

Functional Calculation 0 : Numerical Apéritif

Neville Holmes

School of Computing

University of Tasmania

Launceston 7250 Australia

Email: Neville.Holmes@utas.edu.au

ABSTRACT

This article is the first in a series expounding the joys of functional calculation. Functional calculation does with operations applied to functions and numbers what numerical calculation does with functions applied to numbers. The functional notation used as the vehicle in this series is provided by a freely available calculation tool called \mathcal{J} .

This article gives a detailed illustration of a few numerical uses of the notation supported by the \mathcal{J} interpreter. This explanation is preliminary to following articles, which will review some of that tool's support for numerical calculation. The illustration takes a simple function and shows how \mathcal{J} , or any other tool for functional calculation, can be used to investigate special values of the function.

CALCULATION

Calculation is an honourable and satisfying occupation with a long history and a rich tradition [1].

Relatively recent developments in electronic computation have led to the adoption of three classes of aids to calculation, namely • electronic or programmed calculators based directly on traditional calculation, • spreadsheet programs, and • complex mathematical systems. Electronic calculators are unreliable and very poorly designed [2]. Spreadsheet programs are very powerful and useful for the kind of calculation that suits them, but clumsy and limited otherwise. The complex mathematical systems are extremely powerful, but in an elephantine way.

The kind of functional calculation described in following articles is like an enormous but sympathetic extension of traditional calculation. The extension is particularly effective in dealing with the kinds of tables to which spreadsheet programs are suited but allowing conveniently for tables of many dimensions. It is also useful for collections of tables of different kinds. The extension is of surprising expressive power so that problems which would otherwise require the might of a complex mathematical system can be tackled simply.

This article will go through some simple numerical calculations step by step, explaining them in detail. Readers will get most benefit out of the description if they get their own interpreter and themselves carry out the calculations (and any others suggested by the examples) as they are being described.

The notation used here is called simply \mathcal{J} , and programs (for many of the more popular computing platforms) to evaluate expressions in the notation can be downloaded from Internet location

```
ftp://archive.uwaterloo.ca/languages/j
```

and help with it can be got from the Usenet group

```
news:comp.lang.apl
```

Because the programs which evaluate \mathcal{J} expressions as they are keyed in seem to interpret the notation, they are called *interpreters*. The \mathcal{J} interpreters that can be got for free from the Internet are old versions, but information about up-to-date versions, and much much more, can be found at

```
http://www.jsoftware.com/
```

The primitive functions, those functions with symbols for names, used in this article are few, and very many more such are provided by the interpreter, and are listed and briefly explained by the interpreter's help facilities.

The purpose of this article is to show how simple functions can be used simply to get useful results, the kind of results that show the interpreter in the light of a utility calculator. To allow this to be better appreciated, examples which involve the use of lists of numbers have been chosen.

The calculations used here as an example are based on an exploration of a function named f in the description. Any reader carrying out the example calculations can define f by keying in

```
f =: ^-o.
```

but this definition will not be further explained here because the expression used to define f is a functional expression and, as such, will be explained in later articles of this series.

FINDING ZEROES

Given the function f and a need to explore its behaviour, the first step is to name a useful list of test values, and then to apply the function to those values. Keying in the expression

```
nn =: i.11
```

uses the primitive function $i.$ (pronounced *integers*) to produce the first eleven nonnegative integers, to which the copula $=:$ (pronounced *gets* or *becomes*) gives the name nn .

The copula is only used for conferring names, and it is not a function as such. No result is shown because the interpreter is just naming. However any named

result can be looked at just by keying in its name, in this case

```
nn
0 1 2 3 4 5 6 7 8 9 10
shows nn to be a list of ten integers.
```

Some general remarks are appropriate at this point. The J interpreter takes in the expressions it evaluates indented, but the results are shown unindented. For the first expression shown above, the spaces between the `nn` and the `=:`, and between the `=:` and the `i.`, are not required by the interpreter, but have been put there to make the expression more readable. More than one space could have been used with same result, and spaces could have been put between the `i.` and the `11` without affecting the result. However, putting a space within the `=:` or the `i.` would have affected the result just as putting a space within the `11` would have, because the notation uses suffixed dots and colons to extend the ASCII character set, so that in effect `=` and `=.` and `=:` are three different characters.

The function `f` can now be evaluated for the argument `nn`, and its result displayed, by

```
f nn
1 _0.423311 1.10587 10.6608 42.0318 132.705 384.579 1074.64 2955.83 8074.81 21995
which shows a quite distinctive behaviour indeed.
```

However, the inconveniently small font forced on the printing above of the results of `f nn` makes it difficult here to appreciate fully the nature of that behaviour, so it is useful to define with scant comment

```
show =: ".&(0.1&" :)
which names a function that can be used as in
show f nn
```

```
1 _0.4 1.1 10.7 42 132.7 384.6 1074.6 2955.8 8074.8 21995
to force each value to be displayed with at most one decimal place. This is perhaps not so useful here, but will be so later.
```

One of the values in the result list is negative. Notice that the notation disambiguates the hyphen by representing the property of negativity by the sign `_` so that the hyphen can be kept to stand for the negation and subtraction functions.

To investigate how `f` behaves for negative arguments, the test values can have ten subtracted from them before applying `f` by keying in

```
show f nn-10
31.4 28.3 25.1 22 18.9 15.7 12.6 9.4 6.4 3.5 1
where the values displayed have been rounded to one decimal place by show.
```

To make completely plain what the argument value is for each result value, the *laminare* function `,:` can be used in

```
a ,: show f a =: nn-10
_10 _9 _8 _7 _6 _5 _4 _3 _2 _1 0
31.4 28.3 25.1 22 18.9 15.7 12.6 9.5 6.4 3.5 1
```

where the test values are temporarily named `a` so that they can be stacked by `,:`

on top of the results. Care must be taken in putting together expressions like the one keyed in above. For example, the copula `=:` gives the name to its left to its *predicate*, which is everything to its right, just as in English sentences the predicate of a verb is everything to its right. Providing there is a space between the `f` and the `a`, then there are two names, and the copula only uses the name next to it. If there's no space, the name is `fa` and there is no function `f` anymore. Similarly, the predicate of the laminate function is everything to its right, which is the result of applying `f` to the values of `a`.

The same technique can be used to stack the values of `nn` on the result of `f nn` by

```
nn ,: show f nn
0   1   2   3   4   5   6   7   8   9  10
1 _0.4 1.1 10.7 42 132.7 384.6 1074.6 2955.8 8074.8 21995
which shows quite different behaviour from that for negative argument values.1
```

The most interesting feature here is that the function has two zeroes, one between arguments of 0 and 1, and one between 1 and 2. These zeroes can now be explored one at a time, though the exploration of the second zero is here left as an exercise for the reader.

THE FIRST ZERO

The simplest zero to explore is the one between 0 and 1, and by

```
a ,: show f a=:nn%10
0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
1 0.8 0.6 0.4 0.2 0.1 _0.1 _0.2 _0.3 _0.4 _0.4
```

the eleven values between and including 0 and 1 are displayed. Because the restrictions of the ASCII character set require the `%` symbol to be used for the *divide* function, rather than the more usual `÷` symbol, the expression `nn%10` asks for `nn` to be divided by ten. Thus, the expression `nn%10` gives a list of values between 0 and 1 incrementing by tenths, and from the result of applying `f` to `nn%10` it can easily be seen that the zero lies between argument values of 0.5 0.6, about halfway between.

The limitations of the function `show` are evident here, but it can easily be redefined as

```
show =: ".&(0.2&":)
```

which then returns a second decimal place to sight in

```
a ,: show f a=:nn%10
```

¹On some versions of J you might not get the table of the example looking as neat as it does here. This will be because the host system is putting your numbers out using an unsuitable type font, one that does not space the letters equally, thus causing the misalignment in tables. You should change the font, presumably to the one that works best for me on the screen, that is, `ISIJ`, though `Courier` is the font used in this printed article for examples.

```

0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1
1 0.79 0.59 0.41 0.24 0.08 _0.06 _0.19 _0.29 _0.37 _0.42

```

so that the zero can be predicted to be slightly closer to 0.6 than to 0.5.

Not only are fractional values like 0.5 and 0.6 always displayed with a leading zero, but they must also be keyed in with one. Hence the next step in finding the zero, going down to hundredths, is

```

a ,: show f a=: 0.5+nn%100
0.5 0.51 0.52 0.53 0.54 0.55 0.56 0.57 0.58 0.59 0.6
0.08 0.06 0.05 0.03 0.02 0.01 _0.01 _0.02 _0.04 _0.05 _0.06

```

where the leading zero of the 0.5 of the expression being keyed in is required, not optional. The new display of the results shows that the zero is between argument values 0.55 and 0.56.

The result values are becoming rather too close to zero, and will now need to be scaled up, so in

```

a ,: show 1000*f a=: 0.55+nn%1000
0.55 0.551 0.552 0.553 0.554 0.555 0.556 0.557 0.558 0.559 0.56
5.38 3.97 2.56 1.16 _0.24 _1.64 _3.04 _4.44 _5.83 _7.23 _8.62

```

the results are multiplied by a thousand before being displayed, the * symbol being used to stand for the multiplication function because the × symbol is not provided in the ASCII character set! In the absence of parentheses, the right argument of a function is the entire expression to the right of the function symbol, so that the * in 1000* takes the result of the application of f as its right argument.

By now, the argument values are getting a bit too long to print out comfortably, and the way they correspond should be getting to be clear, so they won't be shown stacked any longer. The argument of f now being from 0.55 to 0.56, and the zero being located between 0.553 and 0.554, the next narrowing down can be done by

```

show 10000*f 0.553+nn%10000
11.6 10.2 8.79 7.39 5.99 4.58 3.18 1.78 0.38 _1.02 _2.42

```

which now places the zero between arguments 0.5538 and 0.5539.

The J interpreter makes it easy enough to bring down the previous line and simply insert the three extra digits needed at each stage of the calculation. But by now the strings of zero digits are becoming a bit hard to keep visual track of, so, using e-notation for the large scaling integer and the small interval size, the next step becomes

```

show 1e5*f 0.5538+nn%1e5
3.79 2.39 0.99 _0.42 _1.82 _3.22 _4.62 _6.02 _7.42 _8.83 _10.23

```

This process of narrowing down has now become somewhat repetitive. Of course, the whole process could be made automatic, but by doing it step by step the succession of numbers gives a better feel for how the function f is behaving. Nevertheless, the repeated part of the expression used just above could relatively easily be encapsulated (by an experienced J user) as in

```

g =: ]*f@(+nn&%)

```

so that the investigation can now proceed as follows.

```
show 0.55382 g 1e6
9.86 8.46 7.06 5.66 4.26 2.85 1.45 0.05 _1.35 _2.75 _4.15
show 0.553827 g 1e7
0.51 _0.89 _2.29 _3.69 _5.09 _6.49 _7.9 _9.3 _10.7 _12.1 _13.5
show 0.5538270 g 1e8
5.14 3.73 2.33 0.93 _0.47 _1.87 _3.27 _4.68 _6.08 _7.48 _8.88
show 0.55382703 g 1e9
9.31 7.91 6.51 5.11 3.71 2.31 0.9 _0.5 _1.9 _3.3 _4.7
```

For some time now the results have been showing a strange uniformity. It might be suspected either that the arithmetic is beginning to give out, or that the variation between successive values has become uniform. This can be checked by taking the result values in pairs and subtracting them. This is done (without explanation here) by

```
show 2-/\0.55382703 g 1e9
1.4 1.4 1.4 1.4 1.4 1.4 1.4 1.4 1.4 1.4
```

which immediately shows the postulated uniformity. Indeed, keying in the expression without the show gives ten values 1.40169 so it is very uniform. This could allow a fairly accurate interpolation by hand of just where the zero is, but, for the purposes of exploration, the estimation process is continued here as before, except that now

```
show =: ".&(0.1&":)
```

because only one decimal place is now needed to be shown.

```
show 0.553827036 g 1e10
9 7.6 6.2 4.8 3.4 2 0.6 _0.8 _2.2 _3.6 _5
show 0.5538270366 g 1e11
6.2 4.8 3.4 2 0.6 _0.8 _2.2 _3.6 _5 _6.4 _7.8
show 0.55382703664 g 1e12
6.3 4.9 3.5 2.1 0.7 _0.7 _2.1 _3.5 _4.9 _6.3 _7.7
show 0.553827036644 g 1e13
7.2 5.8 4.4 3 1.6 0.2 _1.2 _2.6 _4 _5.4 _6.8
show 0.5538270366445 g 1e14
1.9 0.5 _0.9 _2.3 _3.7 _5.1 _6.5 _7.9 _9.3 _10.7 _12.1
show 0.55382703664451 g 1e15
4.9 3.3 2 0.7 _0.7 _2 _3.3 _4.7 _6.2 _7.5 _8.9
show 0.553827036644513 g 1e16
6.7 4.4 6.7 2.2 0 0 0 _2.2 _6.7 _2.2 _6.7
```

The values have become erratic, so much so that doubt much attach to any attempt to add further digits to the value found for the first zero. However, checking with

```
f 0.55382703664451 0.553827036644513 0.5538270366445135
4.88498e_15 6.66134e_16 0
```

suggests that at least the first fifteen decimal places should be reliable, provided the calculation built into the interpreter is reliable to such precision.

It is important to realise that the barrier that has been encountered here is one of precision, not range, since

```
4*4e32 4e_32
1.6e33 1.6e_31
```

shows that a much greater range can be satisfactorily used.

FINDING THE LOWEST VALUE

While the value of the second zero can be as easily found as the first, between the two zeroes all the values of f are negative, it might be presumed. This can be explored by

```
show 0.5 g 5
0.4 _0.9 _1.8 _2.3 _2.1 _1.2 0.7 3.6 7.8 13.7 21.6
```

where the interval between arguments is 0.2 and starting from 0.5. This suggests a minimum value for f of about $_{2.3}\%5$ when the argument is about 1.1, probably just a bit larger than 1.1.

Before going further, it is convenient to redefine f as its negation, thus

```
f =: -&(^-o.)
```

so that the values being looked at will be free of the negative sign, though now what was lowest value will have become the highest. Consequently,

```
show 1.1 g 100
45.2 45.3 45.4 45.4 45.5 45.5 45.4 45.4 45.3 45.1 45
```

suggests that, while the now highest value of f is 0.455, the argument corresponding to this value is round about 1.14. But this is now estimation, and one decimal place for `show` is too few.

The next steps in calculation are therefore

```
show =: ".&(0.2&" :)
show 1.14 g 1000
454.65 454.66 454.67 454.68 454.68 454.68 454.68 454.67 ...
```

which verifies the highest value to be about 0.45468 but it's mainly a guess that the argument for that value is about 1.144. Even when we issue

```
show=: ".&(0.3&" :)
```

to get three decimal places, which then allows

```
show _45468+10*1.144 g 10000
0.151 0.172 0.191 0.206 0.218 0.226 0.232 0.234 0.234 0.23 0.223
```

where the `_45468+10*` reduces the size of numbers displayed and gets an extra decimal place at the same time. The procedure can be continued, thus

```
show _4546823+100*1.1447 g 100000
0.447 0.455 0.46 0.461 0.46 0.455 0.447 0.436 0.422 0.405 0.384
```

which adds an extra digit (a bit uncertainly—is it 2, or 3?) to the value of the argument, and two extra digits to the value of the function. The procedure goes on, here splitting the difference between the 2 and the 3.

```
show _454682346+1000*1.144725 g 1000000
0.102 0.116 0.126 0.134 0.138 0.139 0.137 0.132 0.124 0.113 0.098
```

which confirms 2, and its successor 9, and again adds a couple of digits, 13, maybe even 138, to the value of the function. Now

```
show _454682346138+100000*1.144729 g 1e7
0.132 0.395 0.626 0.825 0.994 1.131 1.236 1.311 1.353 1.364 1.344
```

shows that the last digit of the previous function value should have been 9 rather than 8, but still gives another two function value digits for the single argument digit. In the next step

```
show _45468234613935+1e6*1.1447298 g 1e8
0.273 0.547 0.766 0.984 1.117 1.297 1.344 1.391 1.43 1.477 1.477
```

leads immediately to

```
show _4546823461393647+1e7*1.14472989 g 1e9
10 5 _4 5 1 1 _4 1 1 _4
```

where everything now blows up at a point where the function value is known to about 16 digits, but the argument value to only 9.

This mismatch between the accuracy of the result and that of its argument is a good illustration of the inherent unreliability of simple computer arithmetic. More extreme examples of mismatch could easily be concocted.

CONCLUSION

The procedures described in this article show the way, or the style, in which calculations can be carried out interactively using a tool like \mathcal{J} . If the calculation tool is mastered, most calculations can be carried out in the same style. The barrier is attaining that mastery.

Indeed the repeated searching for an answer by a kind of trial and error is not characteristic of functional calculation, since mastery will enable functions to be defined which can do the searching automatically. Such virtuosity would not be appropriately demonstrated in an *apéritif*.

The interactive style of calculation depends on two factors illustrated in this article. The first is simply the ability to key in an expression and have it evaluated on the fly. The second is the ability to give a name to an expression then use that name from then on as though it were that expression.

This article is only an *apéritif* to the main meal of getting to grips with a ramified functional calculation tool.

References

1. Menninger, K. (1958) *Zahlwort und Ziffer*, Vandenhoeck & Ruprecht, Göttingen,

2nd edition (as *Number Words and Number Symbols* by MIT Press in 1969).

2. Thimbleby, H. W. (1995) A New Calculator and Why it is Necessary, *The Computer Journal*, **38**(6), 418–433.