Coding 50g User RPL math functions for size and speed: stack math, algebraic expressions, and local variables

D.A. Burkett

17 April 2024

We have two options when writing userRPL code to evaluate formulas. The first option is to translate the formula to a sequence of userRPL commands and functions, including commands to move and copy operands on the stack. I will call this option 'stack math'. The second option is to define the formula as a ticked algebraic expression, such as 'a ∗ x + b', where a, x, and b are local variables. This method will be called 'expression math'.

Writing stack math programs to evaluate complicated functions is time-consuming. Using expression math can simplify and speed up program development, but what penalty is incurred in code size and execution time? Fast execution matters if the function is evaluated dozens or hundreds of times for plotting, numerical integration, or numerical solving, or if the program is applied to large lists. Code size is always relevant for calculators with limited memory.

We are interested in approximate numeric results so the mode settings used are Numeric, Approx, Complex, and Radians, with Number Format set to Std. For this comparison we will not optimize the formula or code for accuracy, overflow, underflow, or maximum input domain.

The following equation* is used as a test case.

$$f(x) = \frac{2}{b}\sqrt{(bx+c)\left(bx+c+\frac{b^2}{4}\right)} + \frac{b}{2}\,\text{arsinh}\left(\frac{2\sqrt{bx+c}}{b}\right) \tag{1}$$

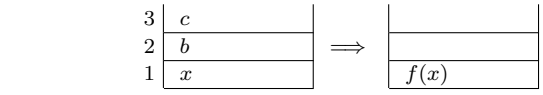
Let

$$t_1 = bx + c, \qquad t_2 = 2/b, \qquad t_3 = b/2$$

then equation (1) can be written as

$$f(x) = t_2\sqrt{t_1(t_1 + t_3^2)} + t_3\,\text{arsinh}(t_2\sqrt{t_1}) \tag{2}$$

We'll write a few programs using stack math and expression math to evaluate equation (1) and compare program size and execution time. All of the programs have the same stack I/O definition:



*The equation finds the arc length of a parabola segment, from section 11.14 of *An Atlas of Functions*, 2e, Keith Oldman et al. 'arsinh' is the inverse hyperbolic sine, the 50g function is ASINH.

The first program, F1 (shown in Table 1), uses stack math to evaluate $f(x)$ without local variables. The code is not particularly optimized but that's sort of the point: we want a fairly naive implementation to compare with the other methods. The capital letters A, B, … in the stack diagram indicate these intermediate results:

$$A = t_2\sqrt{t_1} \qquad B = \operatorname{arsinh}(A) \qquad C = t_3 B$$
$$D = t_1 + t_3^2 \qquad E = t_1 D \qquad F = t_2\sqrt{E}$$

Table 1: Program F1 code and stack diagram (checksum #E284h)

Instruction	1:	2:	3:	4:	5:	6:
«	x	b	c			
OVER	b	x	b	c		
*	bx	b	c			
ROT	c	bx	b			
+	t_1	b				
SWAP	b	t_1				
DUP	b	b	t_1			
2.	2	b	b	t_1		
/	t_3	b	t_1			
2.	2	t_3	b	t_1		
ROT	b	2	t_3	t_1		
/	t_2	t_3	t_1			
3.	3	t_2	t_3	t_1		
DUPN	t_2	t_3	t_1	t_2	t_3	t_1
ROT	t_1	t_2	t_3	t_2	t_3	t_1
√	$\sqrt{t_1}$	t_2	t_3	t_2	t_3	t_1
*	A	t_3	t_2	t_3	t_1	
ASINH	B	t_3	t_2	t_3	t_1	
*	C	t_2	t_3	t_1		
4.	4	C	t_2	t_3	t_1	
ROLLD	t_2	t_3	t_1	C		
SWAP	t_3	t_2	t_1	C		
SQ	t_3^2	t_2	t_1	C		
PICK3	t_1	t_3^2	t_2	t_1	C	
+	D	t_2	t_1	C		
ROT	t_1	D	t_2	C		
*	E	t_2	C			
√	$\sqrt{E}$	t_2	C			
*	F	C				
+	$f(x)$					
»						

Note that eleven of the thirty instructions just copy and move things on the stack, which seems excessive. But we'll press on.

The second program F2 uses expression math and is perhaps the fastest program to write, as long as you get all the parentheses in the right places. The algebraic expression is a (mostly) direct translation of equation (1). Note that EVAL must be used to evaluate the expression to a numeric result, and that local variables can be used within the expression.

Program F2, expression math (checksum #D5D1h)

```
« → c b x
«
'2.*√ ((b*x+c)*(b*x+c+SQ (b)/4.))/b+b*ASINH (2.*√ (b*x+c)/b)/2.'
EVAL
»»
```

The third program F3 also uses expression math, but pre-computes the term $t_1 = bx + c$ and saves it in local variable t to reduce the program size and execution time.

Program F3, expression math, pre-compute t_1 (checksum #BF63h)

```
« OVER * ROT + → b t
«
'2.*√ (t*(t+SQ (b)/4.))/b+b*ASINH(2.*√ t/b)/2.'
EVAL
»»
```

The fourth program F4 uses stack math, but pre-computes terms t_1, t_2, and t_3 and saves them as local variables p, q, and r, respectively.

Program F4, stack math with local variables (checksum #B6FC)

```
« OVER * ROT + SWAP 2. / DUP INV → p r q
«
p √ q * ASINH r * r SQ p + p * √ q * +
»»
```

Each program was tested with $b = 5$, $c = 3$, and $x = 2$, and all four programs return 9.23341584302. The table below shows the program size and the execution time returned by TEVAL. The reported times for F1 and F4 are the average of ten executions.

Program	Description	Bytes	Time (s)
F1	Stack math without local variables	89.0	0.0448
F2	Expression math	163.5	0.4360
F3	Expression math, pre-compute $t_1 = bx + c$	127.0	0.2726
F4	Stack math with local variables	116.5	0.0465

Some conclusions and observations:

- Implementing the function with stack math and without local variables results in the smallest, fastest program (F1).
- Expression math results in larger programs and dramatically increased execution time; compare 0.436 seconds for F2 to 0.0448 seconds for F1: using expression math takes almost ten times as long.
- Execution time and code size of expresssion math programs can be reduced by pre-computing subexpressions, if possible, as shown in program F3.
- Local variables incur a small execution time penalty in stack math programs. Program size is increased, but the resulting code is more straightforward and easier to understand without a stack diagram.
- Despite the longer execution time and larger code size, expression math programs are useful to test formula evaluation for correctness and accuracy before putting the time and effort into a stack math program. The expresson math version can then be used to test the stack math program for correctness.
- If a function is only to be evaluated occasionally from the keyboard, expression math is probably adequate.