ¯2 ¯2 ¯2 ¯1 0 1
```

Next I examined how the two columns of *C* were adjusted and combined on the last two lines. Line [4] simply normalized them to 0 as D, computed the radix A for evaluating D as a first order polynomial (ax+b) to calculate indices on line [5], and used its product to create Z. But we already know (by definition) that ρZ will be 2ρR and thus ρ,Z is ×/2ρR; therefore A must be 2ρR.

```
      ⍉D←C−(ρC)ρ⌊⌿C
1 1 2 2 2 1 0 0 0 0 1 2 3 3 3 3
2 1 1 2 3 3 3 2 1 0 0 0 0 1 2 3
      (⎕←2ρR)≡A←1+⌈⌿D
4 4
1
```

The first item of A in A⊥⍉D is ignored by base value (⊥) except for length conformability checking, so we can use R instead:

```
      (⎕←R⊥⍉D)≡A⊥⍉D
6 5 9 10 11 7 3 2 1 0 4 8 12 13 14 15
1
```

This was getting mighty interesting. R⊥⍉D is a permutation vector (all indices unduplicated). We can test this by knowing that ⍋ inverts a permutation, and so ⍋⍋ should (by inverting the inverse) return the original permutation.

```
      ⍋R⊥⍉D
9 8 7 6 10 1 0 5 11 2 3 4 12 13 14 15
      ⍋⍋R⊥⍉D
6 5 9 10 11 7 3 2 1 0 4 8 12 13 14 15
```

Whoa! Look at the result we want:

```
      EVO R
 9  8  7  6
10  1  0  5
11  2  3  4
12 13 14 15
```

Now look above at ⍋R⊥⍉D. Notice anything? Grade's got it!

```
    ∇ Z←EVOL R;B;C;D;⎕IO
[1]   Z←(R⌊1 1)ρ⎕IO←0 ⋄ →ιR≤1
[2]   B←4 2ρ0 ¯1 1 0 0 1 ¯1 0
[3]   C←0⍪+⍀B[(¯1-R)↓(2/1+ιR)/(2×R)ρι4;]
[4]   D←C−(ρC)ρ⌊⌿C
[5]   Z←(2ρR)ρ⍋R⊥⍉D
    ∇
```

Flushed with success, I also noticed that R⊥⍉D seemed to have a pattern of ups and downs, which upon examination,

```
      ⎕←P←R⊥⍉D
6 5 9 10 11 7 3 2 1 0 4 8 12 13 14 15
      ¯2-/P        ⍝ same as  (1↓P)-¯1↓P
¯1 4 1 1 ¯4 ¯4 ¯1 ¯1 ¯1 4 4 4 1 1 1
      +\6,¯2-/P  ⍝ same as  +\P-¯1↓0,P
6 5 9 10 11 7 3 2 1 0 4 8 12 13 14 15
```

looked like increasing repetitions of cycling ¯1,R,1,-R with a prefix. Experimentation with other even arguments convinced me that the prefixing 6 was actually +/ιR (or better (R×R−1)÷2 or better yet 2!R) [but see below for odd arguments]. The repeat sequence 1 1 2 2 3 3 3 is easy except for that annoying final 3. A venerable method in APL is to generate too much data and then truncate, so we'll use a familiar replication sequence, 1 1 2 2 3 3 4 4, and then eliminate the final five replicated items, this time by taking (↑) the first R×R items rather than dropping (↓) the last ¯1−R items:

```
      'Repeat' 'Value',⊃(2/1+ιR)((2×R)ρ¯1,R,1,-R)
Repeat   1 1 2  2  3 3 4  4
Value   ¯1 4 1 ¯4 ¯1 4 1 ¯4

      (2!R),(2/1+ιR)/(2×R)ρ¯1,R,1,-R
6 ¯1 4 1 1 ¯4 ¯4 ¯1 ¯1 ¯1 4 4 4 1 1 1 1 ¯4 ¯4 ¯4 ¯4

      P≡⎕←+\(R×R)↑(2!R),(2/1+ιR)/(2×R)ρ¯1,R,1,-R
6 5 9 10 11 7 3 2 1 0 4 8 12 13 14 15
1
```

Essentially, rather than manipulating row indices of matrix B and then evaluating the polynomial as we did above, we have precomputed all values of the polynomial,

```
      R⊥⍉B
¯1 4 1 ¯4
```

and then manipulated these items directly. The values actually represent the increments necessary to move left one column (¯1), down one row (4 for a four-column matrix), right one column (1), and up one row (¯4)—the maligned '←↓→↑' or 'WSEN' argument of SPIRAL.

Let's see where we are compared to Joey's algorithm:

```
    ∇ Z←EVOLU R;⎕IO
[1]   Z←(R⌊1 1)ρ⎕IO←0 ⋄ →ιR≤1
[2]   Z←(2ρR)ρ⍋+\(R×R)↑(2!R),(2/1+ιR)/(2×R)ρ¯1,R,1,-R
    ∇

    ∇ Z←EVJKT R;⎕IO
[1]   Z←(2ρR)ρ⍋+\¯1⌽(¯1↓2/(~⎕IO)+ιR)/(¯1+2×R)ρ¯1,R,1,-R
    ∇
```

We're pretty close it seems, except for ⎕IO and that annoying branch. A vestige from SPIRAL, the test seems a bit of false optimization (every caller pays but few callers benefit), but removing it from SPIRAL causes multiple problems when the right argument is 0 (the escape on 1 is a free benefit), the first of which is

```
      ∇SPIRAL[5]⎕IO←0∇

      'WSEN' SPIRAL 0
DOMAIN ERROR
SPIRAL[9] C←⌈(4×R←B−1)*0.5 ⋄ D←0⍪+⍀D[R↑(|-\1+ιC)/CριρA;]
                          ^
      4×R ⋄ →
¯4
```

In fact, *EVJKT* suffers from the same defect:

```
      EVJKT 0
DOMAIN ERROR
EVJKT[1]⋄ Z←(2⍴R)⍴⍋+\¯1⌽(¯1↓2/(~⎕IO)+⍳R)/(¯1+2×R)⍴¯1,R,1,-R
                                                    ∧

      ¯1+2×R ⋄ →
¯1
```

Pleasantly and fortuitously *EVOLUT* executes gracefully. The reason is that *EVOLUT* truncates to the proper length at the end (the venerable method), whereas *EVJKT* tries to calculate the proper lengths intermediately but fails to account for the limit.

```
      ∇EVOLU[1]⎕IO←0∇

      ⍴EVOLU 0
0 0
```

The origin handling can be accommodated in the same way we moved from evJKTio0 to *EVJKT*:

```
    ∇ Z←EVOLUT R
[1]   Z←(2⍴R)⍴⍋+\(R×R)↑(2!R),(2/(~⎕IO)+⍳R)/(2×R)⍴¯1,R,1,-R
    ∇
```

Now we have essentially demonstrated the first contention—that *EVJKT* could be derived from *SPIRAL*. The only differences are slight variations in how the movements are constructed, and the initial item, which Joey neatly brings from the rear. Nonetheless, I have good reason to use 2!*R*!

Remember that the +\ in *EVOL* is a permutation vector, and that ⍋*P* inverts a permutation vector? A much faster method is Z←⍳⍴*P* ⋄ Z[*P*]←Z (or just *P*[*P*]←⍳⍴*P*) because it involves no sorting. So instead of computing Z←(2⍴*R*)⍴⍋+\..., we can set Z←⍳*R*×*R* and then perform Z[+\...]←Z followed by Z←(2⍴*R*)⍴Z. Furthermore, on APL systems which employ so-called "pass-through localization" for system variables (applause for no ⎕*IO IMPLICIT ERROR*'s), we can compute Z←⍳*R*×*R* in the user's global origin, then switch conveniently to origin 1 to compute its permutation, thus simplifying 2/(~⎕*IO*)+⍳*R* to 2/⍳*R*.

Unfortunately, using 2!*R* as the leading item preserves the permutation only for even values of *R*. Notice that the leading item is also the index into which 1↑Z (⎕*IO*) will be assigned—the center of the evolute—which varies in a non-obvious way for odd and even arguments (involutes are easier in this regard):

```
      ⎕IO←1 ⋄ 5⍴E←EVOLUT¨N←⍳10
 1    2 1    9 8 7    10  9  8  7    25 24 23 22 21
      3 4    2 1 6    11  2  1  6    10  9  8  7 20
             3 4 5    12  3  4  5    11  2  1  6 19
                      13 14 15 16    12  3  4  5 18
                                     13 14 15 16 17
```

Experimentation leads us to the correct value:

```
      2!N
0 1 3 6 10 15 21 28 36 45

      (,¨E)⍳¨1
1 2 5 7 13 16 25 29 41 46

      1+(2!N)+(2|N)×⌊N÷2  ⍝ add 1, and ⌊N÷2 if odd
1 2 5 7 13 16 25 29 41 46

      1+(2!N)+×≠0 2⊤N     ⍝ an unusual use of encode
1 2 5 7 13 16 25 29 41 46

    ∇ Z←EVOLUTE R;A;B;⎕IO
[1]   Z←⍳A←R×R ⋄ ⎕IO←1 ⋄ B←1+(2!R)+×≠0 2⊤R
[2]   Z[+\A↑B,(2/⍳R)/(2×R)⍴¯1 0 1 0+0 1 0 ¯1×R]←Z
[3]   Z←(2⍴R)⍴Z
    ∇
```

Note the faster way of calculating ¯1,*R*,1,-*R* on line [2] using only two primitives rather than four. If your APL system does not support pass-through localization, use the following:

```
    ∇ Z←EVOLUTe R;A;B
[1]   Z←⍳A←R×R ⋄ B←⎕IO+(2!R)+×≠0 2⊤R
[2]   Z[+\A↑B,(2/(~⎕IO)+⍳R)/(2×R)⍴¯1 0 1 0+0 1 0 ¯1×R]←Z
[3]   Z←(2⍴R)⍴Z
    ∇
```

Timings confirm that we have indeed evolved to a better solution (all are ratios to *EVOLUTE* on APL+DOS):

R=	5	21	55	89	377	378
'WSEN'*SPIRAL R*×*R*	2.1	3.7	4.5	4.7	5.1	5.1
EVO R	1.9	3.6	4.5	4.7	4.9	4.9
EVOL R ⍝ uses ⍋	1.7	4.0	5.8	6.2	7.3	7.2
EVJKT R ⍝ uses ⍋	1.3	4.8	6.4	6.8	7.3	7.2
EVOLUT R ⍝ uses ⍋	1.3	4.2	6.2	6.5	7.1	4.4
EVOLUTE R	1	1	1	1	1	1

Observe that as *R* increases, the merits of avoiding ⍋ become more apparent. Note particularly the anomalous ratio at *EVOLUT* 378. This is because when *R* is even, the argument to grade in *EVOLUT* is a permutation vector, which is processed faster in APL+DOS. Had we changed (2!*R*) to the more precise (⎕*IO*+(2!*R*)+×≠0 2⊤*R*), then *EVOLUT*'s ratios would have improved for odd as well as even arguments. However, APL systems vary widely in the optimizations they perform, so you might check this particular one on yours.

Gene's article also had a table of timing ratios, and we should point out that all of the solutions we've presented are vastly superior to scalar, iterative, and recursive strategies. His ratios were all relative to verb evJKT, which we translated and consolidated as function *EVJKT*. Extrapolating the 55 and 89 columns above to his other verbs we derive the following ratios

Verb or Function	Method	R=55	R=89
GKPa	scalar	1434	1659
GKPb	scalar	602	721
KS	recursive	1069	WsFull
EEM	iterative	83	109
HUI	array	13	9.5
EVJKT	array	6.4	6.8
SPIRAL	array no ⍋	4.5	4.7
EVOLUTE	array no ⍋	1	1

Could you be persuaded by these magnitudes to avoid certain scalar and recursive solutions?

Finally, we present the grand unification, which uses tables, rather than calculations, to speed processing of the three options (type of volute, starting corner, and rotation). The fully-commented code is at the end of this article. We reintroduced the leading test not to speed the trivial cases, but so that the mainline code need not deal with them. A yet-faster version has distinct code for odd *R* evolutes, even *R* evolutes, and all involutes.

```
    ∇ Z←L VOLUTE R;A;B;C;D;E;⎕IO
  ⍝∇Compute square volute of size R (nonnegative
  ⍝∇integer scalar) and type,start,rotate=2 4 2⊤L.
   Z←⍳A←R×R ⋄ ⎕IO←0 ⋄ →(R<2)/L01
   B←(R×2 0 1 0 ¯1 0 1 ¯1 0)+1 1 0 ¯1 0 0 ¯1,A-0 1
   C←8 4⍴ 1 2 3 4  2 1 4 3  2 3 4 1  3 2 1 4
          4 1 2 3  1 4 3 2  3 4 1 2  4 3 2 1
   E←((D←8|L)⊃5 5 6 6 7 7 8 8)⊃B
   Z[+\E,((6⊃B),2/⌽⍳R)/(↑B)⍴B[C[D;]]]←Z
  L01:Z←(2⍴R)⍴Z ⋄ →⍳8>16|L ⋄ Z←(¯1+A)-Z
    ∇
```

***VOLUTE* can generate all 16 kinds of square volutes. The left argument is (8×*TypeVolute*)+(2×*StartCorner*)+*RotateDirection* —a concise encoding somewhat akin to that used for the circular functions.**

```
      ⎕IO←0
      L←16 1⍴⍳16
      4 8⍴(⊂[1](⍕L),':'),L VOLUTE¨3
  0:  0 1 2    1:  0 7 6    2:  6 7 0    3:  2 1 0
      7 8 3        1 8 5        5 8 1        3 8 7
      6 5 4        2 3 4        4 3 2        4 5 6

  4:  2 3 4    5:  6 5 4    6:  4 5 6    7:  4 3 2
      1 8 5        7 8 3        3 8 7        5 8 1
      0 7 6        0 1 2        2 1 0        6 7 0

  8:  8 7 6    9:  8 1 2   10:  2 1 8   11:  6 7 8
      1 0 5        7 0 3        3 0 7        5 0 1
      2 3 4        6 5 4        4 5 6        4 3 2

 12:  6 5 4   13:  2 3 4   14:  4 3 2   15:  4 5 6
      7 0 3        1 0 5        5 0 1        3 0 7
      8 1 2        8 7 6        6 7 8        2 1 8
```

As is usual in APL, the documentation vastly overwhelms the code. Isn't evolution great? ■

```
    ∇ Z←L VOLUTESRC R;A;B;C;D;E;⎕IO
  ⍝∇Compute square volute of size R (nonnegative
  ⍝∇integer scalar) and type,start,rotate=2 4 2⊤L.
  ⍝ 1Aug97/ROY
  ⍝ 16|L (integer scalar) controls the volute.
  ⍝ Given (t s r)←2 4 2⊤L, then
  ⍝  t is type (L is 0-7 or 8-15):
  ⍝        0=involute (spiraling inward from ⎕IO),
  ⍝        1=evolute (spiraling outward from ⎕IO).
  ⍝  s is starting corner of ⎕IO if involute,
  ⍝               of ¯1+⎕IO+R*2 if evolute:
  ⍝        0=   top left,  1=   top right,
  ⍝        2=bottom left,  3=bottom right.
  ⍝  r is rotation from ⎕IO if involute,
  ⍝          from ¯1+⎕IO+R*2 if evolute:
  ⍝        0=clockwise (L is even),
  ⍝        1=counterclockwise (L is odd).
  ⍝ 8|L (s and r):
  ⍝ 0══╗ 1          2 ╔══3 ╔══>     ^ ^     <══╗
  ⍝    ║ ║          ║ ║    ║        ║ ║        ║
  ⍝    v ╚══> <═════╝ v    4   5════╝ ╚════6   7
  ⍝ ('WSEN'SPIRAL R*2)≡(14-6×2|R)VOLUTE R
  ⍝ (see Vector 13.2, p.144 by EEM).

   Z←⍳A←R×R        ⍝ Z is origin-sensitive, ...
   ⎕IO←0           ⍝ but local origin is 0.
   →(R<2)/L01      ⍝ Done if trivial.

  ⍝ COMPUTE INVOLUTE:
  ⍝   reshape┌─moves──┐┌──────────start─────────┐
  ⍝ B[  0    1  2  3  4  5  6        7        8 ]
  ⍝ B←(1+2×R) 1 R ¯1 (-R) 0 (R-1) (R×R-1)(¯1+R×R)
   B←(R×2 0 1 0 ¯1 0 1 ¯1 0)+1 1 0 ¯1 0 0 ¯1,A-0 1

  ⍝ C is indices of moves in B:
  ⍝ 1=1 moves right 1 col, 3=¯1 moves left 1 col,
  ⍝ 2=R moves down  1 row, 4=-R moves up   1 row.
   C←8 4⍴ 1 2 3 4  2 1 4 3  2 3 4 1  3 2 1 4
          4 1 2 3  1 4 3 2  3 4 1 2  4 3 2 1

  ⍝ E is position of ⎕IO (involute start corner):
  ⍝ 5=0     is top left,  6=R-1    is top right,
  ⍝ 7=R×R-1 bottom left,  8=¯1+R×R bottom right.
   E←((D←8|L)⊃5 5 6 6 7 7 8 8)⊃B

  ⍝ Cumulate the moves and rearrange the indices:
   Z[+\E,((6⊃B),2/⌽⍳R)/(↑B)⍴B[C[D;]]]←Z

  L01:Z←(2⍴R)⍴Z ⍝ Form matrix.
   →⍳8>16|L     ⍝ Done if involute.
   Z←(¯1+A)-Z   ⍝ Subtract if evolute.
    ∇
```

Roy A. Sykes, Jr. is a regular columnist for APL Quote Quad, and is the President of Sykes Systems, Inc., an APL consulting firm. He can be reached at "roysykes@netcom.com".