HP-12C Tried & Tricky Trigonometrics

Valentin Albillo (Ex-PPC member #4747)

I *love* Voyagers. Wish they were still marketed, the most stylish, elegant calculators ever to be produced, well ahead of their time and sadly missing now. How wonderful it would be if you could buy a spare 11C/15C or two at your local shop, so you could carry one neatly in your pocket anytime, anywhere, without fear of losing it or wearing it out, knowing that an indefinite supply was available.

But regrettably, right now there are just a limited, small number of them still in working order, less so still mint, and their numbers are constantly *decreasing*. Say your dog just chewed your 15C. Or you toddler crashed your 11C against the floor, and broke the display. One less, not many more still to go. And when the last one dies, that's it. You'll only see them at some museum. Except for one of their kind.

Enter the HP-12C. Made in the millions. Unsinkable. De facto business standard. Even my boss does have one (or several?). I own three (or four?). But it also has its flaws. Maybe for a business user it's very near perfection, with its clever business function set. But what about us, technically-oriented users? If this is the only Voyager still available for long decades to come, it better suits our needs too, right?

Yes, yes, I know. You want scientific functions, you buy a 32SII or an HP-48/49 or an algebraic model. Now, I know you mean well but *no thanks*. An algebraic model is totally out of the question for many of us would never adapt to it nor won't we ever want to.

Also, an HP48/49 won't fit in my shirt pocket at all, too clumsy to carry easily. Their screen can't compare with the clarity and boldness of the 12C's. Their style can't match its golden elegance either. Their CPU will do things fast, far faster than I need for my daily calculations, and will eat batteries fast too. I've never ever changed *even once* my 12C's batteries, though that's nothing to write home about. After all, it's only 8 years since I bought it, brand new cells included. You won't expect the 12C to have depleted them after such a short time has elapsed. Ah, but you'll say: their power, their awesome power ... well, who needs it to do the usual simple arithmetic tasks, you know, totalling bills, computing taxes and the like? Not me. I want a good programmable RPN calculator, with the usual assortment of math functions thrown in for good measure, but in style.

Now, the 12C comes near to satisfying my math wishes (not needs, I never need to do anything but simple arithmetic for my real-life affairs). It has square root, exponential and logarithm and raising numbers to powers. It even has reciprocals, an exotic factorial function and, talk about luxury, linear regression. But trigonometrics, it has *none at all!* Nothing! Nada! Not even Pi!

Shock! Horror!! How are we going to make do with not being able to compute an occasional sine on the *only* Voyager available? What if our pesky math-oriented

neighbour suddenly ask us to please compute the 5 roots of this 5th-degree equation he was assigned on our oh-so-wonderful 12C? Gosh, that will require some sines and arc sines! What can we do? Perhaps retreat in shame and humbly recognize that our golden marvel isn't so marvellous after all? *Not a chance!*!

We have a purpose in life: to let that disgrace of a human being know that our 12C can do sines and cosines and arc sines (and whatever trigs he cares to throw at it) for breakfast. Let's enter "Serious Mode".

Our mighty goal: Trim & Trusty Trigonometrics for the 12C

Yes. Exactly that. To begin with, let's enumerate the ***5 required specifications*** that our own trigonometrics implementation on the 12C (or any other allegedly decent implementation) *must inexcusably meet* to be worth its salt. In strict order of relevance:

1) All six trigonometric functions in a single program. This is the first and foremost requirement: **all** six principal trigonometric functions, namely sine, cosine, tangent, arc sine, arc cosine, and arc tangent, **must** be conveniently computed using a **single** program (99 steps or less).

Else, we simply haven't achieved our goal. It's all very well to implement a sine function as a single program, then write another for the arc sine and so on. But that's neither practical nor useable. The 12C has no mag cards to quickly load the required program, and keying in the different programs while computing a complicated, mixed trig expression is out of the question. It's all six functions computable using a single program or else we should acknowledge defeat.

2) Full accuracy for all functions. No trade-offs between speed and accuracy will be tolerated. All functions will be computed to the *maximum* possible accuracy.

Else, if we allow reduced accuracy, we can never really trust the results we get, specially in a long chain of trig calculations. Would you trust your HP-xx when computing a sine or arc sine if the manual said the accuracy was "reduced" to save time or memory? Come on! We want 9-10 digits accuracy and we want them now!

We need to *trust* the results we get.

As for the input ranges, you must be able to compute an arc tangent for *any* input value, not being restricted to -1..+1. Real men compute their arc tangents from -Infinity to +Infinity, so our program must allow you to compute arc tangent of, say, 10^10 as easily as arc tangent of 0.1

3) Maximum speed and fast computation times for all functions and all inputs. Yes, we want all 6 functions, we want full accuracy, and we want it all *fast*. Real fast. We want *bounded*, maximum *guaranteed* times for any inputs within the allowed range for every function.

Else you will have the pitfall that many other trig implementations don't tell you at first: *slow convergence, if at all*. Some programmers simply use the usual Taylor Series Expansion for, say, the arc tangent or arc sine in a straightforward manner.

Too bad that when you try to compute arc sines or arc tangents of input values near 1, they will either take ages to find an answer or they'll settle for a much reduced accuracy. In some cases, they'll even fail completely, as when trying to use the straight arc tangent series to compute arc tangent of, say, 2.

On the other hand, our implementation won't choke at those extreme values or any other values for that matter. It will deliver its answers with full accuracy and in times comparable to any non-extreme input values. The computation time for arc sine and arc tangent will remain *bounded*, a few seconds for all arguments within the supported range.

4) Convenience and ease of use. Once the previous sine-qua-non requirements have been met, we can focus on giving the user extra convenience and ease of use.

For instance, it would be nice if the program would also include some provision to return *the value of Pi*, so that it can be used for angular mode conversions or other purposes. Of course, Pi or Pi/2 can be computed as the result of some trigonometric functions, but why should the user know or remember the required formulae? It's far more convenient if Pi or Pi/2 can be returned *without any input from the user at all*.

Also, it would be nice if the three direct trigonometric functions (sine, cosine, tangent) were computed *at once*. That way, the user would not need to remember the initial address (step number) for them, as all three would be computed simultaneously at negligible cost in computing time.

Finally, since preserving the stack between computations is not possible, it would be very convenient to let the user have *as many available storage registers as possible*, preferably 0-4, as the 12C can do storage arithmetic only on those registers. This means our program must use as few storage registers as feasible.

5) Compatibility. It would be desirable if our neat trigonometrics solution were also useable in *other trigonometrically-challenged HP calculators, such as **the HP-16C***, with as little modification as possible. With that goal in mind, we should refrain from using 12C-specific functions or registers. That's why we use a few numbered storage registers in our implementation instead of using the *financial registers* specific to the 12C, but *nonexistent* in the 16C. And that's why we don't use *storage arithmetic* at all even though it would be very convenient to do so, as it is *unavailable* on the 16C.

With that proviso, porting this implementation to the 16C is straightforward: just insert appropriate labels in the required jump points and change all GTOs to refer to those labels instead of the original step numbers. Come to that, you can actually optimize the program a lot using advanced 16C features that the 12C lacks, such as flags, subroutines, and additional tests. This will allow you to even add *extra functionality*, such as degrees/radians conversions for instance. The sky's the limit!

Our fearsome foes

Well, in no particular order: only 99 steps, no labels, no flags, no subroutine capability, no indirect addressing, no increment/decrement branching instructions for the loops, only two boolean tests and in particular no X=Y? or X#Y? tests, no x-squared function, no sign function, no absolute value function, etc. All of them combine to make our task that much more difficult. But succeed, we will.

Program listing

- SQRT is the square root function
- X<>Y is the "X exchange Y" stack operation

```
01 STO 5      26  *         51 STO 5      76  +
02  1         27  -         52 g SQRT     77  -
03 STO 6      28 g SQRT     53 X<>Y       78 g X=0?
04 RCL 5      29 g GTO 00   54  1         79 g GTO 82
05 RCL 5      30 ENTER      55  -         80 g LSTX
06 CHS        31 ENTER      56 g X=0?     81 g GTO 66
07 STO 5      32  *         57 g GTO 59   82 RCL 4
08 RCL 6      33  1         58 g GTO 39   83 g X<=Y?
09  2         34  +         59 g n!       84 g GTO 89
10  +         35 g SQRT     60 STO 6      85 g LSTX
11 STO 6      36  /         61 RCL 5      86  8
12 Y^X        37 STO 4      62  -         87  *
13 g LSTX     38  3         63 g SQRT     88 g GTO 00
14 g n!       39 X<>Y       64  /         89 g LST X
15  /         40 ENTER      65 STO 5      90 CHS
16  +         41  *         66 RCL 5      91 g GTO 86
17  -         42 CHS        67 CHS        92 CLX
18 g X=0?     43  1         68 STO 5      93 ENTER
19 g GTO 22   44  +         69 RCL 6      94  *
20 g LSTX     45 g SQRT     70  2         95 CHS
21 g GTO 05   46 CHS        71  +         96  1
22 g LSTX     47  1         72 STO 6      97  +
23  1         48  +         73 Y^X        98 g SQRT
24 g LSTX     49  2         74 g LSTX     99 g GTO 37
25 g LSTX     50  /         75  /
```

Implementation details

- steps 01-22 compute Sin(x) by using its Taylor Series Expansion:

  $$\text{Sin}(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$$

  the stopping criterium being that the contribution of a new term to the running sum *does not change its value at all*. This ensures maximum accuracy and fastest running time because only as many terms as needed for full accuracy are used. This approach allows also a wide valid input range from –5*Pi to +5*Pi.

- steps 23-28 compute Cos(x) by using the formula: Cos(x) = Sqrt(1-Sin(x)^2)

  As the square root is always taken in its positive value, this restricts the input range for Cos(x) to the usual range from –Pi/2 to +Pi/2. If you need to compute a cosine for values outside that range, either perform previously a range reduction or else compute it using the formula:

  $$Cos(x) = Sin(x + Pi/2)$$

  The reason for not using this formula instead of the one used in the program is that this requires having Pi/2 available, which means either recomputing it each time, doubling processing time, or else requires some initialization and using a dedicated register to hold it.

- steps 30-36 compute ArcTan(x) by using the formula:

  $$ArcTan(x) = ArcSin(x/Sqrt(1+x^2))$$

  this ensures that the full range -Infinity to +Infinity can be used for the argument.

- steps 37-91 compute ArcSin(x). To guarantee fast convergence for all x, a suitable scaling of the input value is previously performed, followed by a change of variable, and finally a Taylor Series Expansion is used to generate an intermediate result, which is then scaled back to give the final result.
- steps 37, 82-84 and 89-91 make sure the result has the correct sign in all cases.
- steps 92 computes Pi/2 using the formula: Pi/2 = ArcCos(0)
- steps 93-99 compute ArcCos(x) using the formula

  $$ArcCos(x) = ArcSin(Sqrt(1-x^2))$$

As the square root is always taken in its positive value, this restricts the input range to values of x from 0 to 1, both included. Should you need to compute the arc cosine of a negative x value, you can use the formula:

$$ArcCos(x) = Pi/2 – ArcSin(x)$$

As before, this formula was not used because it does require having Pi/2 available, which means either recomputing it each time, doubling processing time, or else it requires initialization and using a dedicated register to hold it and there's always the risk of the value being lost if its dedicated register is cleared or overwritten inadvertently.

You can test that the program is loaded correctly, and its accuracy by checking these results, shown as they are displayed in FIX 9 (f 9 in the 12C):

x	*Sin(x)*	*Cos(x)*	*Tan(x)*	*Time*
0.1	0.099833417	0.995004165	0.100334672	6 sec.
0.5	0.479425539	0.877582562	0.546302490	7 sec.
1	0.841470985	0.540302306	1.557407724	9 sec.
Pi/2	1.000000000	0.000000000	would div by 0	10 sec.
2	0.909297427	0.416146836	2.185039869	12 sec.
Pi	-7.098535e-12	1.000000000	-7.098535e-12	19 sec.

x	*ArcSin(x)*	*ArcCos(x)*	*ArcTan(x)*	*Time*
0.1	0.100167425	1.470628906	0.099668661	10 sec.
0.5	0.523598775	1.047197552	0.463647607	12 sec.
1	1.570796327	0.000000000	0.785398163	15 sec.
10	-	-	1.471127675	15 sec.
100	-	-	1.560796637	18 sec.
1000	-	-	1.569796326	18 sec.
1E10	-	-	1.570796327	18 sec.

Usage instructions

As the 12C doesn't have user labels, these are the entry points to compute the functions, with the argument X assumed to be in the display (X-register):

Function	*To compute, press:*	*Input range*
Sin(x)	GTO 00, R/S, X<>Y	#NAME?
Cos(x)	GTO 00, R/S	-Pi/2 to +Pi/2
Tan(x)	GTO 00, R/S, /	-Pi/2 to +Pi/2
ArcSin(x)	GTO 37, R/S	-1 to +1
ArcCos(x)	GTO 93, R/S	0 to +1
ArcTan(x)	GTO 30, R/S	-9.99E49 to +9.99E49
the constant Pi/2	GTO 92, R/S	no input required

Notes

- all angles are assumed to be in radians.
- no stack registers are preserved nor is X stored in LSTX, but R0-R3 are available at all times to store intermediate results or constants. The financial registers can be used as well.
- for Sin(x), Cos(x), and Tan(x) you can use f PRGM or GTO 01 instead of GTO 00. After any function is computed, the program pointer is left again at step 00, so you can compute a series of sines, cosines and tangents by simply pressing R/S.
- Sin(x) and Cos(x) are computed simultaneously. Sin(x) is left in the Y-register, and Cos(x) is left in the X-register (display), so you can obtain Tan(x) by simply pressing the [/] (division) key. If Cos(x) equals 0, you can't perform the division but must assume instead that Tan(x) becomes infinite
- the input range for Sin(x) goes from -5*Pi to +5*Pi, but large values of x (say > 2*Pi) result in reduced accuracy and increased running times, so you'd do well restricting your arguments to the range from -2*Pi to +2*Pi and reduce any larger ones to that range (by taking the remainder modulus 2*Pi).

Example 1

Find all 5 real roots of the quintic: 16 x^5 - 180 x^3 + 405 x -136 = 0

Though 5th-degree equations (a.k.a. quintics) *are not in general soluble* by algebraic means, after some fruitful and enjoyable labour of mathematical creation I came to this neat expression for its 5 real roots:

$$x = 3 * \text{Sin}[\ 1/5\ * (\text{ArcSin}(136/243) + 0*\text{Pi}, \pm 2*\text{Pi}, \pm 4*\text{Pi})\ ]$$

Note that we'll need to compute the sine of arguments exceeding Pi/2, but this is no problem for our program, so let's compute all roots on our 12C easily by following these steps:

1. As we will need Pi frequently, first we'll store a copy of it in R0:

   GTO 92 (Pi/2), R/S: 1.570796327 (this is the value of Pi/2)
   *2, *, STO 0: 3.141592654* (Pi stored in R0)

2. Now we'll compute u = ArcSin(136/243) and, as we'll need it 5 times, we'll store the result in R1:

   136, ENTER, 243, / : 0.559670782
   GTO 37 (ArcSin), R/S, STO 1 : 0.593988482 (u)

3. We have everything we need, now we'll compute all the roots. The 1st one is x1 = 3 * Sin(u / 5):

```
RCL 1 (u), 5, /, fPRGM, R/S : 0.992951849 (Cos u)
                      X<>Y : 0.118518464 (Sin u)
                      3, * : 0.355555391 (x1)
```

4. Encouraged by the success, we go now after the second root, which is: x2 = 3 * Sin[(u + 2*Pi) / 5]:

```
RCL 1 (u), RCL 0 (Pi),2,*,+,5,/ : 1.375434758
                    fPRGM, R/S : 0.194121239 (Cos)
                          X<>Y : 0.980977545 (Sin)
                          3, * : 2.942932636 (x2)
```

5. The remaining roots are computed likewise, and their values are:

```
x3 = 3*Sin[(u - 2*Pi)/5] = -2.723187320
x4 = 3*Sin[(u + 4*Pi)/5] =  1.463277005
x5 = 3*Sin[(u - 4*Pi)/5] = -2.038577707
```

6. Finally, we test the correctness of the roots and their accuracy by computing their sum and their product. The results for our computed roots are:

Sum of the computed roots = 0.000000005 (should be 0)
Product of the roots = 8.499999962 (should be 8.5)

So you see, after a long chain of trigonometric calculations (21 in all) and for quite sizable arguments (>2.63), we've got excellent accuracy throughout (9-10 correct digits for the roots).

Example 2

Check the equality: Tan 50º + Tan 60 º + Tan 70 º = Tan 50º * Tan 60º * Tan 70º

i.e: the *sum* of their tangents is the same as the *product* of their tangents. As the angles are expressed in degrees, we'll have to convert them to radians on the fly. Let's follow these steps:

1. We'll need the conversion factor Pi/180 = (Pi/2)/90, which we'll store in R0:

```
GTO 92 (Pi/2), R/S : 1.570796327 (Pi/2)
      90, /, STO 0 : 0.017453293 (Pi/180)
```

2. Now, we'll compute Tan(50º), which we'll store in R1:

```
50, RCL 0, * :  0.872664626 (50º in radians)
  fPRGM, R/S :  0.642787610 (Cos 50º)
   / , STO 1 :  1.191753592 (Tan 50º)
```

3. Same for Tan(60º), stored in R2:

```
60, RCL 0, * :  1.047197551 (60º in radians)
  fPRGM, R/S :  0.500000000 (Cos 60º)
   / , STO 2 :  1.732050807 (Tan 60º)
```

4. Finally, same for Tan(70º), stored in R3:

```
70, RCL 0, * :  1.221730476 (70º in radians)
  fPRGM, R/S :  0.342020144 (Cos 70º)
   / , STO 3 :  2.747477412 (Tan 70º)
```

5. Now, for the check:

```
RCL 1, RCL 2, +, RCL 3, + :  5.671281811
RCL 1, RCL 2, *, RCL 3, * :  5.671281800
```

and we see that the results agree to 1 unit in the 8th position, so we may say confident that the equality holds.

[Valentin wrote this article in time for the HP-12C anniversary issue (V20N5) but there was insufficient space to include it and I preferred to publish his excellent HP-11C article instead, since it was its anniversary as well. Needless to say, he was mortified to discover, upon reading the issue, that Wlodek planned to write a follow-up article on providing scientific functions, including trig, for the HP-12C.

In a similar vein, Wlodek wrote his article in time for the V20N6 which was the Newton-themed, November – December 2000 issue, hence the reference to the theme in the article and, once again, I was unable to include it due to pressure of space.

I therefore wish to take pains to point out that neither Wlodek nor Valentin have seen each others' articles. Any similarities are a consequence of chance or a result of the constraints imposed by the limitations of the HP-12C.

Thus, while there may be some overlap in material, I trust that your enjoyment of either article – Wlodek's follows – will not be spoilt. Ed.]