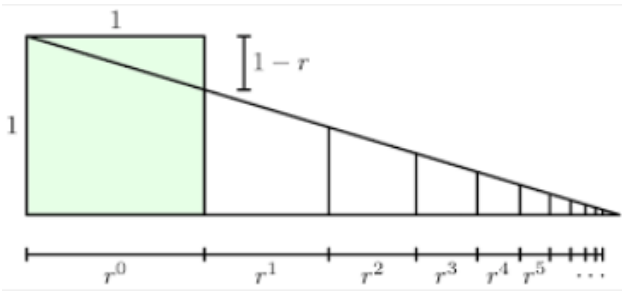
HP-41 Module: Areas, Series & Sums.



Overview

This module includes a selection of functions and FOCAL routines mainly focused on Series and Sums field and other related subjects. For the most part the routines are taken from Jean-Marc Baillard extensive web site, although some others are taken from Poul Kaarup's collection as well. A few are already available in the SandMath Module –even if this version is a more portable implementation that suits itself better for Clonix or NoVRAM owners.

The initial section of the module covers the simple sums of integers and integer powers. This is followed by simple explicit sums for single, double, triple and multiple series; recursive term sums and Euler transformations. Examples are also provided in the FAT for quick familiarization. The second section includes a set of MCODE functions and FOCAL routines dealing with area calculations for several geometric figures, such as circles and triangles, as well as areas and diagonal lengths of cyclic and regular polygons.

Without further ado, see below the list of functions included in the module:

XROM	Function	Description	Input	Author
18,00	**-SERIES 1A**	Section header	n/a	*Ángel Martin*
18,01	**CHSYX**	Sign Change of Y by X: Y*(-1)^X	Value in Y, n in X	*Ángel Martin*
18,02	**NCK**	Combinations of N in sets of K	n in Y, k in X	*Ángel Martin*
18,03	**NPK**	Permutations of N in sets of K	n in Y, k in X	*Ángel Martin*
18,04	**Σ0**	Sum of mantissa digits	argument, X	*Poul Kaarup*
18,05	**Σ1**	Sum of first N integers	argument, X	*Poul Kaarup*
18,06	**Σ1/N**	Harmonic Numbers	argument, X	*Ángel Martin*
18,07	**Σ2**	Sum of Squares of Numbers	argument, X	*Poul Kaarup*
18,08	**Σ3**	Sum of Cubes of Numbers	argument, X	*Poul Kaarup*
18,09	**ΣN^X**	Generalized Faulhaber's Sum	N in Y, exponent In X	*Ángel Martin*
18,10	**ΣUM**	Single Series Sum (Explicit)	arguments in X, Y, ALPHA	*Ángel Martin*
18,11	**ΣΣUM**	Double Series Sum (Explicit)	argument in X, ALPHA	*Ángel Martin*
18,12	**ΣΣΣUM**	Triple Series Sum (Explicit)	arguments in Y,X, ALPHA	*Ángel Martin*
18,13	**"ΣUME**	Euler Transformation	argument in X, Y, Z, ALPHA	*JM Baillard*
18,14	**"ΣUMR"**	Single Series Sum (Iterative)	arguments in X, Y, ALPHA	*JM Baillard*
18,15	**"NΣUM0"**	Multiple Series Sum (Explicit)	bbb.eee in X, data in registers	*JM Baillard*
18,16	**FNRM**	Finite Nested Radical order m	m in Y, n in X	*Ángel Martin*
18,17	**INRM**	Infinite Nested Radical order m	m in Y, n0 in X	*Ángel Martin*
18,18	**PRODX**	Infinite Product w. argument	X in X, n0 in Y	*Ángel Martin*
18,19	**XQRTN**	XQ Return to MCODE	Auxiliary function	*Martin-McClure*
18,20	**X#YR?**	Compares rounded X and Y	YES/NO Skips if false	*Ángel Martin*
18,21	**"NS"**	Example for NΣUM0	n/a	*Martin-Baillard*
18,22	**"S"**	Example for ΣUM	n/a	*Martin-Baillard*
18.23	**"SE"**	Example for ΣUME	n/a	*Martin-Baillard*
18.24	**"SR"**	Example for ΣUMR	n/a	*Martin-Baillard*

18.25	**"SS"**	Example for ΣΣUM	n/a	*Martin-Baillard*
18.26	**"SSS"**	Example for ΣΣΣUM	n/a	*Martin-Baillard*
18.27	**"SIG1**	Single Series Sum (Explicit)	n0 in X, name in ALPHA	*JM Baillard*
18.28	**"SIG2**	Double Series Sum (Explicit)	(n0, k0) in X,Y, name in ALPHA	*JM Baillard*
18.29	**"SIG3**	Triple Series Sum (Explicit)	(n0, k0, m0) in X,Y,Z; ALPHA	*JM Baillard*
18.30	**STK>PT**	Stack to Pointers	Decimal Data in Stack	*Ángel Martin*
18.31	**PT>STK**	Pointers to Stack	Binary Pointers in R00	*Ángel Martin*
18.32	**-AREAS_1B**	Section Header	n/a	*Ángel Martin*
18.33	**AINT**	ALPHA Integer Part	X in X	*Fritz Ferwerda*
18.34	**BRHM**	Brhamagupta formula	4 Sides in stack	*Ángel Martin*
18.35	**CIRCLE**	Circle through three points	Coordinates in R01-R06	*Ángel Martin*
18.36	**DIAG**	Diagonal formula	Used in CPLD	*JM Baillard*
18.37	**HERON**	Heron formula	Triangle sides in Z, Y, Z	*Ángel Martin*
18.38	**RPG1**	Regular Polygon from sides	# sides in Y, length in X	*Poul Kaarup*
18.39	**RPG2**	Regular polygon from circle	# sides in Y, radius in X	*Poul Kaarup*
18.40	**"3PNTS"**	Driver for CIRCLE/Areas	Prompts for coordinates	*Ángel Martin*
18.41	**"CCPA"**	Complex Cyclic Polygon Area	Parameters in stack and R00	*JM Baillard*
18.42	**"CCPA+"**	Driver for CCPA	Uses Newton Method	*Ángel Martin*
18.43	**"CPLA"**	Complex Cyclic Polygon Area	With all Sides known	*JM Baillard*
18.44	**"CPLA+"**	Driver for CPLA	Uses SOLVE	*Ángel Martin*
18.45	**"CPLD"**	Complex Cyclic Polyg. Diagonals	bbb.eee in X, data in Registers	*JM Baillard*
18.46	**"CPLD+**	Driver for CPLD	Prompts for data entry	*Ángel Martin*
18.47	**"PGA**	Polygon Areas	w/ Point Coordinates	*Poul Kaarup*
18.48	**"STLA"**	Star Polygon Area	Parameters in Stack	*JM Baillard*
18.49	**"STLA+"**	Driver for STLA	Prompts for data entry	*Ángel Martin*
18.50	**"#"**	Auxiliary for SOLVE	Under program control	*Ángel Martin*
18.51	**"TRIA**	Driver for ABC	Prompts for values	*Ángel Martin*
18.51	**"ABC**	Triangle Solver	Three knowns in stack	*JM Baillard*
18.53	**"TRIH**	Driver for HABC	Prompts for values	*Ángel Martin*
18.54	**"HABC**	Hyperbolic Triangles	Three knowns in stack	*JM Baillard*
18.55	**"TRIS**	Driver for SABC	Prompts for values	*Ángel Martin*
18.56	**"SABC**	Spherical Triangles	Three knowns in stack	*JM Baillard*
18.57	**"OUT**	Output Registers	Bbb.eee in X	*Ángel Martin*
18.58	**SINH**	Hyperbolic Sine	argument, X	*Ángel Martin*
18.59	**COSH**	Hyperbolic Cosine	argument, X	*Ángel Martin*
18.60	**TANH**	Hyperbolic Tangent	argument, X	*Ángel Martin*
18.61	**ASINH**	Hyperbolic ASIN	argument, X	*Ángel Martin*
18.62	**ACOSH**	Hyperbolic ACOS	argument, X	*Ángel Martin*
18.63	**ATANH**	Hyperbolic ATAN	argument, X	*Ángel Martin*

1 –Sums and Series

The first section includes several functions to calculate sums of integer powers, as well as simple methods to sum series given their general term in explicit or recurrent form.

- **Σ0** Is a small divertimento useful in pseudo-random numbers generation. It simply returns the sum of the mantissa digits of the argument – at light-blazing speed using just a few MCODE instructions. More about random numbers will be covered in the Probability/Stats section later on.

  Example: calculate the sum of all digits of the HP-41's rendition of pi:

  PI, XEQ "**Σ0**" => 40.000000000

- **Σ1/N** Calculates the Harmonic number of the argument in X, that is the sum of the reciprocals of the natural numbers (which excludes zero) lower and equal to n. It will be used in the calculation of the Kelvin functions and the Bessel functions of the second kind, K(n,x) and Y(n,x).

$$H_n = \sum_{k=1}^{n} \frac{1}{k}.$$

  Example: calculate H(5) and H(25).

  5, XEQ "Σ1/N" => 2.283333333
  25, XEQ "Σ1/N" => 3.815958178

- **Σ1, Σ2, Σ3** Are implemented to calculate the sum of integer powers directly based on the corresponding formulas. The number of terms to sum is expected to be in the X- register. Functions calculate the linear sum using the triangular formula; the sum of squares using the pyramidal formulas; and the sum of cubes also using the pyramidal formulas.

$$T_n = \sum_{k=1}^{n} k = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2} = \binom{n+1}{2}$$

$$P_n = \sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} = \frac{2n^3 + 3n^2 + n}{6}.$$

  Example: Calculate the sum of the first 10 natural numbers and their squares and cubes:

  10, **Σ1** quickly returns: 55.00000000
  LASTX, **Σ2**" => 385.0000000
  LASTX, **Σ3** => 3,025.000000

- **ΣN^X** Calculates a generalized value of the Faulhaber's formula for integer values of x. – The few first integer values of x have explicit formulas for the result (which are used in the functions described above), but that's not the case for a general value - which can also be non-integer. Obviously for x=-1 this function returns identical results than **Σ1/N**, albeit slower due to the additional complexity of the definition of the term.

Example: Check the triangular (x=1) and pyramidal (x=2) formulas for n=10 – which are particular cases of the Faulhaber's Formula, involving Binomial coefficients and Bernoulli's numbers. See the link below for details: http://en.wikipedia.org/wiki/Faulhaber%27s_formula

10, ENTER^, 1, XEQ "ΣN^X" => 55.00000000
10, ENTER^, 2, XEQ "ΣN^X" => 385.0000000

And using the convention B(1) = 0.5 the formula is:

$$S_m(n) = \frac{1}{m+1}\sum_{k=0}^{m}\binom{m+1}{k}B_k\, n^{m+1-k},$$

Which could be programmed using a few of the SandMath functions, albeit it would be considerably slower due to the impact of the Zeta algorithms (part of Bernoulli's) – kicking in for n>4.

- **CHSYX** Is related to the same subject, and in general relevant to the summation of alternating series **–** It can be regarded as an extension of **CHS** but dependent of the number in X. Its expression is:CHS(y,x)= y*(-1)^x, and thus changing the sign of Y when the number in X is odd.

- **NPK** calculates Permutations, defined as the number of possible different arrangements of N different items taken in quantities of K items at a time. No item occurs more than once in an arrangement, and different orders of the same R items in an arrangement are counted separately. The formula is in the left side below.

- **NCK** calculates Combinations, defined as the number of possible sets or N different items taken in quantities or K items at a time. No item occurs more than once in a set, and different orders of the same R items is a set are not counted separately. The formula is in the right side below.

$$\frac{n!}{(n-k)!}. \qquad \frac{n!}{k!(n-k)!}$$

The general operation includes the following enhanced features:

- Gets the integer part of the input values, forcing them to be positive.
- Checks that neither one is Zero, and that n>r
- Uses the minimum of {r, (n-r)} to expedite the calculation time
- Checks the Out of Range condition at every multiplication, so if it occurs its determined as soon as possible
- The chain of multiplication proceeds right-to-left, with the largest quotients first.
- The algorithm works within the numeric range of the 41. Example: nCk(335,167) is calculated without problems.
- It doesn't perform any rounding on the results. Partial divisions are done to calculate **NCK**, as opposed to calculating first **NPK** and dividing it by r!

Provision is made for those cases where n=0 and k=0, returning zero and one as respective results. This avoids DATA ERROR situations in running programs, and is consistent with the functions definitions for those singularities.

Note as well that there is no final rounding made to the result. This was the subject of heated debates in the HP Museum forum, with some good arguments for a final rounding to ensure that the result is an integer. This implementation however does not perform such final "conditioning", as the algorithm used seems to always return an integer already. Pls. Report examples of non-conformance if you run into them.

Square, Triangular and Tetrahedral Numbers.

These functions can be used to calculate a few well-known numbers, either directly or indirectly but as very quick examples. For example, the triangular numbers are the direct result of function **Σ1**, whist the square numbers are the direct result of functions**Σ2** and **Σ3** :

$$1+2+3+\cdots+n=\frac{n(n+1)}{2}=\frac{n^2+n}{2}$$ (the triangular numbers)

$$1^2+2^2+3^2+\cdots+n^2=\frac{n(n+1)(2n+1)}{6}=\frac{2n^3+3n^2+n}{6}$$ (the square pyramidal numbers)

$$1^3+2^3+3^3+\cdots+n^3=\left[\frac{n(n+1)}{2}\right]^2=\frac{n^4+2n^3+n^2}{4}$$ (the squared triangular numbers)

The sum of the all triangular numbers up to the n-th triangular number is the n-th tetrahedral number. These have the following expressions as direct value or as binomial coefficients:

$$T_n=\frac{n(n+1)(n+2)}{6}$$, and: $$T_n=\binom{n+2}{3}$$

Furthermore, the tetrahedral numbers can be derived from **Σ2** using the relation:

T(n) = Σ2(n). (n+2)/(2n+1)

Finally, you can also take advantage of the combination function **NCK** using the binomial coefficient formulas for the triangular or tetrahedral numbers. In fact there are many more kinds of "figurate" numbers, as can be seen in the list below:

https://en.wikipedia.org/wiki/Figurate_number

- $P_1(n)=\frac{n}{1}=\binom{n+0}{1}$ (linear numbers),
- $P_2(n)=\frac{n(n+1)}{2}=\binom{n+1}{2}$ (triangular numbers),
- $P_3(n)=\frac{n(n+1)(n+2)}{6}=\binom{n+2}{3}$ (tetrahedral numbers),
- $P_4(n)=\frac{n(n+1)(n+2)(n+3)}{24}=\binom{n+3}{4}$ (pentachoric numbers, pentatopic numbers,

Sums of Series using their General Terms

The following programs allow you to obtain the value of simple, double and triple series for which the general term is known, either as explicit expression or as a recurrent form.

Unlike other functions that utilize FOCAL programs as arguments (like SOLVE and INTEG in the Advantage Pac), these summing functions always use 9 decimal digits to determine the accuracy of the results – irrespective of the FIX settings on the calculator. Note however that this is not the case for the infinite product function, as described later on.

Simple Series.

ΣUM is a MCODE function that calculates a single series sum or function of an argument x, that is:

$$S(x) = (u_k + u_{k+1} + u_{k+2} + u_{k+3} + \ldots\ldots\ldots) = \Sigma\, u_n(x) \quad \text{for } n >= k$$

defined by the general term $u_n(x) = f(n, x)$ where f is a known (i,e, explicit) function.

A program which computes Un = f(n, x) is required as a subroutine. It is done assuming x is in X-register and n in the Y-register upon entry.If no argument x is required you need to enter any value in X, and ensure that the routine drops the stack before getting to work on the index n.

For example, the module includes the routine "**S**" as example for this case, with Un=1/n^n

ALPHA, "S", ALPHA, 1, ENTER^, XEQ "ΣUM" -> X = 1.291285997 (in 5 seconds)

If executed manually, these functions will prompt for the function name as an ALPHA string. When running in a program the function name is expected to be in the ALPHA registers.

As another example let's calculate the exponential function, given by the power series expression:

$$\exp(z) = \sum_{k=0}^{\infty} \frac{z^k}{k!} = 1 + z + \frac{z^2}{2} + \frac{z^3}{6} + \frac{z^4}{24} + \cdots$$

$$\sin x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} x^{2n+1} \qquad \cos x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$

We write the short routine shown below for the general term, and call the **ΣUM** function with zero as initial index in Y and the argument in X. The table summarizes a few results for your convenience. Note that the execution time will depend on the argument's value, and that because the code uses 13-digit sum routines the accuracy should be of 10 decimal digits (full range) with a FIX 9 setting.

ALPHA, "EXP", ALPHA, 0, ENTER^, x, XEQ "ΣUM"

Argument (x)	ΣUM (0, x)	exp(x)	error
0.5	1.648721271	1.648721271	0
1	2.718281828	2.718281828	0
5	148.4131591	148.4131591	0
10	22,026.46579	22,026.46579	0

```
01 LBL "EXP"
02 X<>Y
03 Y^X
04 LASTX
05 FACT
06 /
07 RTN
08 LBL "SINX"
09 SF 03
10 GTO 00
11 LBL "COSX"
12 CF 03
13 LBL 00
14 RCL Y
15 ST+ X
16 FS? 03
17 ISG X
18 NOP
19 Y^X
20 LASTX
21 FACT
22 /
23 X<>y
24 CHSYX
25 END
```

Here's another example to calculate the Erdos-Borwein constant, using two different approaches: one as a single sum, and another as a double sum (obviously less efficient but interesting for example's sake).

```
01 LBL "ERD2"
02 RDN
03 *
04 2
05 X<>Y
06 Y^X
07 1/X
08 RTN
09 LBL "ERD1"
10 RDN
11 2
12 LN
13 *
14 E^X-1
15 1/X
16 END
```

To calculate just type either one of the lines below

a) 1, ENTER^, **ΣUM** _ "ERD1" ->1.606695151
b) 1, ENTER^, ENTER^, **ΣΣUM**_ "ERD2" ->1.606695150

$$E = \sum_{n=1}^{\infty} \frac{1}{2^n - 1} \approx 1.606695152415291763\ldots$$

$$E = \sum_{m=1}^{\infty} \sum_{n=1}^{\infty} \frac{1}{2^{mn}}$$

ΣUM uses user flags 0 and 1, as well as data registers R00, to R04 - which therefore cannot be used by the user function to program the general term.

Note. The partial results will be shown in the display during the execution of the function. This should show a converging process, with the numbers approaching the final result at each iteration. You can always stop the function using the R/S key if there is not a converging progression.

Finite and Infinite Products.

PRODX is a MCODE function that calculates the result of either finite or infinite product expressions, starting at an initial index n1 and ending at a final index n2. Use n2=0 for infinite products. These parameters are expected in the Y registers, in the form "n1,00(n2)". An argument x is also expected in the X register – which should be void if not used by the user code multiplicand function. It is the equivalent to a single series, but using multiplications instead of additions.

$$P(x) = (u_k \cdot u_{k+1} \cdot u_{k+2} \cdot u_{k+3} \cdot \ldots\ldots\ldots) = \prod u_n(x) \quad \text{for } n >= k$$

For the infinite case, the final result is obtained when the contribution of the multiplicands is negligible, i.e. when the multiplicand is equal to one (or very close to it). Typically, the convergence is very slow, so this function will establish that condition on the values rounded to the current decimal digits, therefore the display settings is very important for PRODX.

Let's make it clear that this function is not meant to be used as a general technique, due to the extreme low convergence of the majority of the practical cases – even with small FIX settings the final result won't have the displayed number of decimals accurate either!

Example: Obtain the value of $\pi/2$ using the Wallis' product formula:

$$\frac{\pi}{2} = \frac{2}{1}\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdot\frac{8}{7}\cdot\frac{8}{9}\cdots = \prod_{n=1}^{\infty}\left(\frac{4\cdot n^2}{4\cdot n^2-1}\right)$$

Which we'll program as follows: (note that we'll ignore the argument but it must be entered regardless).

```
01 LBL "WPI"
02  RDN
03  ST+ X
04  X^2
05  RCL X
06  E
07  -
08  /
09  END
```

And type the following: 1, ENTER, PRODX "WPI" =>1.570242 ;

As another example let's attempt to calculate the Gamma function using the Schlömilch representation:

$$\Gamma(z) = \frac{e^{-\gamma z}}{z}\prod_{n=1}^{\infty}\left(1+\frac{z}{n}\right)^{-1} e^{\frac{z}{n}}.$$

The program used for PRODX and the final adjustment routine are listed below:

```
01 LBL "GAMX
02  1
03  X<>Y
04  "GX"
05  PRODX
06  RCL 01
07  GEU
08  *
09  E^X
10  /
10  RCL 01
11  /
12  RTN
13  LBL "GX"
14  X<>Y
15  /
16  E^X
17  LASTX
18  1
19  +
20  /
21  END
```

Using x =1, and setting 6 decimal places (FIX 6) we type:

1, ENTER, XEQ "GAMX" =>0.999501 ; quite a poor accuracy indeed.

All in all, just an academic interest but not really a practical method to say the least.

Simple Series with Recursive General Term.

ΣUMR calculates a single series sum (without any argument x) when Un is given as a recurrence expression with an initial known value, that is: u_k is given and $u_{n+1} = f(u_n;n)$ where f is a known function.

A program which computes un+1 = f(Un, n) is required as a subroutine. It is done assuming Un is in X-register and n is in Y-register upon entry. The module includes the routine "S**R**" as example for this case, with $u_1 = 1$ and $u_{n+1} = u_n/(n+1)$

ALPHA, "SR", ALPHA, 1, ENTER^, XEQ "ΣUMR" -> X = 1.291285997= R01 (in 7 seconds)

Note that on both cases the initial index can also be zero, assuming that's compatible with the definition of Un, which adds more flexibility to the routine. In both cases the function needs to be programmed under a global label, and *its name is expected to be in the ALPHA register when the routines are called.*

STACK	INPUTS	OUTPUTS		STACK	INPUTS	OUTPUTS
Y	n	/		Y	n	/
X	x	ΣUM		X	U_n	ΣUMR
ALPHA	F.NAME	F.NAME		ALPHA	F.NAME	F.NAME

Simple Series with Euler Transformation.

ΣUME calculates the same sum making use of the *Euler Transformation* to accelerate the convergence of alternating series: $S = u_0 - u_1 + u_2 - u_3 + ... + (-1)^n u_n + ...$

The sum is re-written in function of the binomial coefficients, $C_n^p = n! / (p! (n-p)!)$ as follows:

$S = a_0/2 + (C_1^1 a_0 - C_1^0 a_1)/2^2 + (C_2^2 a_0 - C_2^1 a_1 + C_2^0 a_2)/2^3 + (C_3^3 a_0 - C_3^2 a_1 + C_3^1 a_2 - C_3^0 a_3)/2^4 + ...$

This may produce superb acceleration but it can also fail.

A program which computes Un = |f(n)|is required as a subroutine, but *without the alternating sign.* It is done assuming n is in X-register upon entry. The module includes the routine "**SE**" as example for this case, with $f(n) = (n+1)^{-1/2}$

ALPHA, "SE", ALPHA, XEQ "ΣUME" -> X = 0.6048986431 = R01

For this particular example the error is -3 10-10 (!), and only 28 terms are calculated (taking about 6.5 minutes to converge). Without an acceleration method, more than 1,018 terms would be necessary to achieve the same accuracy... which execution time would be much greater than the age of the Universe.

STACK	INPUTS	OUTPUTS
X	/	ΣUME
ALPHA	F.NAME	F.NAME

Double and Triple Series.

ΣΣUM is a MCODE function that calculates a double series sum, or function of an argument x –
that is: $S(x) = \Sigma\Sigma \{u_{n;m}(x)\}$; for $n >= n_0$; $m >= m_0$

The two initial indices are expected to be in the stack and the label name must be in the ALPHA register as well.

STACK	INPUTS	OUTPUTS
Z	m_0	/
Y	n_0	/
X	x	ΣΣ
ALPHA	F.NAME	F.NAME

As with the single sum case before, the general term needs to be programmed under a separate subroutine using a global label. This assumes that the argument x is in the X register, "n" is in the Y-register and "m" in the Z-register upon entry.

The program uses user flags 0, 1, and 2, as well as data registers R00 to R04 - which therefore should not be used in the definition of the general term.

The module includes the routine "**SS**" as example for this case, with the expression: $f(n;m) = 1 / (n^n m!)$

ALPHA, "SS", ALPHA, 1, ENTER^, ENTER^, XEQ "ΣΣUM" -> X = 2.218793264 (in 45")

ΣΣΣUM is a MCODE function that calculates a triple series sum, or function of an argument x –
that is: $S(x) = \Sigma\Sigma\Sigma \{u_{n;m;p}(x)\}$; for $n >= n_0$; $m >= m_0$; $p >= p_0$

This assumes the argument in the X-registers, and the three indexes (n, m, p) in the Y,Z,T stack registers upon entry.

STACK	INPUTS	OUTPUTS
T	p_0	/
Z	m_0	/
Y	n_0	/
X	x	ΣΣΣ
ALPHA	F.NAME	F.NAME

As before, the general term needs to be programmed under a separate subroutine using a global label. The three initial indices are expected to be in the stack and the label name must be in the ALPHA register as well. The module includes the routine "**SSS**" as example for this case, with the expression: $f(n;m;p) = 1 / (n^n m! (p!)^2)$

ALPHA, "SSS", ALPHA, 1, ENTER^, ENTER^, ENTER^, XEQ "ΣΣΣUM" ->X = 2.839135243 (3 min 15s)

With an error ε =-7 E-9

Multiple Series

NΣUM0 calculates a multiple series sum, with k internal summations that all start in zero - that is:

$$S = \Sigma\Sigma...\Sigma\ [\ U(n_1;n_2;\ .....\ ;n_k)\ ]\ \ with\ \ n_1>=0\ ;\ \ n_2>=0\ ;\ .....\ ;\ \ n_k>=0$$

Obviously now the number of indices will be in the X register, and there are no initial indices – which are assumed to be zero. This may need you to re-write the expression of the general term to make it compatible with this condition.

This is the most limiting requirement for this program, which is not suitable for cases that have mutual dependencies between the initial indexes.

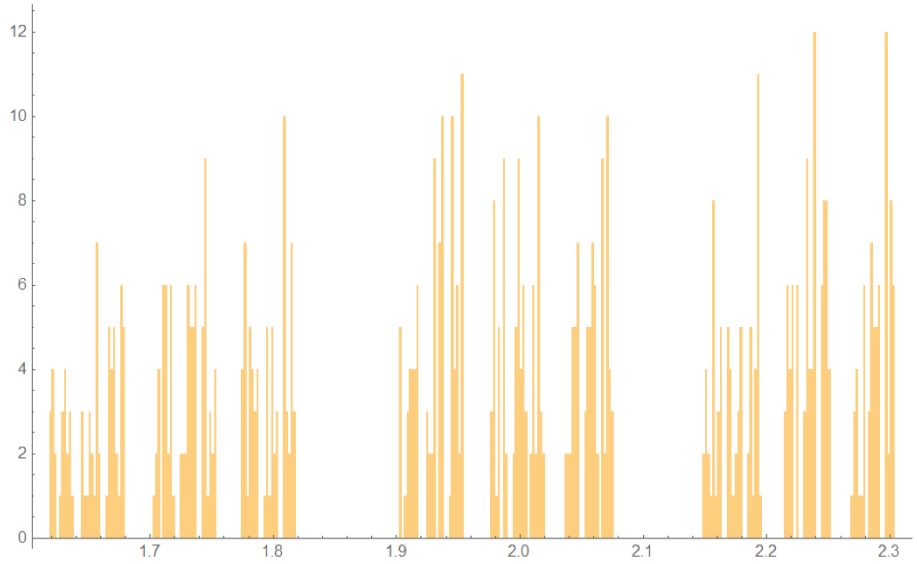
Here too, the general term needs to be programmed under a separate subroutine using a global label, which needs to be entered in ALPHA. This assumes on entry that"n1" is in the R01 register,"n2" in the R02 register, "n3" in R03, and successively so until completing the number of variables.

STACK	INPUTS	OUTPUTS
X	k	NΣ0
ALPHA	F.NAME	/

Only the synthetic registers {M,N,O} are used by the program. The module includes the routine "**NT**" as example for this case, with the expression used in the triple case: $f(n;m;p) = 1 / (n^n\ m!\ (p!)^2)$.

We therefore need to change it to start at the zero indexes for the three variables, i.e. must make a change of arguments to reduce to the standard: n >= 0 ; m >= 0 ; p >= 0 by replacing n with (n+1) ; m with (m+1) ; p with (p+1): $f(n;m;p) = 1 / \{(n+1)^{(n+1)}(m+1)!\ [(p+1)!]^2\}$

ALPHA, "NT", ALPHA, 3, XEQ "NΣUM0" ->X = 2.839135243 = R04 (8 min 39 s)



Nested Radicals

FNRM and **INRM** are MCODE functions to calculate finite and infinite Nested Radicals or root-order m. The definition of the radical is given in a user-provided function under a global label, to generate the n terms that contribute to the radical R(n).

- For the finite case the calculation ends when all the terms are provided and used in the radical.
- For the infinite case, a series of finite radicals of increasing sizes are computed until two of them are equal. This means R(n) = R(n+1), for a given n large enough.

An initial size n0 needs to be provided by the user, which ideally is a balance between the radical size and the number of subsequent radicals to calculate: the larger the radical the longer calculation time, but the less number of radicals likely to calculate.

STACK	INPUTS	OUTPUTS
Y	k	
X	no	NR
ALPHA	F.NAME	/

FNRM and **INRM** use data registers {R00 – R05} as well as user flags UF 001 and UF 01. Refrain from using these resources in the definition of your functions. Note that both the root order m and the term n are available for your user function to use – even if normally only n is used. This allows for more elaborate expressions in the definitions.

For example, let's calculate the value of an infinite nested radical with f(n) = n, as per the expression below:

$$\sqrt{n+\sqrt{n+\sqrt{n+\sqrt{n+\cdots}}}} = \tfrac{1}{2}\left(1+\sqrt{1+4n}\right)$$

For the case n=1 this happens to be the golden ratio Φ = ½ (1+sqr(5)

A trivial user program like this: {LBL PH, 1, RTN}, say we set FIX 9 and then we type:

2, ENTER^ 4, XEQ "INRM"_ PH" => 1.618033989

Using cubic roots instead we'll obtain the "Plastic" Constant:

3, ENTER^, 4, XEQ "INRM"_"PH => 1.324717957

Example 2.Calculate the cubic and quartic root nested radicals for the function F(n) = n^4

Using n0=4 and the trivial user function {LBL "NR4", X^2, X^2, END} we get:

4, ENTER^, 4, XEQ "INRM"_"NR4" =>1.325706774 quartic case
3, ENTER^, 4, XEQ "INRM"_"NR4" =>1.551416993 cubic case

Example 3. Calculate the square nested radical for the function F(n) = n {LBL "NR1", RTN"}

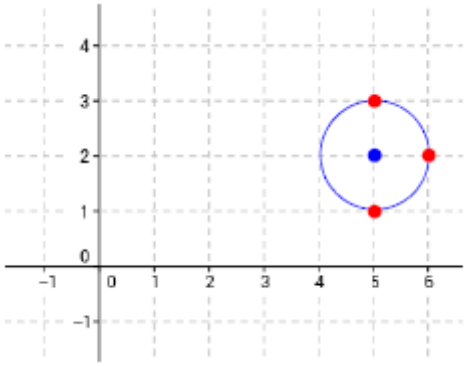
2, ENTER^, 4, 4, XEQ "INRM"_"NR1" =>1.757932757

2 –Areas of Polygons

The second section of the module includes several MCODE functions for triangles, cyclic quadrilaterals and even non-regular polygons.

- **CIRCL** calculates the radius of a circle passing thru three data points, using the point x,y coordinates. The values are expected to be stored in R01-R07. Besides that, it'll also return in the Y-register the area of the circumscribed triangle defined by the three points.

  Example: Calculate the radius of the circle passing thru P(5,1), Q(6,2), and R(5,3)



The results are:

XEQ "CIRCL" => r=1,000000000,
X<>Y => A=1,000000000

The input sequence starts with the abscissa of P1 in R01.

Note that you can use the routines **IN** and **INPUT** available in the SandMath to populate the registers automatically.

- **HERON** calculates the area of a triangle knowing its three sides, using Heron's formula. Just enter the sides values in the stack, and execute the function. The result is stored in X, with the original side saved in LastX. The rest of the stack is unchanged.

  Let the triangle ABC with 3 known sides { a , b , c } and s = (a+b+c)/2 the semi-perimeter

  Heron's formula is: $\text{Area} = [\, s(s-a)(s-b)(s-c)\,]1/2$

  Example: a = 2, b = 3, c = 4
  Type: 2, ENTER^, 3, ENTER^, 4, XEQ "HERON" => Area = 2.904737510

  Note: the function **CIRCL** described above makes use of the HERON formula internally after it first calculates the triangle sides from the point coordinates.

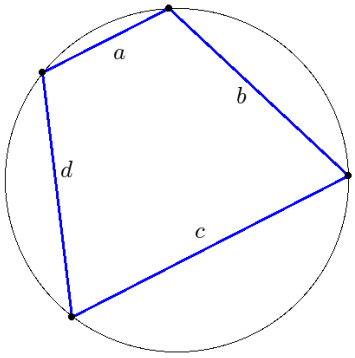
- **BRHM** is related to it, but the calculation for the area of the cyclic quadrilateral - using Brhamagupta's formula. Just enter the four values in the stack and execute the function. The result is stored in X, with the original side saved in LastX. The rest of the stack is unchanged.

Let a, b, c, and d be its sides lengths, and the semi-perimeter s = (a + b + c + d)/2 .The area A of the cyclic quadrilaterals:

$A = [(s-a).(s-b).(s-c).(s-d).]1/2$

Example: a = 4 , b = 5 , c = 6 , d = 7

Type: 4, ENTER^, 5, ENTER^, 6, ENTER^, 7,
XEQ "BRHM" => Area = 28.98275349



- **PG1** Calculates the area of a regular polygon when its side length is known. Input parameters are the number of sides in Y, and the side length in X. The result is left in the X register.

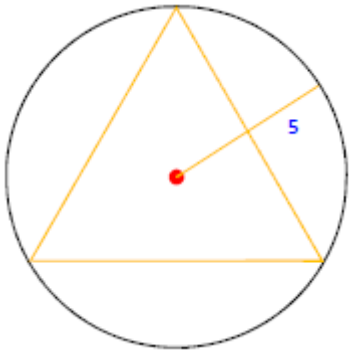
  Example: calculate the area of a triangle with side length a=5 m

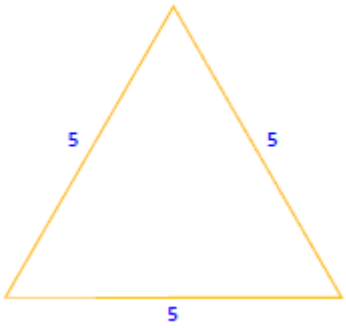
  3, ENTER^, 5, XEQ "PG1" -> 10.83 m^2

- **PG2** Performs the same calculation but using the radius of the circumscribed circle instead of the side length. Same order of parameters for input, with the number of sides in Y.

  Example: calculate the area of a triangle circumscribed in a circle with radius r=5 m

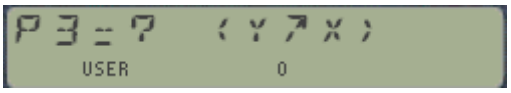
  3, ENTER^, 5, XEQ "PG2" -> 32.48 m^2





- Finally, the Routine "**3PNTS**" is a FOCAL driver for functions **CIRCLE**. You can use it to enter the coordinates of the three points into data registers R01-R06, presented as three screens with the prompts:

...


...


  Once this is accomplished the program offers you a choice for the value to calculate next, either the triangle area or the circle radius. You can Also press [E] to start over with a new set of three points.

  If used on the example listed above, it returns the following results:

and


Convex Cyclic Polygons.

And what about non-regular polygons, you may wonder? Well, those are the subject of the following set of FOCAL routines about to be described.

Programs **"CCPA"** and **"CPLA"** compute the area A and the circumradius R of a convex cyclic polygon assuming all the sides lengths are known. Moreover, we also assume that the *center* of the circumcircle is *inside* the polygon.

If $\{a_1 , a_2 , ... , a_n\}$ are the sides lengths and $\{ \mu_1 , \mu_2 , ... , \mu_n \}$ are the corresponding central angles, we have to solve the system of (n+1) equations:

$$2.R \sin \mu_1/2 = a_1$$
$$2.R \sin \mu_2/2 = a_2$$
$$\ldots\ldots\ldots\ldots\ldots$$
$$2.R \sin \mu_n/2 = a_n$$
$$\mu_1 + \mu_2 + \ldots + \mu_n = 360°$$

Performing a few substitutions lead to an expression with the radius as single unknown, to be resolved iteratively using any root-finding method:

$$\text{asin}\,(a_1/2R) + \text{asin}\,(a_2/2R) + \ldots\ldots\ldots\ldots\ldots + \text{asin}\,(a_n/2R) = 180°$$

After finding R, the Area is given by : $A = (R^2/2)\,(\text{Sin}\,a_1 + \text{Sin}\,a_2 + \ldots\ldots\ldots + \text{Sin}\,a_n)$

There are two versions included in the module – **"CPLA"** uses the SOLVE function in the Advantage and **"CCPA"** uses a built-in root finder based on Newton's method. Each one has advantages and shortcomings, as usual.

Drivers for Data Entry.

The routines expect the sides of the polynomial already stored in contiguous data registers, and the control word "bbb.eee" in the X register before you call the routine. For your convenience, a driver routine is also included that prompts for the side values and does the storing for you, Using **"CPLA+"** or **"CCPA+",** all you need to do is enter the values at each prompt, and once completed it'll direct the execution to the corresponding data engine downstream.

Example. Find the area of a convex cyclic polygon with sides: 4 , 5 , 6 , 7 , 8 , 9 , 10

XEQ "CPLA+"	"N=?"
7, R/S	"d1=?"
4, R/S	"d2=?"
5, R/S	"d3=?"
6, R/S	"d4=?"
7, R/S	"d5=?"
8, R/S	"d6=?"
9, R/S	"d7=?"
10, R/S	-> 174.6757940 the area, and
X<>Y	-> 8.143816980 the radius

Diagonal Lengths.

CPLD calculates the diagonal lengths of a convex cyclic polygon with its side lengths known. The method used involves solving a linear system of (n-3) equations with (n-3) unknowns, which is solved by successive approximations. The convergence is linear only – which results in relative longer execution times.

The iteration starts with all diagonals lengths = 0, which is very simplistic. The successive sums of the differences between 2 consecutive approximations (in absolute values) are displayed. They should tend to zero, however, the termination criterion may lead to an infinite loop. Since there are 319 registers at most, "CPLD" can find the diagonals lengths of a 159-gon – but the execution time will not be small without an emulator.

CPLD expects the number of sides in R00, and the sides of the polynomial already stored in contiguous data registers starting with R01 until Rn+1. Then you must provide the control words "bbb.eee" indicating the location of the data registers that store the side lengths.

STACK	INPUT	OUTPUTS
X	/	bbb.eee

For your convenience, a driver routine is also included that prompts for the side values and does the storing for you, Using "**CPLD+"** all you need to do is enter the values at each prompt, and once completed it'll direct the execution to CPLD downstream.

Example: Find the diagonals lengths of a convex cyclic hexagon with sides: 4 , 5 , 6 , 7 , 8 , 9

XEQ "CPLD+"	"N=?"
6, R/S	"a1=?"
4, R/S	"a2=?"
5, R/S	"a3=?"
6, R/S	"a4=?"
7, R/S	"a5=?"
8, R/S	"a6=?"
9, R/S	shows estimations… -> convergence
	d8=8.46278437
R/S	d9=12.12358502
R/S	d10=12.97690535

There are in fact n(n-3)/2 diagonals whose lengths may be obtained by "rotating" the sides lengths in registers R01 to R06 and we have similarly: n(n-3)/2 = 9 if n = 6

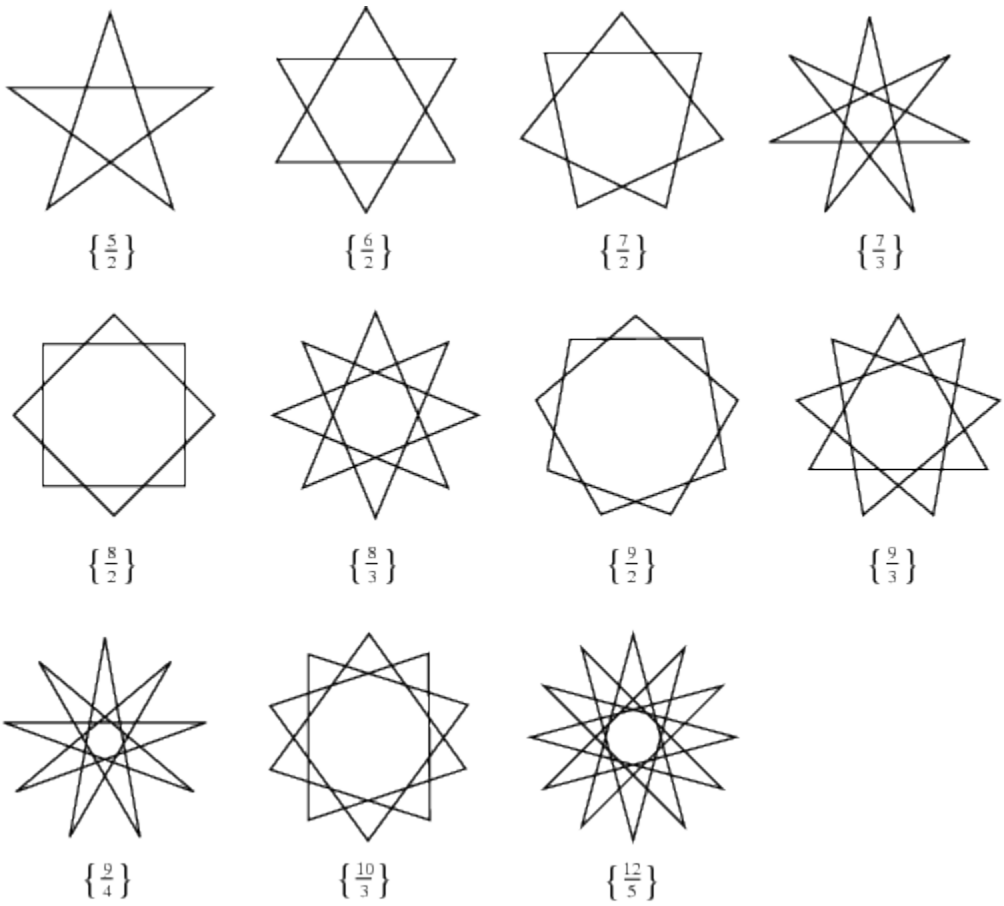
d_4 = 9.998827970	d_7 = 11.30861231
d_5 = 13.01214483	d_8 = 13.06010803
d_6 = 11.49035918	d_9 = 12.32872367

Finally, the MCODE function **DIAG** is used internally by **CPLD** to speed-up the calculations. It computes the following expression, assuming x , y , z , t are in registers X , Y , Z , T upon entry

$$\text{SQRT}\left[\{ x.y\,(z^2 + t^2) + z.t\,(x^2 + y^2) \} / (x.y + z.t) \right]$$

Regular Star Polygons

A star polygon {n/k}, with n,k positive integers, is a figure formed by connecting with straight lines every "k-th" point out of n regularly spaced points lying on a circumference. The number k is called the polygon density of the star polygon. Without loss of generality, take k<n/2. The star polygons were first systematically studied by Thomas Bradwardine.



If k=1, a regular polygon {n} is obtained. Special cases of {n/k} include {5/2} (the pentagram), {6/2} (the hexagram, or star of David), {8/2} (the star of Lakshmi), {8/3} (the octagram), {10/3} (the decagram), and {12/5} (the dodecagram).

"**STLA**" computes the area A, the perimeter P, the in-radius r and the circumradius R of a regular star polygon { n / k } from its edge length a

Formulae:

$A = n R^2 \text{ Sin } (180°/n) \text{ Cos } (180° k/n) / \text{ Cos } [180°(k-1)/n]$
$a = 2.R \text{ Sin } (180° k/n)$
$r = R \text{ Cos } (180° k/n)$
$P = 2.A / r$

The table on the left shows the input and output Required by the program – easy does it!

STACK	INPUTS	OUTPUTS
T	/	R
Z	a	r
Y	n	P
X	k < n/2	A

Examples:

- a = 1 , n = 5 , k = 2

1 ENTER^, 5 ENTER^, 2	XEQ "STLA"->	A = 0.310270701
	RDN	P = 3.819660113
	RDN	r = 0.162459848
	RDN	R = 0.525731112

- a = 1 , n = 10 , k = 3

1 ENTER^, 10 ENTER^, 3, R/S	->	A = 0.857567126
	RDN	P = 4.721359547
	RDN	r = 0.363271264
	RDN	R = 0.618033989

- a = π , n = 41 , k = 13

PI ENTER^, 41, ENTER^, 13, R/S	->	A = 9.855571194
	RDN	P = 19.37713086
	RDN	r = 1.017237409
	RDN	R = 1.871409374

This program works in all angular modes, however, DEG mode should be preferable.

If k = 1, we get the convex regular n-gon. For instance, with a = 1 , n = 5 , k = 1, "STLA" returns what corresponds to the regular pentagon.

A = 1.720477401
P = 5
r = 0.688190960
R = 0.850650808

Driver Program.

Here too you have a convenient driver program to guide you thru the data entry process: program **"STLA+"** will prompt for the input values and will present the results sequentially after the calculations are done.

, 

, 

, 

, etc…

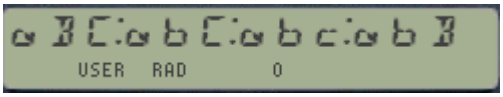
$$F(x,v) = \sum_{k=0}^{n} \sum_{m1=0}^{k} \sum_{m2=0}^{k} \alpha_{m1,m2,k} R^k y^{k-m1} (1-y)^{m1} z^{k-m2} (1-z)^{m2}$$

1 –Triangle Solutions

The core routines to solve plane, hyperbolic and spherical triangles were written by JM Baillard, and are grouped here under three "driver" programs for convenience. The three drivers (**TRIA**, **TRIH**, and **TRIH**) basically provide menu-based selection choices for the specific case, depending on which elements of the triangle are known (sides or angles).

They work in all angular modes but angles must be entered as decimals. The standard triangle notation is employed (lower-case for sides, upper case for angles; A opposite a ... etc.)

Rectangular Triangles.

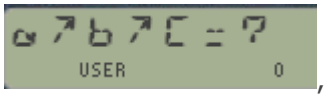


TRIA offers the following 4 choices:

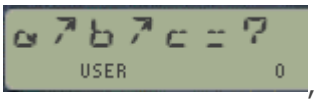
- One Side, two adjacent angles (with other sides)
- Two sides and included angle between them
- Three sides
- Two sides, and opposite angle

These are accessible pressing the top keys [**A**] to [**D**]. Additionally, pressing R/S at this point or [**E**] at any moment will bring back the same options menu again.

Once the case is selected using the top keys, the program presents a prompt to enter the known elements:



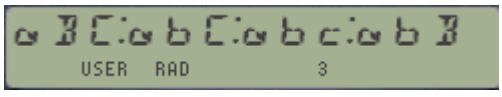
, or:

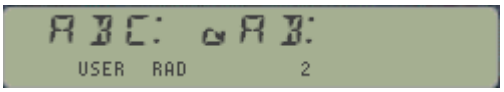

, or:


, etc.

Formulas for plane triangles: a/sinA = b/sinB = c/sinC ;
a2 = b2 + c2 - 2 b.c.cosA and 2 similar relations ;
Area = b.c.(sinA)/2

Hyperbolic and Spherical Triangles.



;


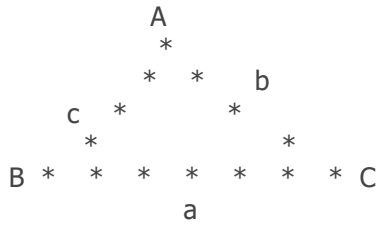
TRIH and **TRIS** add to the cases shown before two more combinations of known elements, as follows:

- Three angles
- One side, the opposite angle and another angle

These are accessible pressing the top keys: either [A]/[B] for the first and [C]/[D] for the second. Also pressing R/S at this prompt or [E] at any time will bring the main menu back for the selection.

Formulas for Hyperbolic triangles:

sinh a / sin A = sinh b / sin B = sinh c / sin C
cosh a = cosh b . cosh c - sinh b . sinh c . cos A
cos A = - cos B . cos C + sin B . sin C . cosh a
c = arctanh (cos A tanh b) + arctanh (cos B tanh a)



we must have A + B + C < 180°

Formulas for Spherical Triangles:

sin a / sin A = sin b / sin B = sin c / sin C
cos a = cos b . cos c + sin b . sin c . cos A
cos A = -cos B . cos C + sin B . sin C . cos a
c = arctan (cos A . tan b) + arctan (cos B. tan a) (modulo 180°)

This last formula is used in cases n°5 and n°6.

Other formulae can be used, for example:

tan c/2 = (tan (a-b)/2).(sin (A+B)/2) / (sin (A-B)/2) (F1)
tan c/2 = (tan (a+b)/2).(cos (A+B)/2) / (cos (A-B)/2) (F2)

but (F1) cannot be applied if a = b & A = B and (F2) doesn't work if a + b = A + B = 180°

Examples.