APL programming without tears
→ It is time for a change

P. Naeve, B. Strohmeier,* P. Wolf
Fakultät für Wirtschaftswissenschaften
Universität Bielefeld
Germany

*email:bstrohm@erasmus.hrz.uni-bielefeld.de

ABSTRACT

To overcome the unreadability of traditional *APL*-code Knuth's idea of literate programming is adapted to *APL*. The *APL*2WEB system of structured documentation is introduced as a new way of *APL*-programming. An example is given to highlight the merits of this combination.

POINT OF VIEW

This paper is written from a statisticians viewpoint as well as from the viewpoint of a computer scientist. In our daily work, we are often concerned with statistical and computational questions. So, the choice of *APL* as our favourite programming language is not hard to understand, since most of the statistical packages do not fit our wishes. One of the most valuable merits *APL* offers to statisticians is its ability to transform mathematical formulas almost directly into *APL*-code.

keyword	formula	*APL*-code
mean	$\bar{x} = \frac{1}{n}\sum x_i$	`MEAN←(÷n)×+/X`
eq-system	$x = A^{-1}b$	`X←(⌹A)+.×B`
		`X←B⌹A`
least squares	$\hat{\beta} = (X'X)^{-1}X'y$	`BETA←(Y+.×X)⌹(⍉X)+.×X`

However, if one has many functions and workspaces written in *APL* the need for a proper documentation of these becomes more and more urgent for reasons every *APL*-er has certainly experienced: After a while, one cannot read and understand the own functions. Moreover, you have difficulties to decide, whether other people's functions work correctly or not.

The first point is annoying for your and if you are self-confident you will be convinced that your statements do exactly what they are supposed to do. The second point struck us whenever we had to debug *APL*-programs written by others.

We really appreciate *APL* and its features, but there are people who don't. Why? Let us look at some points which might be considered to be disadvantages of *APL*. Our intention is to show how these can be overcome, so that only the benefits of *APL* will remain.

The advantages of *APL*, the power of its primitive functions, the flexible data structures especially in *APL*2 and so on are well known but there must be a reason why not everybody appreciates *APL*. So, what are the drawbacks of *APL*? From our view, the main items are:

- The power of *APL*'s primitive functions is not understood, e.g. we often see unnecessary loop constructions and clumsy data structures.
- People do not get familiar with the hieroglyphs of *APL*.
- The traditional process of writing *APL*-code is awkward.

The first two points are the most frequent arguments against *APL*. But it can be argued that one has to learn all the features of a language if you want to use it efficiently and as far as the hieroglyphs are concerned, the critics should remember that the mathematical language mainly consists of hieroglyphs, too, and only some very lazy people are complaining about that.

The third point is new and our goal is to describe how the process of writing *APL*-programs can be improved so that our beloved *APL*-language becomes even more valuable. Let us look at the traditional process of writing *APL*-code: Many *APL*-statements result from the trial-and-error principle: You know what you want to get and you begin to permute primitive functions until the desired result is reached. Nothing is wrong with that, the *APL*-interpreter features this way of programming, but you should be able to explain the final statement and you should *do* it to achieve two goals:

- The statement becomes clear for other human beings.
- You get deeper insights in what you did and in most cases you can optimize the statement.

For example take the following function which may be considered as the result of true hackerdom.

```
      ∇ z←l fkt r;a;b
  [1] z←⍬ ⋄ a←(⌽⍳l)⊤r
  [2] b←¯1↑a
  [3] z←b,z+b≤z
  [4] a←¯1↓a
  [5] →(0<⍴a)/2
      ∇
```

Who can tell what this function does? Though it is rather short it is hard to find out. Can you read it? Then, find the idea behind the algorithm. This function works but imagine it has a hidden bug. What would you do then? Do you think there is any possibility to debug it without knowing the idea behind it?

The reader might be tempted to say: "I am using the *lamp* in all my programs frequently. The data structures and the function calling syntax are perfectly clear. All my programs are well documented."

Our answer to this is we do not need "traditional documentation", since documentation is an act which is performed after the programs are written. First comes the program, then the documentation. People who document a program tend to overlook the idea behind the code and that is why we believe that programming and documentation must be performed simultaneously.

Furthermore, the way people think is not necessarily the sequential way the *APL*-interpreter expects the statements. It must be possible to do the things in an arbitrary order. The example below gives an impression.

The reader might insist: "In *APL*, I can debug every function stepwise. I will type in all the staments step by step in immediate execution mode and finally I will find out what the function does."

Yes, he might find out what the function does but is he sure to understand the underlying idea? Our experience shows that you can not understand a top-down idea by using bottom-up techniques.

How can anyone be sure that a function works correctly without being able to understand it? It is obvious that even moderately sized projects are running into trouble if they are not well documented. Even worse, the *APL2* extensions of data structures, primitive functions, and operators lead to greater difficulties in understanding the ideas if they are not described properly.

The point we want to make is: *It is necessary to write down the ideas.*

To avoid these disadvantages of *APL*, we have to find another way to handle ideas than the traditional one to become more confident in our own and other people's functions and to achieve better programs.

IDEAS MATTER

In the last section, we listed several reasons which might cause so many people to dislike *APL*. Here, we want to concentrate on the most essential one. In doing so we will be able to offer a way out.

Most people will possibly remember Iverson's [?] paper *Notation as a Tool of Thought* where he claimed that

> *APL* is a notational tool!

So, *APL* should be classified as a notational tool capable of being executed (interpreted) by a computer system.

Let us change the scene for a moment. Computer scientists are deeply concerned about the state-of-the art in computer programming. In spite of all efforts, there still are problems with program documentation. One might even say there is nothing like a well documented program. Can the *APL*-community deny the truth of this statement when changing "program" to "function"?

Yet one distinguished computer scientist offered something like the magic stone to improve the situation. In 1984, Donald E. Knuth wrote in his paper on *Literate programming [?]:*

> Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do.

The essence of this statement is a plea for a radical change in attitude. Programming has to be seen as a communication process between human partners. They "talk" about the computer but not to the computer. So they can concentrate on the main task which is problem solving, not programming. The computer is just another but important tool for this process. This should be made explicit. Bringing this into action calls for a change in paradigm within the computer science community. This might be the explanation for the fact that Knuth's ideas were not received by the programmers community as we would have predicted. As stated by Kuhn [?] the scientific communities are always very reluctant to change their paradigms.

But as we feel the urgent need for a change let us have a brief look at the consequences of Knuth' proposal.

→ Telling other people how we want a computer to do the job changes "programming" to "writing works of literature". That's why Knuth chose the title of his paper to be *Literate Programming*. But for our purposes we need more than just our "natural" language to express our thoughts. We must have access to the whole bunch of (scientific) languages such as mathematics, graphics, and so on.

→ But nobody wants to do a job twice. Having told my colleague how I want the computer to do the job should suffice. One cannot see any reason why one has to switch to "ordinary" programming afterwards. (Imagine the computer could listen to your conversation with your colleague. Then he should have got all he needed to proceed without any further information.) Therefore Knuth came up with his WEB-System which provides exactly this feature. All the user has to do is to finally break down his thoughts into pieces in such a way that they can be expressed in the programming language of his choice. As we will see this is supported by having the notion of a section which consists of two parts. The "comment"-part where all kind of languages are feasible and a "code"-part where a programming language is the only language allowed.

→ As Knuth [?] put it: *we understand a complicated system by understanding its simple parts, and by understanding the simple relations between those parts and their immediate neighbors. If we express a program as a web of ideas, we can emphasize its structural properties in a natural and satisfying way.* To achieve this the system must allow us to break the whole into pieces in a controlled way and to handle those pieces in an appropriate way, for instance referencing them. Expressed in computer science terminology what we need is support for the process of stepwise refinement. But top-down should not be mandatory. The system must tolerate other ways of thinking i.e. bottom-up, too.

→ Up to now, documenting a program boiled down to an ex post documentation of the code — "K is a loop variable" is a famous example. Now, we can get an up-to-date documentation of more than the code. It is a documentation of all the ideas and decisions which lead to that code. It is really striking that computer scientists don't see this perspective immediately.

Why don't we melt Iverson's and Knuth's philosophies? A thorough and far reaching notational tool such as *APL* combined with a powerful system like WEB should result in an even mightier system. We call it *APL2*WEB[1]. The rest of the paper is a demonstration that such a system does exist. That part of our paper depends on the work of our colleague Christoph von Basum. Luckily for us he implemented more than a prototype of such a system as a by-product of his Ph. D.-thesis [?].

But before we give an example we briefly like to stress that *APL2*WEB is an appropriate answer to many problems. These other problems will make the new attitude towards "programming" even more appealing. To mention just two:

→ More and more things are handled by the computer. Gone are the days when statisticians were seen walking around with statistical tables under their arms. Now, they depend on their statistical software which provides percentage points, P-values or what ever they need. But in the old days almost all of them used the same kind of tables, notably *Biometrika tables for statisticians* [?]. So, everybody had the same numbers computed according to well documented formulas and procedures. Do you know how your statistical software does it? Why do you trust your software? Establishing confidence and trust in a program (function) is an emerging problem.

→ Many journals nowadays publish algorithms, so does *APL* QUOTEQUAD. Far too often, later a correction has to be added, so

[1] This name was chosen by C. v. Basum.

in *APL* QUOTEQUAD, too. Someone simply did some mistyping when copying (which here means retyping) the algorithm again and again. *APL*2WEB will improve this situation for now we have an unique source of information, the *APL*2WEB-document.

AN *APL*2WEB EXAMPLE

We try to demonstrate by example that an *APL*-solution gets readable after translating it into *APL*2WEB. Moreover, we are sure that the reader will have confidence in the resulting code afterwards.

As example, we chose the program **INFDIV** that was proposed by Danial[?]. This example has two properties: It has been discussed at an *APL*-conference and it deals with a problem belonging to the field of statistics. So, we hope it will be of interest for *APL*-programmers as well as for statisticians.

The paper of Danial can be divided into three parts. The first part discusses aspects of the statistical theory being involved. The second one contains the definition of the function **INFDIV** and the last shows a typical result of using **INFDIV**.

For there is a large gap between explanation and code — part one and two — you have to believe in the correctness of the code or you have to invent the function once more. So we did, and here is the result.[2]

1. infdiv – checking infinite divisibility of UGWD.

This function is similar to the function **INFDIV** *written by Danial (1989).*

The probability function of the univariate generalized Waring distribution (UGWD) is given by

$$p_i = \frac{\Gamma(a+\rho)\Gamma(k+\rho)}{\Gamma(a)\Gamma(k)\Gamma(\rho)}\,\frac{\Gamma(a+i)\Gamma(k+i)}{\Gamma(a+k+\rho+i)i!} \qquad i = 0, 1, \ldots$$

The property of the infinite divisibility can be investigated in terms of the series $\{K_i\}$ and $\{\pi_i\}$ that are defined by the following formulas:

$$K_i = \frac{p_i}{p_{i-1}} \qquad i = 1, 2, \ldots$$

$$\pi_i = ip_i^* - \sum_{j=1}^{i-1} \pi_{i-j}p_j^* \quad \text{and} \quad p_i^* = \frac{p_i}{p_0} \qquad i = 1, 2, \ldots$$

If $\pi_i \geq 0$ for $i = 1, 2, \ldots$ the discrete distribution under consideration is infinite divisible (Katti (1967)). Danial (1988) showed that a sufficient condition for infinite divisibility is that $\{K_i\}_{i=m,m+1,}$ forms a monotone increasing sequence and $\pi_i \geq 0$, $i = 2, 3, \ldots, m+s, s \geq 0$, where s is the smallest integer for which $p_1/p_0 \leq p_{m+s}/p_{m+s-1}$. For details, see the references.

The program **infdiv** computes K_i and π_i for $i = 1, \ldots, N$.

References: Danial, E. J.: *APL* as a tool of research for the mathematical scientist, *APL* QUOTEQUAD, vol. 19, no. 4, 1989.

Danial, E. J.: Generalization to the sufficient conditions for a random variable to be infinite divisible, Probability and Statistics Letters, vol. 6, no. 4, 1988.

Katti, S. K.: Infinite divisibility of integer-valued random variables, Annals of Mathematical Statistics, vol. 38, no. 3, 1967.

[2]Due to the restriction of the \twocolumn layout some lines of *APL*-code were split by TeX.

2. At first, let us divide the function[3] into input, computation, and output explicitly.

```
∇ infdiv  2
⟨read parameters to choose a UGWD distribution and the
     number of values to be computed      5⟩
⟨compute the required number of Kᵢ's and πᵢs      9⟩
⟨construct a nice output      4⟩
```

infdiv appears in sections 1, 4, 5, and 10.

3. Before refining one of these parts we should fix names and data structures for the most important quantities:

APL-variable	structure	meaning
a	scalar	parameter a of the UGWD
k	scalar	parameter k of the UGWD
rho	scalar	parameter ρ of the UGWD
N	scalar	number of π_i's and K_i's to compute
p_star	vector, length N	$(p_1^*, \cdots, p_N^*)$
pi	vector, length N	$(pi_1, \cdots, pi_N)$
K	vector, length N	$(K_1, \cdots, K_N)$

They are defined as local variables:

```
⟨Local Variables of infdiv(2)      3⟩ ≡
a,k,rho,N,p_star,pi,K
```

See also section 7.

4. The Output of **infdiv**. It is easy to arrange the vectors **pi** and **K** as columns of a matrix. To increase readability the first column shows the index of the vector elements.

```
⟨construct a nice output      4⟩ ≡
⎕←'The columns of the following table contain:
    (⍳N), pi, K',[.1]'¯'
(⍳N),pi,[1.1]K
```

This code is used in section 2.

5. The Input of **infdiv**. The input module asks for the parameters of the UGWD and the number of values that should be computed.

```
⟨read parameters to choose a UGWD distribution and the
     number of values to be computed      5⟩ ≡
'Value of parameter a of the UGWD
    distribution?' ⋄ a←⎕
'Value of parameter k of the UGWD
    distribution?' ⋄ k←⎕
'Value of parameter rho of the UGWD
    distribution?' ⋄ rho←⎕
'How many terms should be computed?' ⋄ N←⎕
```

This code is used in section 2.

6. Computation of the K_i's and π_i's. By replacing the probabilities we get

[3]Strictly speaking **invdif** is a defined sequence. As mentioned below it is good style to introduce the header of a function or defined operator in a separate section.

$$K_i = \frac{\dfrac{\Gamma(a+\rho)\Gamma(k+\rho)}{\Gamma(a)\Gamma(k)\Gamma(\rho)} \dfrac{\Gamma(a+i)\Gamma(k+i)}{\Gamma(a+k+\rho+i)i!}}{\dfrac{\Gamma(a+\rho)\Gamma(k+\rho)}{\Gamma(a)\Gamma(k)\Gamma(\rho)} \dfrac{\Gamma(a+i-1)\Gamma(k+i-1)}{\Gamma(a+k+\rho+i-1)(i-1)!}}$$

Due to the recursive property of the Γ-function, this can be simplified to:

$$K_i = \frac{(a+i-1)(k+i-1)}{(a+k+\rho+i-1)i} \qquad i = 1,2,\cdots$$

Keeping the required indices in the variable `I` the translation of the formula into *APL* is easy achieved.

⟨ compute vector **K** 6 ⟩ ≡

```
K←(a+I-1)×(k+I-1)÷(a+k+rho+I-1)×I←⍳N
```

This code is used in section 9.

7. We want `I` to be a local variable!

⟨ Local Variables of **infdiv**(2) 3 ⟩ +≡

```
I
```

8. For computing the π_i values we need the p^*'s which are derived from **K** by:

$$p_i^* = \prod_{j=1}^{i} K_j$$

π_1 and p_1^* are identical. The other π_i's are successively computed in a simple loop according to $\pi_i = ip_i^* - \sum_{j=1}^{i-1} \pi_{i-j}p_j^*$. The variable `I` always points to the element of the vector **pi** that is just processed.

⟨ compute vector **pi** 8 ⟩ ≡

```
 pi←1↑p_star←×\K
 I←2
loop:pi←pi,(I×p_star[I])-(⌽pi)+.×(I-1)↑p_star
 →(N≥I←I+1)/loop
```

This code is used in section 9.

9. Putting the last refinements together will finish the job.

⟨ compute the required number of K_i's and π_i's 9 ⟩ ≡

⟨ compute vector **K** 6 ⟩

⟨ compute vector **pi** 8 ⟩

This code is used in section 2.

10. As each section is simple to understand we are sure that the communication with the reader is successful, confidence in the function **infdiv** is established, and that the resulting function runs without error.

The extracted — *(tangled)* — function **infdiv** looks like this:

```
 ∇ infdiv;a;k;rho;N;p_star;pi;K;I
⍝ ∇ Initially declared in WEB-file <DIVI.awb>,
    line 42; date: Mon Jan 04 17:52:58 1993
⍝   infdiv: 2
⍝ 3: a,k,rho,N,p_star,pi,K
⍝ 7: I
⍝ 2:, 5:
 'Value of parameter a of the UGWD
     distribution?' ⋄ a←⎕
 'Value of parameter k of the UGWD
     distribution?' ⋄ k←⎕
 'Value of parameter rho of the UGWD
     distribution?' ⋄ rho←⎕
 'How many terms should be computed?' ⋄ N←⎕
⍝ :5, 9:, 6:
 K←(a+I-1)×(k+I-1)÷(a+k+rho+I-1)×I←⍳N
⍝ :6, 8:
 pi←1↑p_star←×\K
 I←2
loop:pi←pi,(I×p_star[I])-(⌽pi)+.×(I-1)↑p_star
 →(N≥I←I+1)/loop
⍝ :8, :9, 4:
 ⎕←'The columns of the following table
     contains: (⍳N), pi, K',[.1]'¯'
 (⍳N),pi,[1.1]K
⍝ :4, :2
```

This canonical representation of the function should be used only for localizing errors but not for understanding the algorithm behind it.

REFLECTIONS ON THE EXAMPLE

There is no difficulty to construct a simple example and to ask: "Is there anybody who doesn't understand it?" However, we think that this example is not as simple as it seems. The apparent simplicity of the resulting function hides all the considerations done during the engineering process of creating the function. Try to comment on this function without the *APL*2WEB-paper in mind!

You see we are convinced that problem solving using the *APL*2WEB-style has many advantages. The following lists some of them as assertions:

- The problem formulation gets more precise if you have to write it down.
- A written solution can be approved step by step.
- A well-structured solution yields a better code.
- A thoroughly worked out solution contains less errors.
- The saving of debugging time exceeds the additional time for writing down the ideas of the solution process.
- Every reader will enjoy your *APL*2WEB-solution more than *APL*-listings and this is true for you, too.

Unfortunately, an important question is still open. The *APL*2WEB-system is something like a *tool* but tools do not explain *how* to use them properly at all:

> How has an *APL*2WEB-file to be designed to reach the aim of being understandable?

To start a discussion we now pose six rules we consider to be important. In our opinion adopting the ideas behind those rules will result in readable solutions. In the light of further experiences these rules may be modified and new ones will have to be added.

Rule 1: Define the problem and the point where the argumentation starts from in a clear form.

In the first section of the example some pieces of statistical theory and references for further information are given. So the context is well defined especially for statisticians. It is also defined which problem should be solved by **infdiv**.

Rule 2: Define layers and argument in terms of these layers.

Sometimes it is evident how to divide a problem into parts. So input, computation, and output are very often treated as different *modules*. (See also the corresponding sections of **infdiv**!) However in the case of larger problems adequate *levels of abstractions* have to be defined.

Rule 3: Divide as much as necessary, but not more.

The author of an *APL*2WEB-document should divide every problem in so many parts that the reader is able to understand the resulting sections as single steps of the whole story. Perhaps a *screen principle* is an orientation for the external appearance: *A section should not cross the screen margins of your editor!*

Rule 4: Use already defined solutions.

Every program designer has *libraries* filled with a rich variety programs, definitions, data structures, explanations, etc. — things that are often used. If the content of such a file is documented well, e.g. in *APL*2WEB-style, old sections will be easily activated for the problem at hand by copying them into the new paper.

Rule 5: Define data structures explicitly.

In the age of nested arrays confusion is caused not only by complicated combinations of operators and functions but also by subtle constructions of the variables and the meaning of their components. It seems to be very advantageous to discuss them in separate sections of the *APL*2WEB-paper. Although our example uses only simple objects, the summary of the variables will ease the reading.

Rule 6: The *APL*-code of a section has to match to the comment text of the same section.

This rule emphasizes that code and text of a section should be in balance. The code is the result of the considerations written down in the same section. Vice versa it is confusing if the reader is waiting for an announced *APL*-translation in vain.

SOME RULES OF C. V. BASUM

As already mentioned the WEB-system we are working with was designed by C. v. Basum. In his thesis [?] he discusses many aspects of writing WEB-documents. Some of the rules and short comments are taken from his thesis and listed below.

- "Any *APL*2 function can be divided into two parts. Its first part is the header, i.e., line 0 of a function. The other part consists of the statements that manipulate data. The first part is called the declarative element of a function definition, the second part is the procedural element.
  The distinction between these two basic elements of program design should be mirrored in the definition of any function. The *APL*2WEB system permits the separate documentation of function header and function body. A first rule is formulated from this observation."
  Devote a single section to specify a function's header.

- "The basic unit, the 'unit of composition,' is the *section* in WEB. The relations to other parts are interlaced by WEAVE automatically and optionally, enhanced by a user's manual index entries.
  The subject of a simple section must be comprehensible. This implies that sections will rarely extend over more than one page. As a rule of thumb, one should demand that no more than, say, five lines of *APL*2-code appear in a section. However, five lines of code in *APL*2 can be far too intricate to be still considered a unit."
  Keep a section self-contained.

- "A computer needs not to be asked whether it is willing to perform a job, the machine must obey the command. Telling another person what a computer is supposed to do means using the imperative as the grammatical form for the structure of a top-level description. Consequently, such a top-level description will begin with a verb.[...] The imperative seems to be the natural way to circumscribe a sequence of statements."
  Formulate a top-level description as an imparative.

TOTAL

To **take a closer look** we must wipe out the tears.

REFERENCES

[1] v. Basum C.: Making *APL* readable — A new direction for design, Lit Verlag, Münster, 1993.

[2] Biometrika tables for statisticians vol. I, Pearson E. S., Hartley H. O. (eds.), Cambridge University Press, 1954.

[3] Danial E. J.: *APL* as a tool of research for the mathematical scientist, *APL* QUOTEQUAD, vol. 19, no. 4, pp. 113–116, 1989.

[4] Danial E. J.: Generalization to the sufficient conditions for a random variable to be infinite divisible, Probability and Statistics Letters, vol. 6, no. 4, pp. 379–382, 1988.

[5] Iverson K. E.: Notation as a tool of thought, Comm. of the ACM, 23(8), pp. 444–465, 1980.

[6] Katti S. K.: Infinite divisibility of integer-valued random variables, Annals of Mathematical Statistics, vol. 38, no. 3, pp.1306–1308, 1967.

[7] Knuth D. E.: Literate Programming, Computer Journal, 27(2), pp. 97–111, 1984.

[8] Kuhn T. S.: The structure of scientific revolution, University of Chicago Press, 2nd ed., 1970.