Sparse Matrix Technology Tools in APL

Ferdinand Hendriks
Wai-Mee Ching

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, NY 10598, U.S.A.

ABSTRACT

We have implemented sparse matrix technology tools in APL. Such tools have been conspicuously scarce, because APL has not been the language of choice for solving boundary value problems governed by partial differential equations. But when carefully coded, APL is able to tackle problems governed by partial differential equations in a way that ads flexibility and, on account of its compactness, maintainability. The main criticism of APL in numerically intensive applications has been execution speed. APL compilation addresses this drawback and shows factors of speed improvement of better than about three. Timings will be presented for some benchmark elliptical boundary value problems, both for interpretive and compiled APL.
Examples are given of common tasks that are encountered in conjunction with the finite element method, such as determination of the symbolic form of the stiffness matrix, and the more universal task of solution of a sparse (symmetric) set of equations using the conjugate gradient method.

INTRODUCTION

The use of APL in such numerically intensive applications as the solution of partial differential equations (PDEs) is infrequent. Among the few examples is Foote et al.[1], who solve a parabolic PDE that arises in finance using finite differences. APL can play a role in the area of large scale field problems, which is now dominated by compiled languages such as FORTRAN and C. APL is liked for its elegance, expressivity and compactness which adds maintainability. From the point of view of the programmer, writing a finite element code in APL complete with pre- and post-processor functions then becomes a task that can be tackled by a very small programming team; perhaps even a single individual. The present paper presents some of the essential tools that are needed for solution of sparse, linear systems and other functions that are encountered in the finite element method. To tackle sparse systems, rectangular data structures must be replaced by other, more efficient ones, such as lists. Many of the primitive functions of APL that operate on rectangular arrays or those that create them, such as the $\circ.$ constructs, must be reformulated. Another example is the ⌹ function, suitable for dense matrices, but not for large sparse ones.*. The tools we developed are expressed in VSAPL. APL2 offers significant advantages on which we hope to report in the future. For example, a finite element mesh can be expressed as the single variable *MESH* containing the nodal coordinates expressed as floating point data, and the connectivity matrix which is composed of integers.
To test the various techniques, we discuss a benchmark problem and its solution for several, increasingly finer, spatial discretizations. The sparse solver of the benchmark problem will be run both in interpretive APL and compiled APL. This provides insight in the performance gains of compiled APL code as a function of problem size. As finite element problems go, the largest problem discussed (about 4000 degrees of freedom) must still be considered small in an industry where it is not unusual to see finite element problems with several hundred thousand degrees of freedom.

BENCHMARK PROBLEM

A suitable, and technically important, benchmark problem for sparse APL-based tools and APL compilation is the following elliptical partial differential equation

$$\nabla . \varepsilon \nabla \Phi = F \tag{1}$$

in the Cartesian x,y plane, on the domain $-1 \leq x \leq 1$, $-1 \leq y \leq 1$. Homogeneous Dirichlet boundary conditions are imposed as follows

$$\Phi(x,1) = 0 \tag{2}$$

$$\Phi(x,-1) = 0 \tag{3}$$

and periodicity conditions are imposed elsewhere

$$\Phi(-1,y) = \Phi(1,y). \tag{4}$$

* This fact would make it attractive to evaluate an APL-based boundary element method. The latter gives rise to a linear system with a smaller, but full, matrix to which ⌹ can be applied efficiently.

The diffusivity ε is given the value 10 within the unit square around the origin, and 1 elsewhere. The forcing function shall simply be $F = -10$ throughout. Physically, this problem corresponds (for example) to the deflection Φ under uniform pressure of a thin elastic membrane with a central square region which is much stiffer than the rest. The problem becomes more ill-conditioned (stiff in a mathematical sense), the greater the disparity in ε. The problem can also be of electrostatic origin. Φ then corresponds to the electrostatic potential, ε to the dielectric constant and F corresponds to a space charge. We solve this problem using simple linear triangular finite elements on the grid shown in Fig.1.

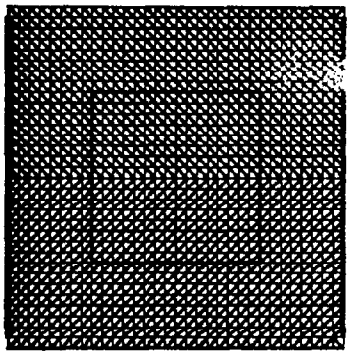

Fig. 1. Finite element benchmark grid

A surface plot of the solution Φ is shown in Fig. 2.

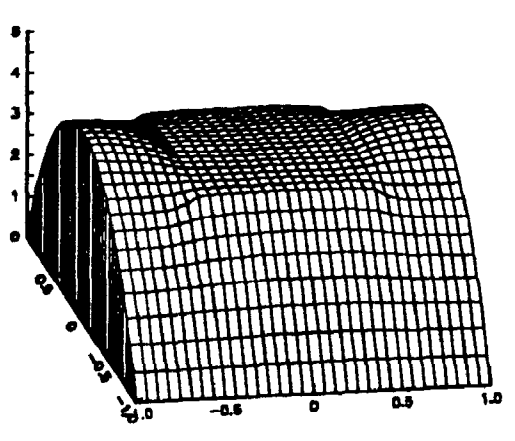

Fig. 2. Solution to benchmark problem

The Galerkin formulation[2] is used to represent the partial differential equation as a set of linear algebraic equations for the value of Φ at the nodes. The coefficient matrix, often-called "stiffness matrix," is symmetric and positive definite. That makes it a prime candidate for iterative solution techniques, such as the conjugate gradient method or one of its many variations[3]. Conjugate gradient techniques have become popular for a select class of very large problems that lend themselves well to parallelization. The success of this algorithm depends heavily on a sparse matrix-vector multiplier. Because of its inherently parallel notation it should not be surprising that these algorithms lend themselves very well to coding in APL. We demonstrate this in detail in the following sections.

SPARSE MATRIX TECHNIQUES

Without sparse storage techniques for linear systems stemming from partial differential or integral equations it would be impossible to solve problems of practical significance, i.e. of sufficient size. We adopt the following row-wise storage scheme for sparse matrices. The sparse matrix is characterized by two vectors *BSP* and *ASP*. *BSP* is the catenation of *B1*, *B2*, *B3* where *B1* is the number of rows, *B2* are the number of non-zero entries in each row, and *B3* are the column numbers of the nonzero entries of *A*. The non-zero floating point entries of the sparse matrix are contained in variable *ASP*. Note that the integers that make up *BSP* completely determine the structure (also called symbolic form) of the sparse matrix. The structure of the matrix in turn depends only on the local-to-global map (connectivity matrix) *LGΔMAP* of the finite element grid. *LGΔMAP* has 3 columns. Each row contains the three nodes that comprise each triangular element, mentioned counterclockwise. The function *COMPΔBSP* creates the global variable *BSP* which defines the symbolic sparse system.

```
BSP←COMPΔBSP LGΔMAP;F;M;IM
⍝ Computation of the symbolic structure of
⍝ a sparse linear system when the
⍝ connectivity matrix LGΔMAP is given.
⍝ ---------------------------------------
M←⌈/,LGΔMAP ⍝          Number of rows
⍝                     (largest node no.).
F←DROPΔREDUNDANTΔEDGES COMPΔEDGES⌽LGΔMAP
⍝                     Compute all
⍝                     (single) edges.
IM←⍳M ⍝               Ordered row of nodal
⍝                     indices.
F←((F,[1]⌽F),[1]⌽IM,[0.1] IM),
                      [1]⌽IM,[0.1] M⍴1+M
⍝                     2 col. matrix.
⍝                     Col 1: Nodes from
⍝                            edge matrix.
⍝                     Col 2: Nodes that
⍝                            connect to
⍝                     those in column 1.
⍝                     Assure 2nd column is
⍝                     piecewise up-ordered:
F←F[⍋(2⍴1+M)⊥⌽F;] ⍝   NOTE:
⍝                     F is also the SPARSE-
⍝                     TO-GLOBAL map.
⍝                     1st term of B3 has
⍝                     location F[1;].
⍝                     2nd term of B3 has
⍝                     location F[2;].
⍝                     3rd  ..  .. .. ...
⍝                     ..... F[3;].
⍝                     etc.
```

```
⍝                    SPARSE TO GLOBAL MAP.
⍝
BSP←M,((SUMΔOFΔCLUSTERS F[;1])÷IM),F[;2]
⍝                    Explanation:
⍝ B1←M ⍝             Number of rows
⍝ B2←(SUMΔOFΔCLUSTERS F[;1])÷IM
⍝                    Number of nonzero's in
⍝                    each row.
⍝ B3←F[;2] ⍝         Cols. of nonzero's,
⍝                    ordered per row.
⍝ BSP←B1,B2,B3 ⍝     This is BSP.
```

The functions *COMPΔEDGES*, *DROPΔREDUNDANTΔEDGES*, *SUMΔOFΔCLUSTERS* are given in the Appendix. Their purpose is self-explanatory and further documented within the function itself.

Note the use of ⍤ and ⍤⁻¹ to perform mathematical operations on matrices using one of the rows as keys. We often use this technique to map mathematical operations on two and more dimensional objects, such as a 2 column matrix of integers representing edges, onto a single dimension. See for example the function *DROPΔREDUNDANTΔEDGES*, shown in the Appendix, which removes duplicate edges from a set of bi-directed edges. This is useful for display of a finite element grid without displaying each edge twice.

After obtaining the symbolic form of the sparse linear system, the numerical values of all terms of the linear system are determined. This is done in a process called assembly. Again, in APL it is imperative for performance to do the assembly for all elements at once. The final step is to impose the boundary conditions (or other constraints that may be imposed), which amounts to modification of the sparse system. Sometimes, knowledge of the boundary conditions is used to decrease the size (condense) the linear system. At this stage the sparse linear system must be solved. The APL implementation of the conjugate gradient method without pre-conditioning is shown below.

```
R←SPA SPARSEΔMINΔCG RHS;X;D;RO;RN;T;BETA;
                    I;RESI
⍝                   Sparse implementation
⍝                   of the CONJUGATE
⍝                   GRADIENT method.
⍝                   No pre-conditioning.
⍝                   SPA is the sparse
⍝                   representation BSP,
⍝                   ASP of a square
⍝                   matrix A, side N.
⍝                   RHS is a vector rhs of
⍝                   shape N, (Nx1),
⍝                   or (1xN).
⍝ ------------------------------------------
X←RHS ⍝             Starting vector
I←0 ⍝               Initialize iteration
⍝                   number.
RO←SPA SPARSEΔFΔDER X ⍝ Residue (old):
⍝                   A x - b
RESI←(+/(((SPA SMVP X)-RHS)*2))*0.5
⍝                   Rms residue.
RESI←RESI÷(+/X*2)*0.5 ⍝ Normalized
⍝                   residue.
D←-RO ⍝             Find initial direction
T←-((⍉D)+.×RO)÷((⍉D)+.×(SPA SMVP D))
⍝                   Scalar excursion
⍝                   along D.
T←(⍴D)⍴T
X←X+T×D ⍝           New estimate for
⍝                   location of minimum
⍝                   of objective fcn.
⍝ --------------------
L1:I←I+1 ⍝          Inherent conjugate
⍝                   gradient loop.
RN←SPA SPARSEΔFΔDER X ⍝ New residue
BETA←((⍉RN)+.×RN)÷((⍉RO)+.×RO) ⍝ Note:
⍝                   need old residue to
⍝                   compute
BETA←(⍴D)⍴BETA
D←(-RN)+BETA×D ⍝    New direction
T←-((⍉D)+.×RN)÷((⍉D)+.×(SPA SMVP D))
⍝                   Find excursion along
⍝                   new direction.
T←(⍴D)⍴T
X←X+T×D ⍝           New X.
RO←RN ⍝             Latest becomes
⍝                   old residue.
RESI←(+/(((SPA SMVP X)-RHS)*2))*0.5 ⍝ Rms
⍝                   residue.
RESI←RESI÷(+/X*2)*0.5 ⍝ Normalized residue
→L1×⍳(I<1↑⍴RHS)∧(EPSILON<,RESI) ⍝ EPSILON:
⍝                   convergence criterion
⍝                   Go do next cycle as
⍝                   long as no. of
⍝                   iterations  less than
⍝                   size of linear system
⍝                   AND normalized residue
⍝                   larger than the
⍝                   convergence criterion.
R←X ⍝               R is soln. if there is
⍝                   convergence.
```

The loop within the function is inherent to the algorithm and cannot be avoided. We see that a critical function that occurs in the conjugate gradient algorithm is *SMVP* (sparse matrix-vector multiplication). The function *SPARSEΔFΔDER* is problem-dependent. It computes the residue of the yet-unsolved linear system. It calls *SMVP* once. Using the row-wise sparse storage scheme, an APL version of such a function is as follows.

```
R←MA SMVP V;ASP;BSP;B1;B2;B3
⍝                   "Sparse Matrix Vector
⍝                   Product." MA is BSP,
⍝                   ASP. V can be either
⍝                   column or row vector.
⍝                   Returned is a vector:
⍝                   ⍴⍴V is 0.
⍝                   ASP and BSP are the
⍝                   sparse representation
⍝                   of the matrix which
⍝                   multiplies vector V.
⍝                   The possible sparsity
⍝                   of the vector V is not
⍝                   exploited.
⍝ ------------------------------------------
B1←1↑MA ⍝           No. of rows in matrix.
B2←B1↑1↓MA ⍝        No. of nonzero's
⍝                   in each row.
B3←(+/B2)↑(1↓B1)↓MA ⍝ Column no's of
```

```
⍝                          nonzero's.
A⍋SP←(-⍴B3)↑MA ⍝           Non-zero's themselves.
R←(⍳B1)REPLICATE B2 ⍝      Replicate left arg,
⍝                          right arg B2 times.
R←1↓(R,1↑R)-¯1⌽R,1↑R ⍝     In-line DDIF
⍝                          function, without TOL
R←(0≠1,R)/0,+\A⍋SP×V[B3] ⍝ Carry out the
⍝                          multiplication.
R←¯1↓1↓(R,1↑R)-¯1⌽R,1↑R ⍝  ¯1↓DDIF R
```

The first four executable lines of the function dissect the sparse representation. The product is created in the last four lines with the help of the utility function *REPLICATE*, which replicates each member of its left-argument right-argument times each. This function is also central to matrix-vector multiplications that must be done without loops over all finite elements. If there are N finite elements, one often has to form products of type A.V, where ρA is $N,3,3$ and ρV is $N,3,1$. The result has rank $N,3,1$. As far as we are aware, no efficient primitives exist to do such a "restricted inner product." (*RIP*). The following function serves as a substitute

```
R←A RIP B;N;I;J;K
⍝                          Restricted Inner
⍝                          Product.
⍝                          A has rank N,I,J;
⍝                          B has rank N,J,K;
⍝                          R has rank N,I,K.
⍝ ----------------------------------------
N←1↑⍴A
I←1↓1↓⍴A
J←¯1↓⍴A
K←¯1↑⍴B
R←+/[3](N,I,J,K)⍴((,A)REPLICATE
          (⍴,A)⍴K)×,2 1 3 4⍉(I,N,J,K)⍴B
```

Unfortunately, on large finite element problems the *REPLICATE* function tends to be a storage bottleneck.
APL2 allows a very elegant, compact expression of this restricted inner product, namely (except for a minor reshaping)

```
R←(⊂[2 3]A)+.×"⊂[2 3]V
```

where A and V may be taken as $A←(N,3,3)\rho F$ and $V←(N,3,1)\rho F$ where F is an arbitrary floating point number. But, both in terms of execution time and storage, the function *RIP* is still to be preferred over the latter, more concise expression. The *AII*[2] timing is a factor of approximately four in favor of *RIP*. Similar remarks apply to restricted outer products, equally important in finite element applications.

COMPILED AND INTERPRETED APL

When large codes mature to the production-level stage, the interpretive nature of APL inhibits performance, both in terms of storage requirements and execution speed. We present our experience with an experimental compiler developed by one of the authors, W.-M. Ching[4]. We concentrate our attention on a very computationally intensive part, namely the sparse linear solver based on the conjugate gradient algorithm described above. The APL-based conjugate gradient algorithm without any pre-conditioning is compiled and applied to the benchmark problem (1) through (4). Four cases are chosen, ranging from coarse to fine discretization, having 81, 289, 1089 as in Fig. 1, and 4225 nodes respectively. The interpretive runs are timed using *□AI*[2]. The compiler is fitted with its own timing device. The results are shown in Fig. 3. Note that both axes have logarithmic scales. The timings are for an IBM 3090-200E vector processor running APL2 (vector) under VM/XA. Several trends emerge from the timing data. As a rule of thumb, the compiled code speeds up a production run on our largest problems by about 50 percent. On small problems, the speedup is much larger. This is caused by the fact that the finite element code operating on few data is bound by the interpretive overhead. We should emphasize that the speed-up factors will be much larger than those reported here, when less efficient use is made of APL idioms; and vastly larger for non-APL-style APL programs. In some sense, the compiler timings are an indication for the quality of the APL code.

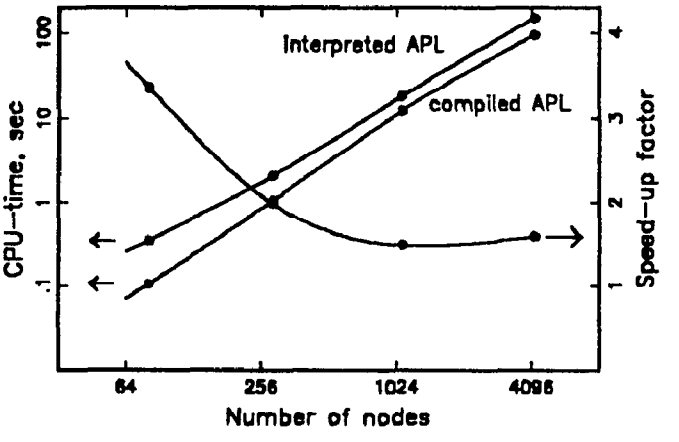

Fig. 3 Execution times of the conjugate gradient function.

CONCLUSION

We applied APL to an area of numerically intensive computing where it has been largely absent. A benchmark problem was chosen on a set of succesively refined grids. Sparse matrix technology tools in APL style were demonstrated. The performance of the conjugate gradient algorithm was measured both for interpreted and compiled APL. From this we conclude that APL is quite able to tackle PDEs provided it is carefully coded. Compilation offers significant performance gains, even as the problem size gets larger. That the compiled code still outperforms the interpreter by 50% on the largest data set may come as a surprise. The reason is, we believe, that the APL370 compiler not only saves interpretation time, but also has its own, built-in garbage collection mechanism, thus avoiding garbage collection during run-time.

REFERENCES

1. W.Foote, J. Kraemer, G. Foster, "APL2 Implementation of Numerical Asset Pricing Models," APL88 Conf. Proceedings, APL Quote Quad, Vol. 18, No. 2, Dec 1987., ACM Press, pp 120-125.
2. H.R. Schwarz, "Finite Element Methods," Computational Mathematics and Applications, Academic Press, 1988, Chap. 1.
3. G. Radicati di Brozolo and M. Vitaletti, "Conjugate Gradient Subroutines for the IBM 3090 Vector Facility," IBM J. Res. Develop. Vol. 33, No. 2, March 1989, pp. 125 - 135.
4. W.-M. Ching and A. Xu, "A vector Code Back End of the APL370 Compiler on IBM 3090 and Some Performance Comparisons," APL Quote Quad, Vol. 18, No. 2, Dec. 1987, ACM Press, pp 69-76.

APPENDIX

The following is a set of functions called by those discussed in the text.

```
R←SUM∆OF∆CLUSTERS A;S
⍝                       Let  A  be a string of
⍝                       real numbers. This fcn
⍝                       returns the sum of the
⍝                       clusters in A.
⍝                       Example:  A:   9 .1 .1
⍝                       ¯.1 ¯.1 3 3  4 7 7 5 0
⍝                              R:   9 .2 ¯.2
⍝                       6 4 14 5 0
⍝ ---------------------------------------
→J1×⍳0=+/|R←DDIF A ⍝ Check if all terms
⍝                       identical
S←⍳0 ⍝
⍎(0=(|¯1↓A)+1↓A)/'S←0' ⍝ Special case: A
⍝                       starts and ends with 0
R←¯1↓DDIF(0≠1,R)/0,+⍀A
R←R,S
→0
J1:R←+/A


R←DDIF T;TOL
⍝                       T is considered a
⍝                       circular string.
⍝                       This fcn returns the
⍝                       difference of an
⍝                       element in  the string
⍝                       T and its LEFT
⍝                       neighbor.
⍝                       Example: T ← 1 2 3 4 5
⍝                       R : 1 1 1 1 ¯4
⍝ ---------------------------------------
R←1↓(T,1↓T)-¯1⌽T,1↓T


R←COMP∆EDGES A;N
⍝                       Compute the edges
⍝                       defined by A.
⍝                       The result is a 2
⍝                       column matrix  of node
⍝                       numbers, each  row  of
⍝                       which defines an edge.
⍝                       Note: most edges  will
⍝                       occur  twice  because
⍝                       elements are adjacent.
⍝                       Only  boundary  edges
⍝                       occur once.
⍝ ---------------------------------------
⍝                       Assume A is
⍝                       a single row of
⍝                       indices. If A is multi
→L1×⍳(1≠ρρA) ⍝          dimensional, jump.
R← ¯1 0 ↓⍉A,[0.1] 1⌽A ⍝ Note  the  drop of
⍝                       Last row.
→0
L1: ⍝                   Check if A has 2
⍝                       Dimensions.
→L2×⍳2≠ρρA
→L3×⍳3=¯1↑ρA ⍝          If 3 columns, go to L3
R←A ⍝                   If two columns, (1D
⍝                       finite elements.)
⍝                       Connectivity matrix is
⍝                       identical
→0 ⍝                    to the matrix of edges.
L3: R←R[(N⍴(1,1,1,0))/⍳N←1↑ρR←⍉R,
                        [0.1] 1⌽R←,A, 0 ¯2 ↑A;]
→0
L2: 'Unexpected no. of arguments'


R←DROP∆REDUNDANT∆EDGES EDGES;NN;R1;R2;
                                DRPS;RIR
⍝                       Function  that  drops
⍝                       redundant  edges  from
⍝                       EDGES.
⍝ ---------------------------------------
NN←2⍴1+⌈/,EDGES ⍝
R1←NN⊥⍉EDGES ⍝           Consider each edge
⍝                       the encoded
⍝                       representation of
⍝                       some larger,
⍝                       SINGLE number. R1 is
⍝                       a string of decoded
⍝                       edges.
R1←DROP∆EXTRA R1 ⍝      Discard
⍝                       copies of EDGES
R2←NN⊥⍉⌽EDGES ⍝         Do the same for
⍝                       "flipped" edges.
RIR←R1 ⍳ R2 ⍝           Where in R2 does R1
⍝                       Occur?
DRPS←(RIR>⍴RIR)/RIR ⍝
R←⍉NN⊤R1[(⍳⍴R1) EXCEPT DRPS] ⍝ Unique
⍝                       directed edges


R←N EXCEPT M
⍝                       Delete M from N.
⍝                       Entries of R will be
⍝                       UNIQUE.
⍝ ---------------------------------------
R←(~N∈M)/N
```

```
R←DROPΔEXTRA A;S;IN
⍝                    This function removes
⍝                    redundant entries from
⍝                    vector A.
⍝                    For example, if A is
⍝                    1 1 1 3 3 2  2 2 2 7 7
⍝                                   7 0 0
⍝                    then R is  1 3 2 7 0.
⍝                    Note that the order
⍝                    of the unique entries
⍝                    is maintained.
⍝ ------------------------------------------
IN←⍋A ⍝              This function keeps
R←A[IN] ⍝            order and drops
⍝                    redundant entries
S←1,1↓R≠¯1⌽R ⍝       starting at the right.
R←S/R
R←R[⍋S/IN]


R←S REPLICATE M;H
⍝                    Replicate the entries
⍝                    in S, M times each.
⍝                    Vectors S and M have
⍝                    equal length. M is a
⍝                    vector of integers.
⍝                    Example:
⍝                           M←1 1 4 3 2 0 3
⍝                           S←1 8 9 3 2 1 7
⍝                    Gives  R:  1 8 9 9 9 9
⍝                           3 3 3 2 2 7 7 7
⍝
⍝ ------------------------------------------
H←0≠M ⍝              Flag nonzeros in M.
M←H/M ⍝              Restrict M to nonzeros
S←H/S ⍝              Restrict S to nonzeros
R←(+/M)⍴0 ⍝          Initialize result.
R[¯1↓1+0,+\M]←¯1↓0,DDIF S ⍝  Insert jumps
⍝                    in R to give the
⍝                    "derivative" of the
⍝                    result. Integrate to
R←(1↓S)++\R ⍝        get the final answer.
```