Functional calculation

1: Numerical ingredients

Neville Holmes, University of Tasmania
Neville.Holmes@utas.edu.au

Introducion

This article is the second in a series (numbered origin-zero *Ed.*) 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 J. This article reviews the numerical calculation capabilities of J which are the basis for its functional calculation capabilities, and which must be understood before the functional calculation capabilities proper can be understood. The capabilities described in this article will be illustrated and explained in detail in the next article.

Calculation

Calculation is generally reckoned to be the systematic manipulation of numeric values. Our schools, and our pocket calculators, are burdened with the idea that there are four kinds of arithmetic, called addition, subtraction, multiplication and division. The teaching of these traditional four arithmetic functions has not been blessed with success recently, and one of the reasons why it has not might be because there are in fact many arithmetic functions – some simpler than the traditional four, some more complex.

There are also many kinds of numbers, but this is another richness ignored by schools and pocket calculators.

This article reviews both the arithmetic functions provided by the functional calculating language J (as introduced in the first article in this series), and the numbers which are recognised by J. Using these functions, and their numbers, is not in itself functional calculation, but is part of the basis for it.

Numbers

Calculation changes numbers by systematic alteration or by combination. The apparent result of a calculation depends on up to three aspects of the numbers involved:

- what any starting numbers were keyed in as,

- what any numbers were stored as within the calculator, and
- what the result was displayed as.

Of course, it also depends on the functions applied to the numbers.

Number representation

Ideally, a calculator will store all the numbers it has to deal with as well as is possible. Just how they are stored depends on the underlying electronics, which typically used a binary representation that is unsuitable for direct use by humans. Anyone using a calculator should be as unaffected as possible by the method used to store numbers, and any decisions about the kind of representation to be used should be determined by the value to be stored, not by the user.

Although J interpreters shield their users from details of how numbers are stored, they nevertheless have to take numbers in from the keyboard and put them out to the display screen. The only practical way of doing this is using the unfortunate standard ASCII character set, unfortunate because of its poverty, lacking as it does even the × and ÷ symbols! The following table sums up the provisions of the J notation for expressing individual numbers.

`0 1 … 8 9`	decimal digits	`7 364 529081`	
`.`	fraction point	`0.7 3.64 52908.1`	must be digit to left of point
`_`	negative sign	`_7 _364 _52908.1`	underbar, not hyphen
`e`	scale point	`7e0 _3.6e4 5290e81`	scaling basis is ten
`r`	ratio point	`1r7 3.6r4 5290r_81`	
`j`	cartesian point	`1j7 3.6j4 5e90j_81`	real `j` imaginary
`ad`	degree point	`1ad7 3.6ad4 5e90ad_81`	magnitude `ad/ar` angle
`ar`	radian point	`1ar7 3.6ar4 5e90ar_81`	a radian is about 57°
`p`	pi point	`1p2` (π^2) `2r3p_1` ($2/3\pi$)	
`x`	euler point	`1x2` (e^2) `2r3x_1` ($2/3e$)	
`b`	base point	`2b1111` (15) `16bff` (255)	
`_`	infinity	`_ (1r0) __ (_1r0)`	these two notations
`_.`	indeterminacy	`_. (_-_)`	are of special numbers

Furthermore, if an integer has an `x` suffix then it is stored exactly and exact arithmetic is used in conjunction with it if possible.

All the most commonly needed numbers can be succinctly expressed, as shown by the examples of the table, but several points need to be explained and emphasised.

- The elements in the list are applied in the sequence given. For example, the scale point is effective before the ratio point. So keying `1e2r3` in gives `33.3333`, while `1r2e3` gives `0.0005`. Also, keying `1.2e3` in by itself will give a display of `1200` but keying `1e2.3` in will display an error message *ill formed number* because the `e` sees `2.3` as its suffixing component but that component must be an integer.

- The elements not separated in the table by a horizontal rule are strict alternatives. So keying `1j1p1` in yields `1j3.14159`, but keying `1x1p1` in yields *ill formed number*.

- The symbol for the negative sign is not the same as the symbol for subtraction. Negativeness is a property, subtraction is a function, and always the twain should be distinct. An overbar is often used for the negative sign, but the ASCII character set lacks an overbar so the underbar must be used.

- The elements following the negative sign in the table are letters of the alphabet. They are, when immediately preceded and followed by numbers of lower form, arithmetic signs and not function symbols. They are, as is the fraction point, infixes, and in this the negative sign is an exception, being a prefix.

- Apart from the above, the letter `x` is used as a suffix to integers to signal that exact arithmetic is to be used on them if possible. This capability is very useful but was added late to the notation and is not yet fully and cleanly worked out.

- The last two lines of the table show some special uses of the underscore symbol, and the symbols given there are individual values, not elements.

- Only the elementary symbols given in the table may be used to compose an individual number. Parentheses and blanks are not allowed within individual numbers, though they may be used to separate individual numbers.

Any number may be keyed in in a variety of ways, but a number will, unless the user specifies a particular format, always be shown on the screen in the same way, however it may have been keyed in or calculated.

Some examples

For the most part, simple numbers like `123 4.5 6e7` and `8.9e10` will give unsurprising results, at least for people with the experience that tells them that the `e`

signifies that the number to its left is to be scaled by the power of ten specified by the number to its right. Indeed, these particular numbers will display exactly as they are keyed in.

Some simple numbers, like `1e2` and `3e4`, will display in a simpler form, in this case as `100` and `30000` – not as `30,000`! Should you key `30,000` in, you will get `30 0` back. Keying `1,234` in to yield `1 234` seems less strange, but even that is not what it might seem to be, because keying `1+1,234` gives `2 235`. The explanation for this is that the comma is a function symbol in J, here standing for a function usually called *catenate*, so `1,234` specifies that the `1` is to be catenated with the `234` to give the two numbers `1` and `234`. So `2+30,000` will give `32 2`, but notice (this will be explained later) that `30,000+2` will give `30 2`.

Numeric functions

The numeric functions known as *scalar functions* are applied to one or two numbers and produce a single number as a result. The *primitive functions* are those functions which have a simple symbol for a name. Some of the scalar functions provided by J as primitive functions are given in the following table.

`+`	conjugate	add	`+.`		GCD	`+:`	double	nor
`-`	negate	subtract	`-.`	not		`-:`	halve	
`*`	signum	times	`*.`		LCM	`*:`	square	nand
`%`	reciprocal	divide				`%:`	square root	root
`|`	magnitude	residue						
`^`	exp	power	`^.`	$\log_e$	logarithm			
`=`		equal	`=.`		(local is)	`=:`		(global is)
`<`		less than	`<.`	floor	lesser	`<:`	decrement	not more
`>`		more than	`>.`	ceiling	greater	`>:`	increment	not less
						`~:`		not equal
`!`	factorial	choices						
`?`	roll		`o.`	pi times	circular	`p:`	prime	
`[`	(same)	(left)						
`]`	(same)	(right)	`".`	(do)		`":`	(format)	(format)

The functions are given by name, a name which is meant to suggest what the function does. If there is any doubt about this, experiment with the interpreter can

be used to dispel the doubt. Again, some explanation is necessary to clarify the table.

- The dot (`.`) and colon (`:`) are used as character set extenders, and, when suffixed to a plain function symbol, effectively provide a new function symbol, though the symbols that differ only in their suffix are usually related in some way.

- A primitive function symbol may stand for two different functions, which is why there are two columns of names after the function symbol in the table. The name on the left is for a monadic function, that is, for a function which only has one argument, a right argument. The name on the right is for a dyadic function, that is, for a function which has two arguments, one on its left and one on its right.

- Some functions are restricted in the results they can produce. In particular, dyadic `= < > ~: <:` and `>:` can only produce a `0` or a `1`, though these results are ordinary numbers in the sense that there is no restriction on their use in further calculations.

- Some functions are restricted in the arguments they will take. For example, monadic `?` and `p:` take in only non-negative integers.

- The symbols `=.` and `=:` do not stand for functions at all. They are called *copulas* and are used for naming results. The name to the left of the copula is given to the value on the right, whatever that might happen to be. These symbols are sometimes called *gets* or *becomes*, and the global version should normally be used.

- Otherwise, the functions whose names are given in parentheses are not scalar functions, but are useful in connection with them. The functions that the brackets `[` and `]` stand for are very useful, though their result is always one of their arguments. The *format* function converts its right argument into a character string, optionally under control of its left argument, while the *do* function can convert the character string back to a number, though often not the same number.

- The circular functions have left arguments restricted to the values shown in the table below, and, where relevant, their right arguments are taken as radians. Actually, these functions include hyperbolic and pythagorean functions, as shown in the following table. Note that, in the following table, the left argument constant is separated from its function symbol by a blank, which is necessary to avoid the error message caused by an attempt to interpret the `o.` as part of a constant, the round character in the `o.` symbol being a lower case *o*. This is

not the case with the other function symbols introduced so far, except `p:`, because they use nonalphabetic characters.

						`0 o.`	sqrt(1-x^2)
`1 o.`	sine	`2 o.`	cosine	`3 o.`	tangent	`4 o.`	sqrt(1+x^2)
`5 o.`	sinh	`6 o.`	cosh	`7 o.`	tanh	`8 o.`	sqrt(_1-x^2)
`_1 o.`	arcsine	`_2 o.`	arccosine	`_3 o.`	arctangent	`_4 o.`	sqrt(_1+x^2)
`_5 o.`	arcsinh	`_6 o.`	arccosh	`_7 o.`	arctanh	`_8 o.`	sqrt(_1-x^2)

The hierarchy of functions

Clearly, J provides many primitive functions, as well as the ability to give a name to any, by the expression `sqrt=:%:` for instance. Dealing with these cleanly leads to a break with tradition that provokes a strong rejection from arithmetic traditionalists.

The primitive functions are traditionally considered to be arranged in a hierarchy of strength and direction, and the higher primitive functions are represented by notational peculiarities which imply a hierarchy. For J, the richness of primitive functions and the uniformity of expression forced by the ASCII character set and the linearity of its use, make a hierarchy practically impossible. Thus, `5+%|x` stands for *five plus the reciprocal of the magnitude of x*, while `a+7*b` stands for *a plus the product of seven and b*.

The absence of a hierarchy of functions leads to expressions having meanings which, though completely reasonable, are upsettingly untraditional. For example, `a*7+b` stands for *a times the sum of seven and b*, while `a-7-b` stands for *a minus the subtraction of b from seven*. If the traditional meanings are required for the arguments in that sequence, then parentheses must be used in J, as `(a-7)-b` and `(a*7)+b`. The gain is simplicity, the price is a break with tradition.

Summary

This article is like a list of ingredients that can be used for numerical calculation using the notation provided by J. At their simplest, these ingredients can be put together in the manner learned in elementary school, because the simplest of the expressions learned in elementary school, expressions like `1+2` and `7.2-5.75`, can be keyed in to the interpreter to give the result expected in elementary school.

However, the need to expand the number of elementary functions available, coupled to the restrictions of the ASCII character set, mean that some calculations and their results will not be quite like the results expected from elementary school. Never-

theless the changes are consistent and systematic, and their adoption allows elementary calculation to be extended in elementary ways, ways which allow simple use of an interpreter to evaluate expressions.

The next article in this series will explain and illustrate how numerical calculation is done with the J interpreter, as a preliminary to further treatment of functional calculation.

Footnote: This essay was written a decade ago to introduce J to classes of honours students as explained in the introductory essay "Tacit J and I". No attempt has been made to upgrade the text to incorporate subsequent changes to J, but I believe that the original design of J was exceptionally robust and will not have causes errors to crop up in the description above.